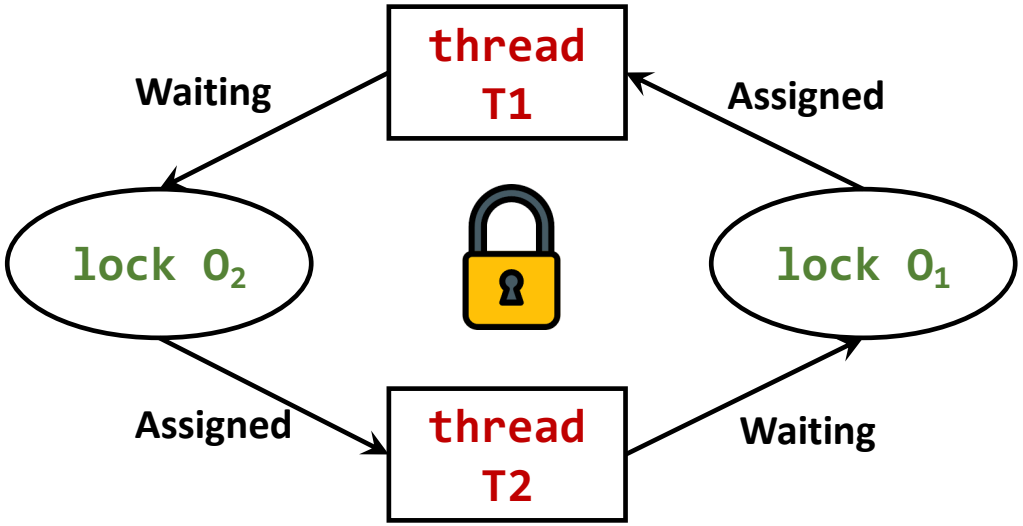
When Threads Meet Interrupts: Effective Static Detection of Interrupt-Based Deadlocks in Linux

Chengfeng Ye, Yuandao Cai, Charles Zhang
The Hong Kong University of Science and Technology

Traditional Deadlock

- Each thread inside a group attempts to acquire the lock already held by others



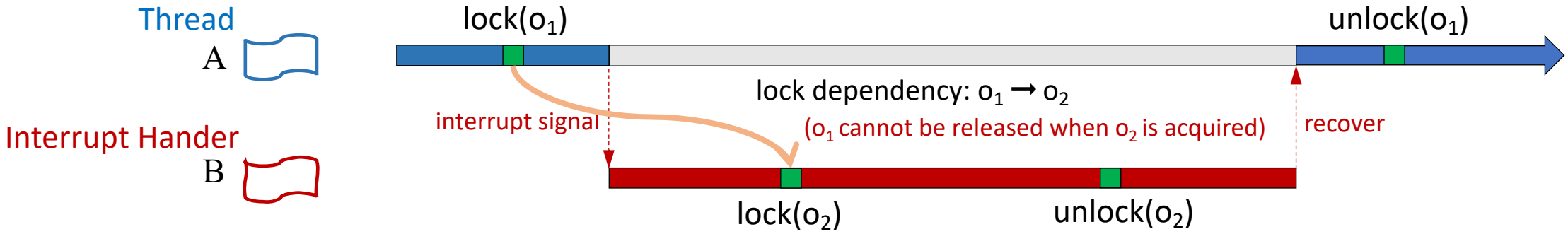

```
thread T1
  lock(O1)
  ...
  lock(O2)
```

```
thread T2
  lock(O2)
  ...
  lock(O1)
```

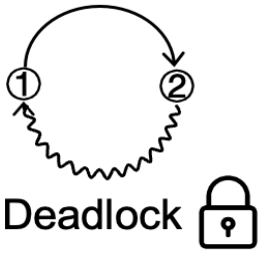
- Characterized by **cyclic lock dependency** introduced by **nested lock acquisition**

Interrupt-Based Deadlock in Kernel

- **Interrupt preemption** could introduce **Interrupt Lock Dependency**
  - a preempted thread cannot be rescheduled until interrupt finishes
  - any locks held by a preempted thread cannot be released when interrupt is executing



- Interrupt lock dependency could lead to **Interrupt-Based Deadlock**
  - forming **lock dependency cycle** on exclusive lock (e.g., spinlock, rwlock)
  - at least lock dependency is introduced by interrupt preemption
  - blocking the whole CPU execution under the interrupt context



A Real-World Example in Linux

- *drivers/media/video/exynos/mfc/s5p_mfc.c*

```
166 static void s5p_mfc_watchdog_worker(...) {
167   ......
184   spin_lock_irqsave(&dev->irqlock, flags);
185                          ①
186   s5p_mfc_clock_off();
187
188   for (i = 0; i < MFC_NUM_CONTEXTS; i++) {
189     ......
195     clear_work_bit(ctx);
196     wake_up_ctx(...);
197   }
198   ......
199 }
```

```
49 void clear_work_bit(...) {
50   struct s5p_mfc_dev *dev = ctx->dev;
51                          ②
52   spin_lock(&dev->condlock);
53   __clear_bit(ctx->num, ...);
54   spin_unlock(&dev->condlock);
55 }
```

Nested Lock Dependency: ① → ② **irqlock** **condlock**

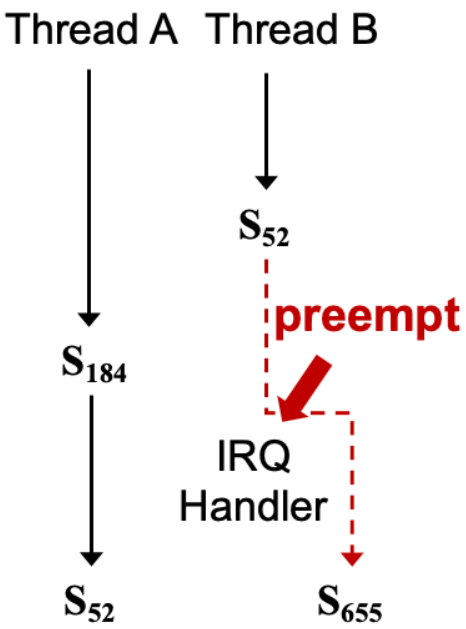
preempt

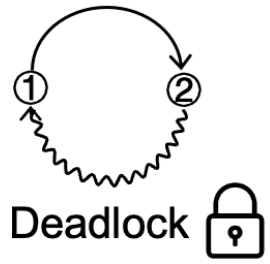
```
213 static int enc_post_frame_start(...) {
214   ......
299   if (!src_ready  ...)
300     clear_work_bit(ctx);
301
302   return 0;
303 }
```

```
49 void clear_work_bit(...) {
50   struct s5p_mfc_dev *dev = ctx->dev;
51                          ②
52   spin_lock(&dev->condlock);
53   __clear_bit(ctx->num, ...);
54   spin_unlock(&dev->condlock);
55 }
```

```
645 static irqreturn_t s5p_mfc_irq(...) {
646   ......
653   /* Reset the timeout watchdog */
654   atomic_set(&dev->watchdog_cnt, 0);
655   spin_lock(&dev->irqlock);
656   ......          ①
763 }
```

Interrupt Lock Dependency: ② ⇝ ① **condlock** **irqlock**

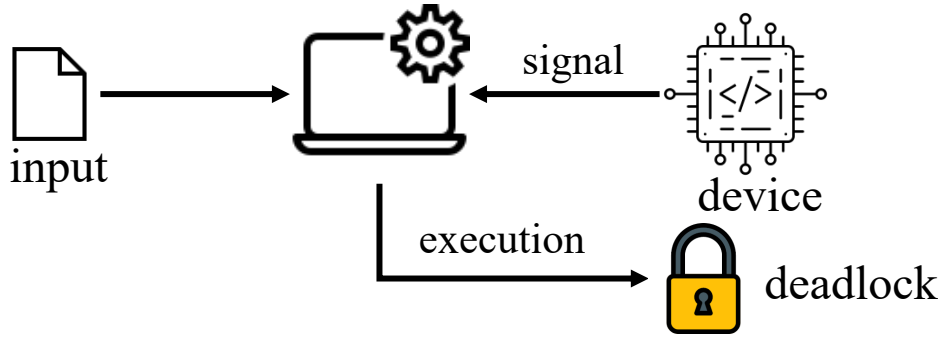




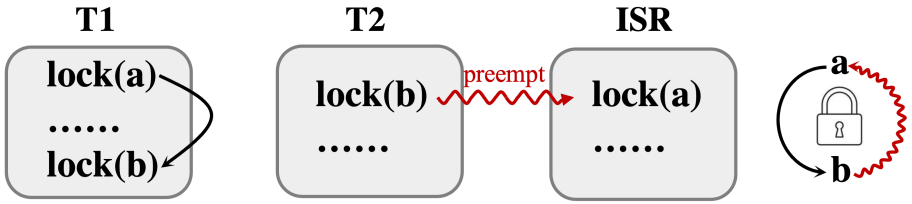
Interrupt-Based Deadlock Detection

- Dynamic Testing **limitation**: rely on testcase and external devices
  - **Lockdep** - Linux runtime locking correctness validator

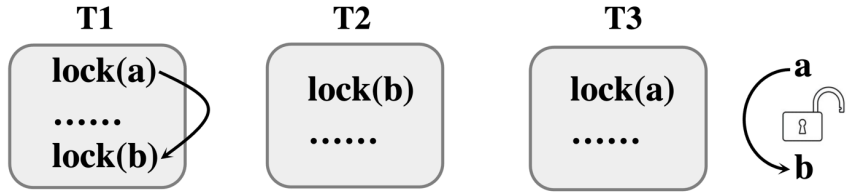
```
Possible interrupt unsafe locking scenario:

      CPU0                    CPU1
      ----                    ----
 lock(&htab->buckets[i].lock);
                              local_irq_disable();
                              lock(&rq->__lock);
                              lock(&htab->buckets[i].lock);
 <Interrupt>
   lock(&rq->__lock);
```



- Static Deadlock Detection **limitation**: cannot detect interrupt-based deadlock
  - **RacerX** (SOSP' 03), **CBMC**(ASE'16), **Peahen** (FSE' 22), **DLOS** (ATC' 22) …



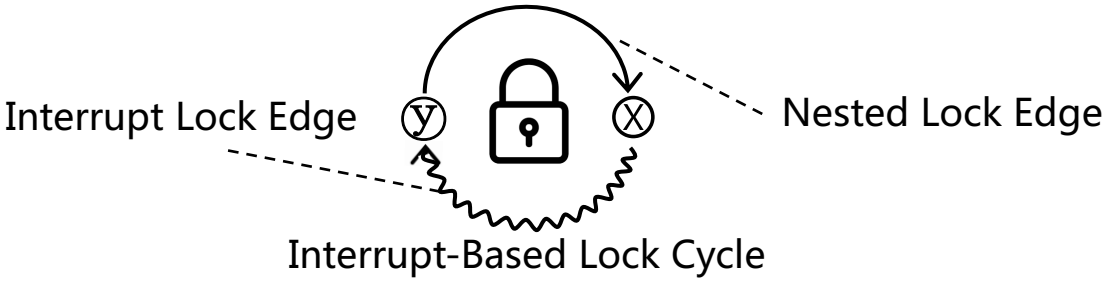
A deadlock program with an ISR



Missed deadlock due to ignoring interrupt preemption

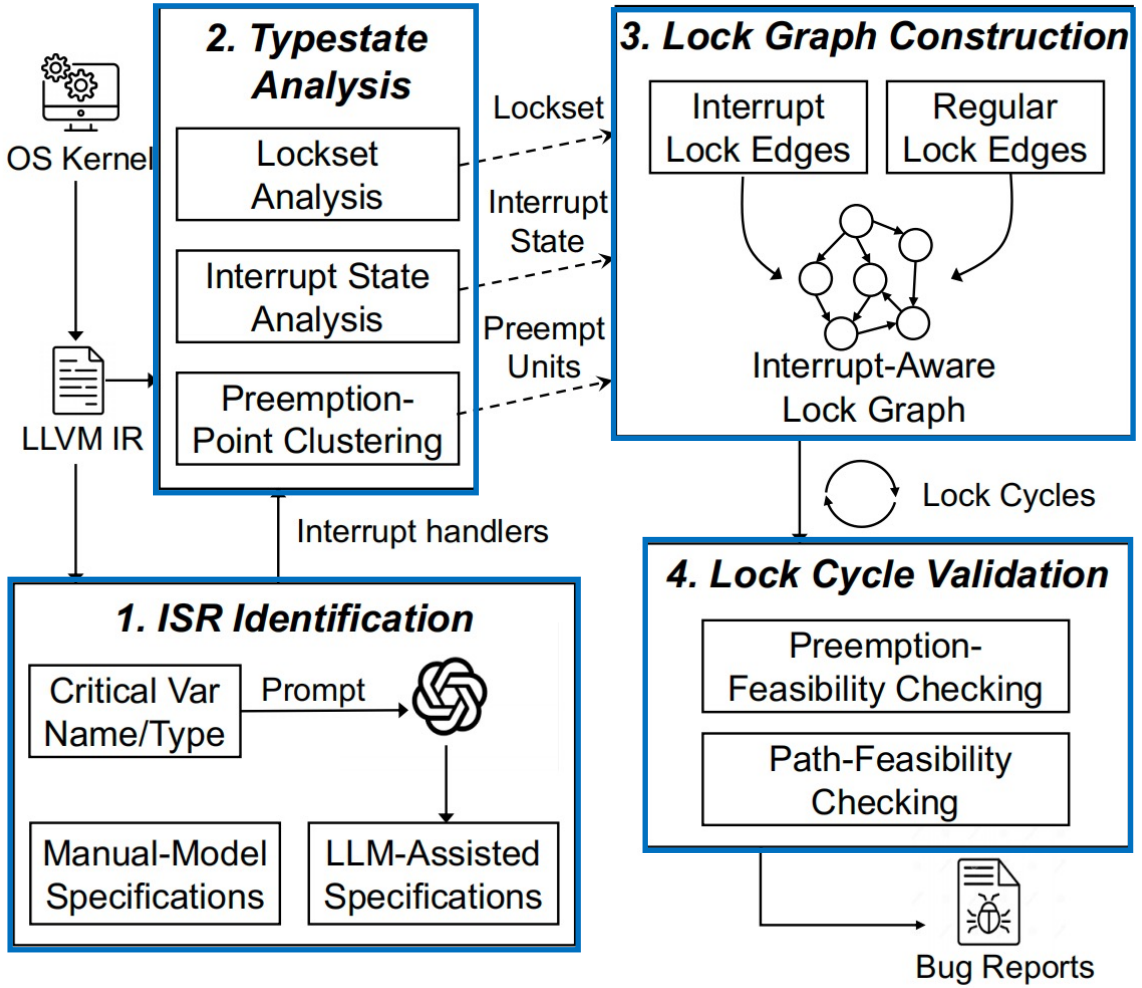
Archerfish : Interrupt-Based Deadlock Static Detection

- **Interrupt-Aware Lock Graph (ILG)**
  - **Interrupt** lock edge
  - **Nested** lock edge



Conceptual Workflow

- (I) identify interrupt service routines (ISRs)
- (II) analyze interrupt state and lock usage
- (III) construct ILG and detect ILCs
- (IV) feasibility validation to reduce false positives



Challenge 1 – Identification of Interrupt Handlers

- (C1) How to accurately identify **interrupt service routine (ISR)** in Linux
  - Various APIs/Interfaces to register an ISR under different subsystem
  - Different types of ISRs have different priority

```
struct net_device_ops ks_wlan_netdev_ops = {
    .ndo_start_xmit = ks_wlan_start_xmit,
    .ndo_open = ks_wlan_open,
    .ndo_stop = ks_wlan_close,
    .ndo_do_ioctl = ks_wlan_netdev_ioctl
    ……
}
```

softirq ISR

(a) Registered via a structure field interface

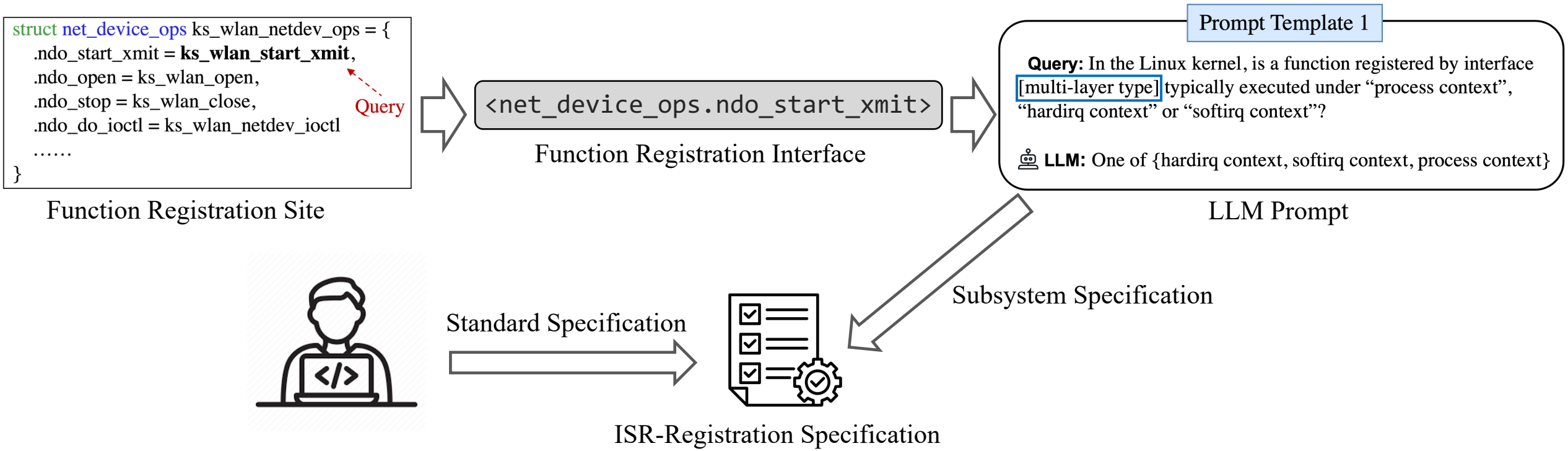
```
request_irq(IO_INTERRUPT, do_cio_interrupt, 0,
"I/O", NULL);
```

hardirq ISR

(b) Registered via an API call

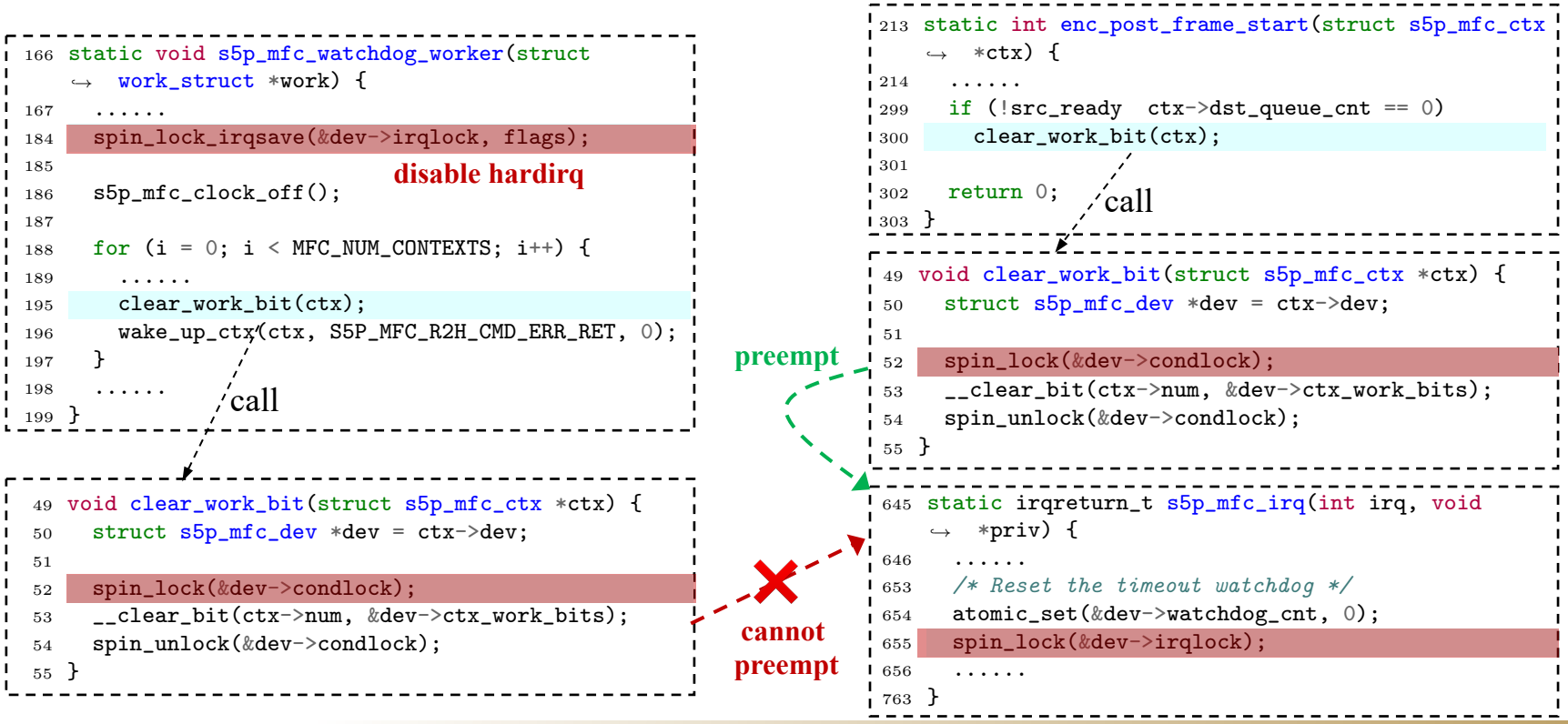
(I) LLM-Assisted ISR Identification

- Idea: supplement **ISR-Registration Specification** with **Large Language Model (LLM)**
  - Collect function registration *interface* and *API* candidates
  - Leverage LLM to decide whether the *interface* or *API* is used for ISR registration



Challenge 2 – Large Concurrent Interleaving Space

- (C2) Interrupt-involving **concurrent interleaving space** is large
  - Each statement could be preempted by any enabled ISRs
  - Enabled/Disabled state of ISR varies under different calling contexts

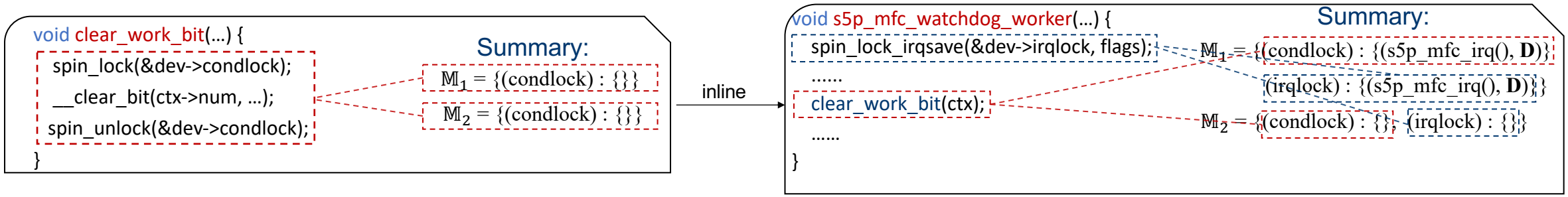


(II) Preemption/Lock Summary Construction

- Idea: treat **critical section** as **unit of interrupt preemption**

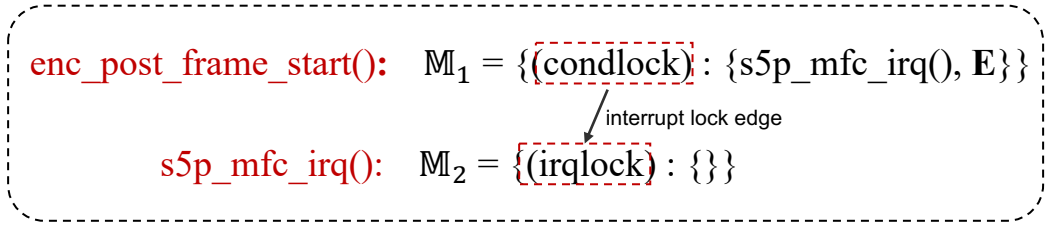
- Flow- and context-sensitive typestate analysis
  - Lockset: $lock \rightarrow ty \in \{\boldsymbol{Acquired}, \boldsymbol{Released}, \perp\}$
  - Interrupt State: $isr \rightarrow st \in \{\boldsymbol{Enabled}, \boldsymbol{Disabled}, \perp\}$

- **Compact summary** construction
  - ($\mathbb{M}_1$) Preempt Unit Summary: $critical\ section \rightarrow \{isr \mid enabled\ inside\ the\ critical\ section\}$
  - ($\mathbb{M}_2$) Locking Summary: $lock \rightarrow \{lock \mid has\ been\ released\}$

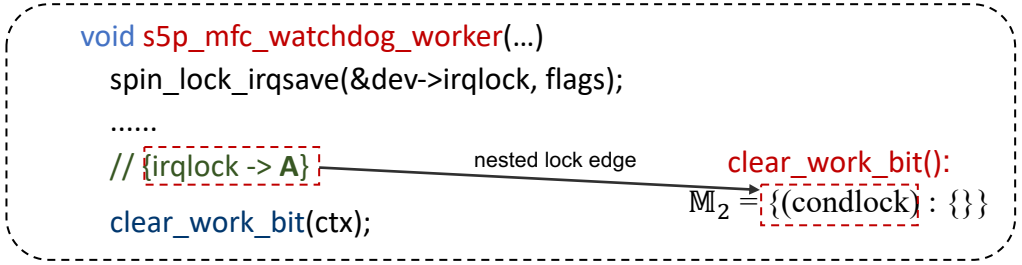


(III) Interrupt Lock Graph (ILG) Construction

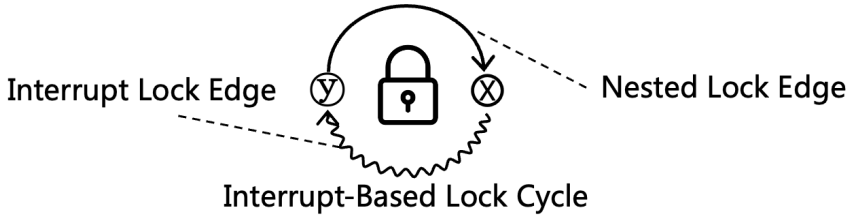
- Interrupt Lock Edge:
  - ∀ critical section o_1 with *isr* enabled
  - ∀ lock o_2 acquired by *isr*
  - Establish an interrupt lock edge $o_1 \rightsquigarrow o_2$



- Nested Lock Edge:
  - ∀ lock o_1 already acquired at lock(o_2)
  - Establish a nested lock edge $o_1 \not\rightarrow o_2$



- ILC Detection
  - Detect ILC with Johnson algorithm[1]



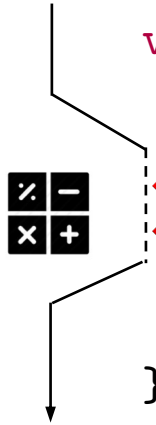
[1], D. B. Finding all the elementary circuits of a directed graph. SIAM Journal on Computing 4, 1 (1975), 77–84.

Challenge 3 – Efficient Feasibility Validation

- (C3) How to **efficiently validate lock dependencies feasibility**
  - Requires heavy-weight analysis (e.g., SMT Solving)
  - Analysis space is large (numerous paths to explore)

```
void vq_thread(...) {
    dev->vq = nullptr;
    ...
    spin_lock(dev->s_lock);
    dev->vq = kmalloc(...);
    spin_unlock(dev->s_lock)
    ...
}
```

preempt

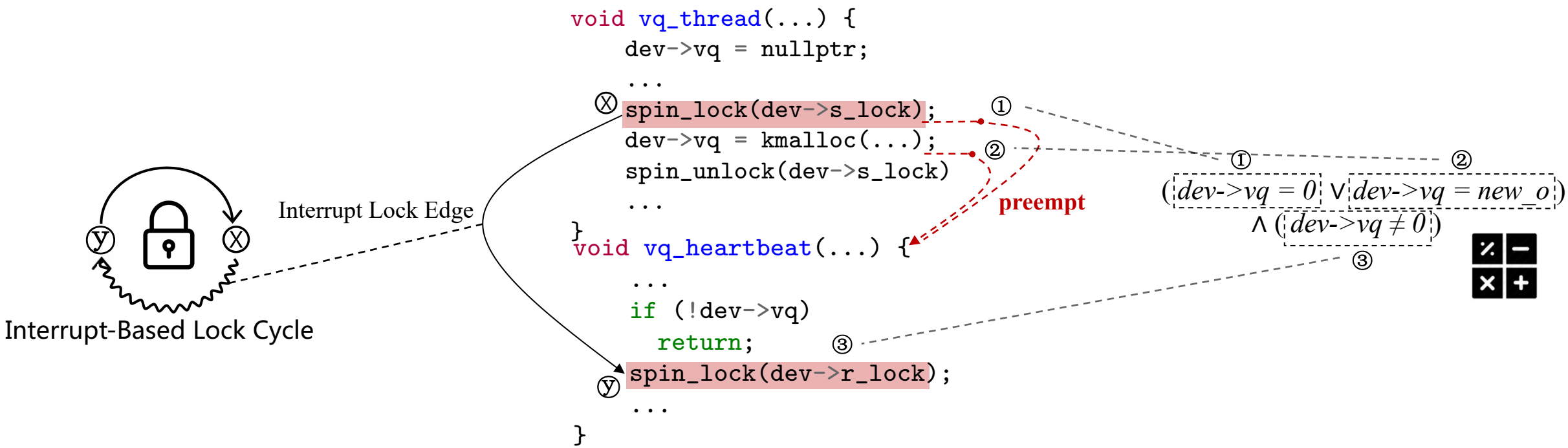


```
void vq_heartbeat(...) {
    ...
    if (!dev->vq)
      return;
    spin_lock(dev->r_lock);
    ...
}
```

infeasible

(IV) Lazy Feasibility Validation

- Idea: only validate the **program paths leading to ILC**
  - Interrupt **happen-before** validation: interrupt preemption only occurs after it is registered
  - Interrupt **path-feasibility** validation: guarded path condition should be feasible



Implementation and Evaluation

- **Archerfish Implementation**
  - static analysis engine: LLVM framework and Z3 solver
  - specification generation: OpenAI ChatGPT-4.0

- **Evaluation on Linux kernel v6.4:**
  1. **Effectiveness**: found **76** new interrupt-based deadlocks with **53** confirmed by the community, and **2** new CVE IDs assigned
  2. **Precision**: overall false positive rate is **49.7%** (75/151)
  3. **Performance**: finish analysis in **68.6** minutes with **38.3** GB peak memory

- **Ablation Study**
  1. **LLM-Assisted Specifications**: identify **1,729 (47.1%)** more ISRs with sampling precision of **79.6%**, help with detect **11** more real bugs
  2. **Summary Design**: save **70.6%** memory and **58.8%** analysis time
  3. **Lazy Feasibility Validation**: reduce **57.2%** false bugs and finish in **7.9** minutes

More details can be found in the paper: https://www.usenix.org/system/files/sec24fall-prepub-514-ye.pdf

Thank you!

Q&A