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

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

https://www.usenix.org/conference/usenixsecurity24/presentation/ye

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

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
{*cyeaa, ycaibb, charlesz*}@*cse.ust.hk*

Abstract

Deadlocking is an unresponsive state of software that arises when threads hold locks while trying to acquire other locks that are already held by other threads, resulting in a circular lock dependency. Interrupt-based deadlocks, a specific and prevalent type of deadlocks that occur within the OS kernel due to interrupt preemption, pose significant risks to system functionality, performance, and security. However, existing static analysis tools focus on resource-based deadlocks without characterizing the interrupt preemption. In this paper, we introduce Archerfish, the first static analysis approach for effectively identifying interrupt-based deadlocks in the large-scale Linux kernel. At its core, Archerfish utilizes an Interrupt-Aware Lock Graph (ILG) to capture both regular and interrupt-related lock dependencies, reducing the deadlock detection problem to graph cycle discovery and refinement. Furthermore, Archerfish incorporates four effective analysis components to construct ILG and refine the deadlock cycles, addressing three core challenges, including the extensive interrupt-involving concurrency space, identifying potential interrupt handlers, and validating the feasibility of deadlock cycles. Our experimental results show that Archerfish can precisely analyze the Linux kernel (19.8 MLoC) in approximately one hour. At the time of writing, we have discovered 76 previously unknown deadlocks, with 53 bugs confirmed, 46 bugs already fixed by the Linux community, and 2 CVE IDs assigned. Notably, those found deadlocks are long-latent, hiding for an average of 9.9 years.

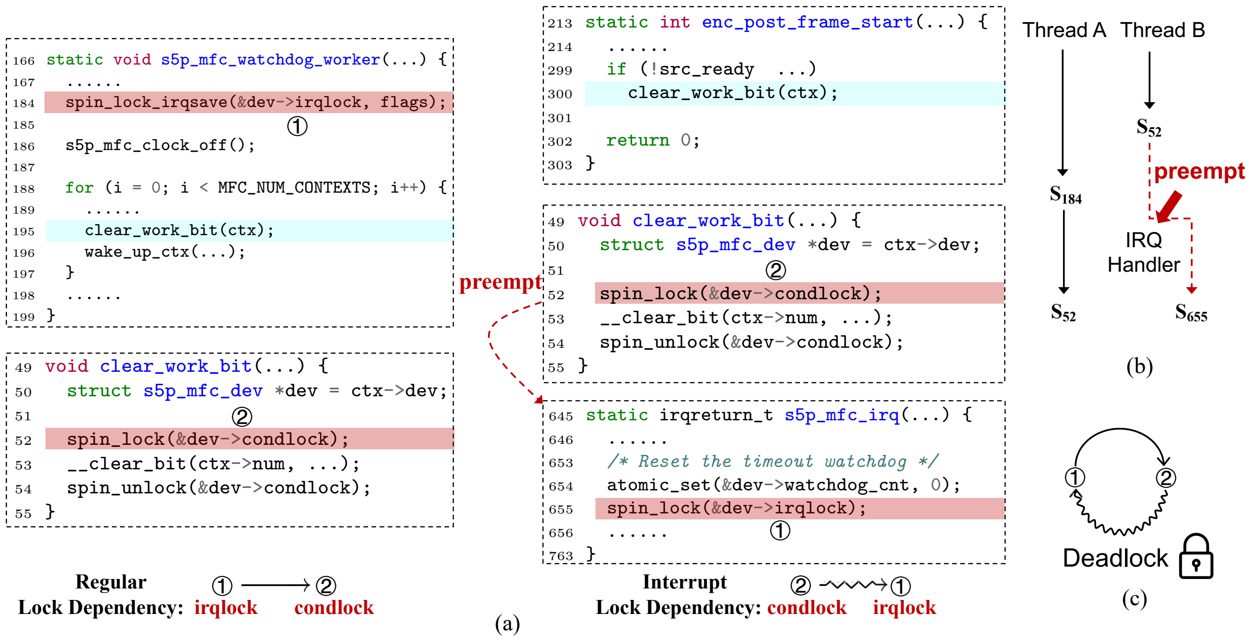
1 Introduction

Deadlocking [7, 15, 21, 33] commonly refers to an unresponsive condition caused by circular lock dependencies, in which, each thread within a group of threads holds a lock while also attempting to acquire another lock that is already held by other threads. In the OS kernel, the interrupt preemption [68] can cause additional lock dependencies, when preempted threads, which have already acquired locks, attempt to acquire additional locks within the interrupt handling procedure [4]. In this paper, we refer to the circular lock dependencies related to the interrupt preemption as *interrupt-based deadlocks*.

There is abundant evidence that deadlocks can significantly hurt the functionality, performance [5], and even security of modern software [50–55]. Furthermore, the interrupt-based deadlocks within the OS kernel can pose a more severe danger as they occur within the interrupt context [43], potentially leading to the entire unresponsive CPU core and even a whole system breakdown [70]. Unfortunately, there are no effective tools for identifying interrupt-based deadlocks, as both the previous static and dynamic approaches have certain limitations.

First, existing static approaches [7, 10, 14, 15, 18, 20–22, 34, 63] primarily focus on detecting resource-based deadlocks [15] caused by thread interleaving, thus overlooking interrupt-based deadlocks. Specifically, most past static analysis tools [14, 15, 18, 34, 35] target deadlock detection in userland software without considering interrupt preemption, a unique feature of the OS kernel. Also, although a few tools [7] aim to detect deadlocks in OS kernel, they still use traditional thread interleaving models by directly treating interrupts as threads, missing lock dependencies caused by interrupt preemption.

Second, dynamic approaches [11, 13, 17, 30, 31, 40, 57, 84, 85] often suffer from low coverage and reliance on the runtime execution environment such as external devices. Specifically, although dynamic fuzzing can detect such deadlocks with the assistance of Lockdep [73], the built-in Linux kernel lock validator, they require high-quality seed inputs or configurations to guide the execution towards a faulty path. Besides, the interrupt preemption [68] significantly enlarges the exploration space of dynamic concurrent execution, degrading the effectiveness. Moreover, the dynamic triggering of interrupts heavily depends on signals generated by hardware devices, which sometimes are hard to be properly estab-



```
166 static void s5p_mfc_watchdog_worker(...) {
167   ......
184   spin_lock_irqsave(&dev->irqlock, flags);
185
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
52   spin_lock(&dev->condlock);
53   __clear_bit(ctx->num, ...);
54   spin_unlock(&dev->condlock);
55 }
```

```
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
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   ......
763 }
```

Figure 1: An eleven-year-old deadlock detected by Archerfish in Linux media subsystem (s denotes a statement).

lished or available [60].

We use an example shown in Figure 1(a), which was introduced in Linux kernel [74] v3.7-rc1 eleven years ago [41] and discovered by us. The example shows an interrupt-based deadlock caused by circular lock dependencies between two threads, Thread A and Thread B. First, in the left-hand side of Figure 1(a), Thread A acquires `&dev->irqlock` ① at Line 184 and then calls the function `clear_work_bit()` at Line 195, which further acquires `&dev->condlock` ② at Line 52. This *regular execution flow*, establishes a **regular lock dependency** from `&dev->irqlock` ① to `&dev->condlock` ②. Second, on the right-hand side of Figure 1(a), Thread B, in the callback function `enc_post_frame_start()`, also calls `clear_work_bit()` at Line 300 and acquires `&dev->condlock` ②. Since the interrupt handler `s5p_mfs_irq()` is not disabled at that moment, there is a possibility of preemption by the interrupt handler between Line 52 and Line 54. If the interrupt handler acquires `&dev->irqlock` ① during this preemption, it creates an **interrupt lock dependency** from `&dev->condlock` ② to `&dev->irqlock` ①. Note that this interrupt lock dependency cannot be captured by existing static methods [7,14,15] without characterizing interrupt preemption. Consequently, in certain concurrent scheduling, an interrupt-based deadlock may occur, as depicted in Figure 1(b), resulting in a system halt.

Our Approach. In this work, we present Archerfish, the first static analysis approach to characterize the interrupt preemption to detect the interrupt-based deadlocks in the Linux kernel. At the core, Archerfish is empowered by the extended notion of an Interrupt-Aware Lock Graph (ILG), which captures both kinds of lock dependencies in the Linux kernel, *namely regular lock dependencies and interrupt lock dependencies*, caused by regular execution flow of threads and interrupt preemption, respectively. By effectively constructing the ILG, Archerfish can reduce the detection of interrupt-based deadlocks to the discovery and validation of dependence cycles. For example, the deadlock in Figure 1(a) can be detected by finding the cycle shown in Figure 1(c).

However, there are three core challenges of effectively analyzing the Linux kernel to identify the lock dependencies and refine deadlock cycles. **(1)** First, identifying the *Interrupt Service Routines (ISRs)* is challenging without domain-specific knowledge, as different subsystems in the Linux kernel often have their own unique APIs and callback interfaces for registering ISRs [71]. **(2)** Second, the interrupt preemption enlarges the concurrency reasoning space of static analysis. As each interrupt-enabled program location can be potentially interrupted [68], it is necessary to consider preemption at each statement to capture all possible interrupt-induced lock dependencies. However, the large concurrency reasoning space can significantly degrade the effectiveness of static analysis. **(3)** Third, it is crucial to identify feasible lock dependencies, taking into account the feasibility of *both interrupts and program paths*. However, for example, directly validating the path-feasibility of numerous lock dependencies using SMT solving [82] may cause low efficiency.

To address the three challenges, we design four stages shown in Figure 2 with several tailored techniques.

- First, to address the first challenge, Archerfish harnesses the power of Large Language Models (LLMs) to generate interrupt specifications that complement manually modeled ones for identifying interrupts. This is because LLMs have been trained on a vast amount of online textual data, including information related to the Linux kernel [61].

- Second, we employ a summary-based data-flow analysis to compute the lockset [32,81] and the state of interrupt enabling or disabling at relevant statements. To tackle the second challenge of the vast reasoning space of the interrupt preemption, we introduce the notion of *a preemption unit*, clustering all statements within a critical section. Our key idea is that a critical section [78] shares the same lockset so that any interrupts occurring within the region hold the same set of locks. Thus, separately computing each individual program point within the region is unnecessary during lockset analysis.
- Third, using the computed data-flow facts, Archerfish constructs the ILG and captures both kinds of lock dependencies as graph edges. In particular, it identifies dependency cycles within the ILG, each representing an interrupt-based deadlock.
- Fourth, to overcome the third challenge of expensive deadlock feasibility checking, our key idea is that not all lock dependencies are related to deadlock detection. Thus, we lazily check the feasibility of *both interrupts and paths*, deferring the costly validation for each discovered cycle instead of early validating each edge during ILG construction.

We have implemented the Archerfish prototype based on the LLVM framework [36] and evaluated it on the Linux kernel v6.4. Archerfish can analyze the Linux kernel (19.8 MLoC) in about one hour with a relatively low positive (49.7%). More excitingly, we have found 76 new and long-latent interrupt-based deadlocks in the Linux kernel and reported all of them to the Linux community. In total, 53 bugs have been confirmed by developers, and 46 of them have already been fixed in the mainstream Linux kernel with 2 CVE IDs assigned.

Contribution. We make the following contributions.

- We propose the notion of the Interrupt-Aware Lock Graph (ILG), which captures both kinds of regular and interrupt-related lock dependencies as edges, enabling the reduction of *interrupt-based deadlock* detection via graph cycle discovery and refinement.
- We present Archerfish, the first static interrupt-based deadlock detection for the Linux kernel, addressing three specific challenges of effectively constructing ILG and refining cycles.
- We evaluate Archerfish on Linux kernel v6.4, where it revealed 76 new interrupt-based deadlocks that across various subsystems in the Linux kernel.

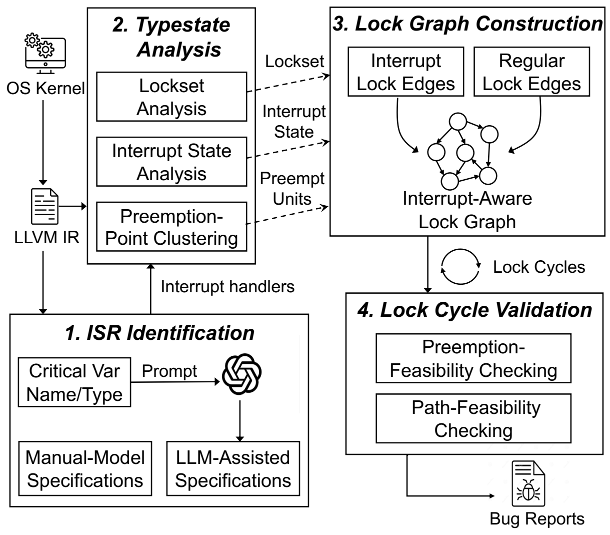


Figure 2: The design of Archerfish framework.

2 Background

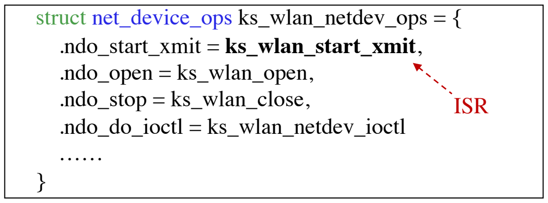
We present the interrupts' concept, the interrupt-based deadlocks, the limitations of existing works, and the threat model in § 2.1, § 2.2, § 2.3, and § 2.4, respectively.

2.1 Interrupts in Linux kernel

Interrupt Handling. An interrupt [43] in the Linux kernel is a special event that alters the regular execution flow of a program, and such action is also called *interrupt preemption* [68]. In contrast, the action that one thread switches to another thread is called *thread preemption* [72]. Their major difference is that interrupt preemption is *asymmetric*, indicating that interrupt runs in higher priority and cannot switch back to the previously preempted execution flow until the interrupt finishes its execution [43]. Thus, interrupt preemption could introduce new lock dependencies and cause deadlocks [2].

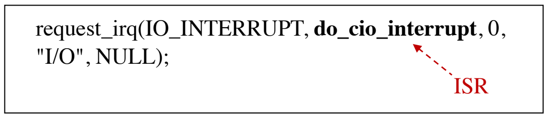
Interrupt Priority. Interrupts in the Linux kernel have two categories: hardware interrupts [42] and software interrupts [44], also known as *hardirqs* and *softirqs*, respectively. Hardirqs are triggered by interrupt signals from hardware devices and execute under the context called the *hardirq context*. In recent Linux kernel, local interrupt is disabled inside *hardirq context* by default, so hardirqs normally run with the highest priority and cannot be preempted by any other execution units, including other hardirqs [43].

Because hardirqs should execute quickly in response to hardware signals, time-consuming tasks would be deferred to softirqs [44]. Softirqs run in the execution context called the *softirq context* with lower priority than the hardirq context and can be preempted by hardirqs [44]. Nevertheless, both the hardirqs and softirqs belong to



```
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
    ......
}
```

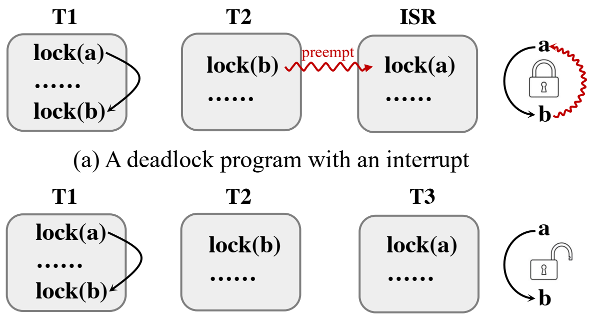
(a) Registered via a structure field interface



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

(b) Registered via an API call

Figure 3: Examples of ISR registration in Linux.



(a) A deadlock program with an interrupt

(b) The thread model missing interrupt preemption

Figure 4: (a) shows a deadlock program while (b) shows its previous thread modeling with one lock edge missed.

interrupts and have higher priority than normal kernel threads executing under the *process context* [23]. The priority-based preemption relationship leads to the specific concurrency model in the Linux kernel. Therefore, any static analyzer that aims to reason about interrupt preemption should consider this relationship.

Interrupt Service Routine (ISR) Registration. An ISR [3] is a function in response to interrupt signals from hardware devices or software interrupt. In the Linux kernel, an ISR is typically registered in two ways. First, an ISR can be registered by being assigned to a structure field. In Figure 3(a), function `ks_wlan_start_xmit()` is registered as a softirq ISR by being assigned to the `net_device_ops.ndo_start_xmit` structure field, a callback interface responsible for handling network packet transmission [71]. Second, an ISR can be registered by being passed as an argument to a specific API call. For example, as shown in Figure 3(b), the function `do_cio_interrupt()` is registered as a hardirq ISR for the interrupt line `IO_INTERRUPT` using the standard API `request_irq()` [3]. The large codebase of Linux contains diverse subsystems, each with its own interfaces or APIs for ISR registration. The variation presents a challenge for identifying ISRs and capturing interrupt preemption in static deadlock detection.

2.2 Interrupt-Based Deadlocks

Distinguishing interrupt-based deadlocks from other types of deadlocks [14, 15] is the presence of lock dependencies introduced by *interrupt preemption* [68]. Specifically, interrupt preemption is an asymmetric process [43], in which a preempted thread is required to wait until the ISR completes before it can proceed. Consequently, any locks held by the preempted thread cannot be released until the ISR finishes its execution. When the ISR itself attempts to acquire locks, it can give rise to lock dependencies, potentially resulting in cyclic lock acquisitions.

For example, consider Figure 4(a), in which lock b is held by thread $T2$ and the thread is preempted by an ISR that acquires another lock a. In this scenario, the interrupt lock dependency $b \rightsquigarrow a$ appears, as $T2$ cannot release lock b until the ISR finishes. When considering the regular lock dependency $a \rightarrow b$ introduced by $T1$ together, an interrupt-based deadlock could happen under the specific thread and the interrupt interleaving.

Note that the thread switching cannot introduce the lock dependencies (e.g., $b \rightsquigarrow a$) as interrupt preemption. This is because if a thread is holding a lock and is switched out, it could be switched back at any time to release the lock it acquired. This paper refers to the lock dependencies introduced by interrupt preemption as *interrupt lock dependencies* and deadlocks that involve interrupt lock dependencies as *interrupt-based deadlocks*.

2.3 Limitations of Existing Static Detectors

Existing static deadlock detection techniques [7, 14, 15, 18, 34, 35] do not consider the interrupt lock dependencies and model interrupts in the same way as threads. As a result, they cannot detect interrupt-based deadlocks. Consider the example program in Figure 4(b), where the interrupt in Figure 4(a) is modeled as thread $T3$ by the existing deadlock detectors and then the interrupt preemption is not captured to identify the lock dependency $b \rightsquigarrow a$. Consequently, the existing static deadlock detectors, which model interrupts as threads, are unable to detect deadlock situations that arise from interrupts.

Some works [38, 75] detect other specific interrupt-related concurrency bugs like data race [46, 75, 80] or atomicity violation [28, 38]. However, the challenges associated with modeling and detecting these types of bugs differ from those encountered in deadlock detection. Therefore, their approaches cannot be directly applied to identify interrupt-based deadlocks effectively.

2.4 Security Impacts of Interrupt-Based Deadlocks in the Linux Kernel

Threat Model. We characterize the deadlock vulnerability as potential lock acquisitions that can be exploited by local or remote adversaries. These adversaries have the capability to trigger specific deadlock-introducing ISRs through various means, such as remote network packets, local system call execution, or physical hardware access, manipulating the control flow. When these deadlock-introducing ISRs in the system are periodically activated and happen to preempt the execution of code involving vulnerable and cyclic lock acquisitions, a deadlock situation can arise. Once this deadlock is exploited, the corresponding ISRs can seize CPU cores, causing them to remain indefinitely stuck [2,4,5] and leading to issues like lockup [70] or system crashes. A concrete instance of the deadlock (CVE-2023-0160) [56] was reported in the BPF subsystem, allowing low-privileged local attackers to exploit the deadlock and crash the system.

Conventional deadlocks primarily impact blocked threads, allowing other threads to continue execution [7, 15]. In contrast, interrupt-based deadlocks pose a more severe impact by monopolizing the CPU with deadlock-introducing ISRs, resulting in a complete blockage of the entire system's execution and halting further progress.

3 Archerfish in a Nutshell

We present the problem formulation, the challenges, and our solutions to interrupt-based deadlock detection.

3.1 Problem Formulation and Challenges

In essence, interrupt-based deadlocks are cyclic lock dependencies that occur due to interrupt preemption. To identify and address these deadlocks, we introduce ILG by extending the conventional notion of lock graphs [15, 33], which incorporate both interrupt lock dependencies and regular lock dependencies as lock edges. As a result, the problem of interrupt-based deadlock detection is formulated as the construction of ILG and the discovery of feasible deadlock cycles within the graph. However, three challenges hinder the precise and efficient ILG construction and deadlock cycle validation.

C1: Lack of ISR Registration Specifications. Various methods exist to register interrupt handlers in Linux, a complex and feature-rich OS kernel containing many subsystems [74]. As shown in Figure 3 (a), without any domain-specific knowledge, it is difficult to identify that `net_device_ops.ndo_start_xmit` is a callback interface executed under the softirq context. However, identifying functions executed under interrupt contexts is the prerequisite for capturing interrupt preemption and interrupt lock dependencies. Thus, lacking domain-specific knowledge about ISR registration poses a significant challenge for interrupt-based deadlock detection.

C2: Large Interrupt-Involving Reasoning Space. The concurrency reasoning space of thread interleaving already grows exponentially with the number of statements [19, 65], and this complexity is further amplified by the presence of interrupt preemption. In particular, efficiently and precisely capturing lock dependencies becomes challenging when considering the interrupt enabling/disabling state at each statement, and the state could vary under different calling contexts, which require expensive context-sensitive analysis [79]. For example, in the Figure 1(a), `clear_work_bit()` could be preempted by the ISR `s5p_mfs_irq()` when called by function `enc_post_frame_start()`, but not when called by function `s5p_mfc_watchdog_worker()` because hardirq is already disabled by `spin_lock_irqsave()` at Line 184. Worse still, the large codebase of the Linux kernel also further complicates the interruption-involving concurrency reasoning space.

C3: Validation of Feasible Lock Dependencies. Finally, not all lock dependencies are feasible during runtime execution. First, the interrupt preemption leading to some lock dependencies may be infeasible as the preemption cannot happen before the corresponding ISR is registered [3]. Second, the path conditions of certain lock dependencies could also be infeasible. However, the large number of lock dependencies to validate and the high overhead (e.g., SMT solving [82]) of feasibility checking make efficient validation challenging.

3.2 Four Core Stages in Archerfish

The architecture of Archerfish is shown in Figure 2, consisting of four main stages to address these challenges.

S1: Interrupt Service Routine Identification (§5.1). Public LLMs like ChatGPT-4.0 are trained with vast online textual data, including those of Linux kernel [61]. Thus, we can consider LLMs as Linux *experts* and extract domain-specific knowledge about subsystem ISR registration from them. The subsystem specifications generated by LLMs would complement a set of standard ISR registration specifications modeled by humans.

S2: Lockset and Interrupt-State Analysis (§5.2). To effectively reason about the interrupt preemption and capture interrupt lock dependencies, our basic idea is that statements inside the same critical section share the same lockset. Consequently, instead of enumerating the possibility of interrupt preemption at each program location, we treat all statements in a critical section as a *preemption unit*, calculating a unified state of interrupts for the unit and reducing the preemption reasoning space. Furthermore, we propose a compositional and summary-based data-flow analysis to compute the lockset and interrupt state at interesting statements.

S3: ILG Construction (§5.3). By inspecting the computed data-flow facts, both *Interrupt Lock Edge (ILE)* and *Regular Lock Edge (RLE)* are captured to construct an ILG. Specifically, we capture RLEs by examining the lockset on statements of interest, while we capture ILEs by pairwise checking between several precomputed summaries on preempted threads and ISRs. After constructing ILG, a standard cycle detection algorithm [29] is run to identify potential deadlock cycles.

S4: Deadlock Cycle Validation (§5.4). Instead of performing heavyweight feasibility checking on each identified lock dependency edge, we delay the analysis by only performing feasibility checking on those forming a deadlock cycle. At this stage, Archerfish performs happen-before analysis [45,83] regarding ISR registration to examine the preemption feasibility and path-feasibility checking on each potential deadlock-leading dependency edge. Finally, deadlock reports are generated on validated interrupt-based deadlock cycles.

4 Preliminary

In this section, we introduce the definitions used in the paper and formulate the problem we aim to solve.

Abstract Domain. Figure 5 shows the basic abstract domain, where the symbol $s \in \mathbb{S}$ represents each statement in a standard LLVM-like language. Specifically, lock- and interrupt-related API call sites are modeled during static analysis. The symbol $o \in \mathbb{O}$ represents each lock object, and a cyclic dependency among them forms a deadlock. The symbol $t \in \mathbb{T}$ represents a kernel thread executing under the process context, while $isr \in \mathbb{R}$ represents an ISR [3] executing under the interrupt context, including both hardirq and softirq. Note that identification of ISRs is important for Archerfish to capture interrupt lock dependencies and perform deadlock detection. We give more details of ISR identification in § 5.1.

The $\mathbb{LS}$ and $\mathbb{RS}$ represent the lockset and the interrupt states computed by typestate analysis. Lockset $\mathbb{LS}$ maintains whether a specific lock $o \in \mathbb{O}$ is <u>**A**</u>cquired, <u>**R**</u>eleased or its state is Unknown ($\perp$). Interrupt state $\mathbb{RS}$ maintains whether a specific $isr \in \mathbb{R}$ is <u>**E**</u>nabled, <u>**D**</u>isabled or its state is Unknown ($\perp$). To reach flow-sensitivity, we compute a unique lockset $\mathbb{LS}$ and interrupt state $\mathbb{RS}$ at each statement $s \in \mathbb{S}$. For example, at Line 186 of the program in Figure 1(a), we identify that $\mathbb{LS} = (irqlock, \mathbf{A})$ and $\mathbb{RS} = (s5p\_mfc\_irq(), \mathbf{D})$, indicating *irqlock* is <u>**A**</u>cquired and $s5p\_mfc\_irq()$ is <u>**D**</u>isabled. Both $\mathbb{LS}$ and $\mathbb{RS}$ are derived from a compositional typestate analysis described in § 5.2, forming the foundation of deadlock detection.

Next, we formally define the notion of ILG.

$$
\begin{array}{rllll}
\text{Statements} & s \in \mathbb{S} & \text{Locks} & o \in \mathbb{O} \\
\text{Threads} & t \in \mathbb{T} & \text{Interrupt Handlers} & isr \in \mathbb{R} \\
\text{Lockset} & \multicolumn{3}{l}{\mathbb{LS} := \mathbb{S} \to (\mathbb{O},\ TY),} \\
 & \multicolumn{3}{l}{\quad \textit{where } TY = \{\underline{\mathbf{A}}\textit{cquired},\ \underline{\mathbf{R}}\textit{eleased},\ \perp\}} \\
\text{Interrupt State} & \multicolumn{3}{l}{\mathbb{RS} := \mathbb{S} \to (\mathbb{R},\ ST),} \\
 & \multicolumn{3}{l}{\quad \textit{where } ST = \{\underline{\mathbf{E}}\textit{nabled},\ \underline{\mathbf{D}}\textit{isabled},\ \perp\}}
\end{array}
$$

Figure 5: Basic abstract domain.

Definition 1. The Interrupt-Aware Lock Graph (ILG) is defined as a three-tuple, $\mathrm{G} = (\mathrm{N},\ \mathrm{E_S},\ \mathrm{E_I})$.

- N denotes the set of vertices in ILG, where each vertex represents a unique lock object o.
- $\mathrm{E_S}$ represents a set of regular lock edges with the form $o_1 \to o_2$, indicating that during thread or ISR execution, lock o_2 could be acquired with o_1 held.
- $\mathrm{E_I}$ represents a set of interrupt lock edges with the form $o_1 \rightsquigarrow o_2$, indicating that at certain program points, lock o_1 is held while an ISR preempts the execution and subsequently acquires o_2.

At a high level, interrupt lock edges are constructed by pairwise matching between preemption units and summaries of lock acquisition inside ISRs, both of which are constructed in § 5.2. We give more details of ILG construction in § 5.3. With the definition of ILG, we further define the concept of Interrupt-Based Lock Cycle (ILC).

Definition 2. An Interrupt-Based Lock Cycle (ILC) is a cycle on ILG where at least one of the edges on the cycle belongs to $\mathrm{E_I}$, representing a potential deadlock introduced by interrupt preemption.

Figure 1(c) shows an example of ILC constituting an interrupt lock edge ②$\rightsquigarrow$① and a regular lock edge ①$\to$②. Note that an ILC does not necessarily represent a real interrupt-based deadlock in practice, so further validation is required to reduce false positives. We give a detailed illustration in § 5.4. In addition, this paper focuses on detecting interrupt-based deadlocks, despite Archerfish being capable of detecting conventional deadlocks.

5 The Approach of Archerfish

In this section, we detail each core stage in Archerfish.

5.1 ISR Identification

Prompt queries with LLMs allow us to automatically extract ISR registration specifications in Linux subsystems, which supplement manually modeled standard specifications with a complete list shown in appendix A.3.

Prompt Template 1

Query: In the Linux kernel, is a function registered by interface [multi-layer type] typically executed under "process context", "hardirq context" or "softirq context"?

LLM: One of {hardirq context, softirq context, process context}

(a) Prompt template for ISR registration interface

Prompt Template 2

Query: In the Linux kernel, is a function registered by API [function name] via [index] 'th argument typically executed under "process context", "hardirq context" or "softirq context"?

LLM: One of {hardirq context, softirq context, process context}

(b) Prompt template for ISR registration API

Figure 6: Prompt templates for LLM-assisted specs.

To construct a prompt query [66], Archerfish performs static analysis to trace each address-taken function along its def-use chain until reaching one of the two cases:

- **Registration Interface:** If the function pointer $fptr$ is used as $*p = fptr$, we take the struct *type* of p as a potential ISR registration interface. The *type* would be used to construct a prompt query via the example template shown in Figure 6(a) to determine under which context the interface is executed.
- **Registration API:** If the function pointer $fptr$ is used as *call* $bar(..., fptr, ...)$, we extract the callee function name and its argument index, treating them as a potential ISR registration API. Similarly, these information would be used to construct a prompt query via the template in Figure 6(b).

LLMs can act as a classifier for the execution context of queried APIs or interfaces. With such specifications, we can identify whether a function is an ISR or a kernel thread by inspecting whether it is passed to these APIs or interfaces. Specifically, if an address-taken function is classified as $hardirq$ or $softirq$, it would be identified as an $isr \in \mathbb{R}$, otherwise it would be identified as $t \in \mathbb{T}$.

Example 5.1. To determine whether the function `ks_wlan_start_xmit()` in Figure 3(a) is an ISR or a thread, Archerfish traces its def-use chain and reaches the structure field `net_device_ops.ndo_start_xmit`. This structure field is an interface commonly used by Linux network developers for `NET_TX_SOFTIRQ` softirq registration. In our experiment, LLM can correctly recognize that it is executed under the $softirq$ context.

5.2 Lockset and Interrupt-State Analysis

Next, we perform a flow- and context-sensitive typestate analysis for computing lockset and interrupt states. Specifically, Archerfish runs forward data-flow analyses

$$\frac{\texttt{s}:\texttt{lock}(\texttt{o})}{\mathbb{LS}(s)\cup\{o\mapsto\mathbf{A}\}}\ (1)\qquad\frac{\texttt{s}:\texttt{unlock}(\texttt{o})}{\mathbb{LS}(s)\cup\{o\mapsto\mathbf{R}\}}\ (2)$$

$$\frac{\texttt{s}:\texttt{disable\_irq}(\texttt{isr})}{\mathbb{RS}(s)\cup\{isr\mapsto\mathbf{D}\}}\ (3)\qquad\frac{\texttt{s}:\texttt{enable\_irq}(\texttt{isr})}{\mathbb{RS}(s)\cup\{isr\mapsto\mathbf{E}\}}\ (4)$$

$$\frac{\begin{array}{c}\texttt{s}:\texttt{merge}(\texttt{s}_1,\texttt{s}_2)\\(o\mapsto ty_1)\in\mathbb{LS}(s_1),\ \ (o\mapsto ty_2)\in\mathbb{LS}(s_2)\\(isr\mapsto st_1)\in\mathbb{RS}(s_1),\ \ (isr\mapsto st_2)\in\mathbb{RS}(s_2)\end{array}}{\begin{array}{c}\mathbb{LS}(s)\cup\{o\mapsto LS_{merge}(ty_1,ty_2)\}\\\mathbb{RS}(s)\cup\{isr\mapsto RS_{merge}(st_1,st_2)\}\end{array}}\ (5)$$

$$LS_{merge}(ty_1,ty_2)=\begin{cases}\mathbf{A} & \texttt{if}\ ty_1=\mathbf{A}\ \texttt{or}\ ty_2=\mathbf{A}\\\mathbf{R} & \texttt{if}\ ty_1=\mathbf{R}\ \texttt{and}\ ty_2=\mathbf{R}\\\bot & \texttt{other cases}\end{cases}$$

$$RS_{merge}(st_1,st_2)=\begin{cases}\mathbf{E} & \texttt{if}\ st_1=\mathbf{E}\ \texttt{or}\ st_2=\mathbf{E}\\\mathbf{D} & \texttt{if}\ st_1=\mathbf{D}\ \texttt{and}\ st_2=\mathbf{D}\\\bot & \texttt{other cases}\end{cases}$$

Figure 7: Intra-procedural abstract operations

for each function following the reverse topological order on Call Graph (CG) [12, 16] and computes function summaries. These summaries can be inlined by callers to achieve efficient inter-procedural context-sensitive analysis and to be used for efficient ILG construction.

5.2.1 Intra-Procedural Data-Flow Analysis

At the entry of an analyzed function, the state of each lock o and isr are initialized as $\bot$. Next, Archerfish proceeds forward along the control flow graph to analyze each instruction sequentially until the function returns. The state at each instruction is initialized as a merge of states at its predecessors, and then the corresponding abstract operation shown in Figure 7 is used. A concrete list of modeled APIs can be found in appendix A.2.

Next, we provide a detailed illustration of the abstract operations. Note that the strong update [25], denoted by the $\mapsto$ operator, is applied to achieve flow-sensitivity.

- **Lock Acquisition and Release**: At each lock-acquisition API call $lock(o)$, Archerfish applies (1) $o \mapsto \mathbf{A}$ on the current lockset, indicating the state of lock o is updated as **A**cquired. A similar operation (2) is applied at each lock-release API call $unlock(o)$.
- **Interrupt Disabling and Enabling**: At each ISR-disabling API call $disable\_irq(isr)$, Archerfish performs the operation (3) $isr \mapsto \mathbf{D}$ on the current interrupt state to mark isr as **D**isable. The opposite operation (4) is performed at ISR-enabling API calls.
- **Merging.** The (5) merge operation first retrieves the lockset and interrupt states at predecessor in-

Preempt Summary	$\mathbb{M}_1 := (\mathbb{O},\ \mathbb{S}) \to (\mathbb{R}, ST)$, *where* $ST = \{\mathbf{E},\ \mathbf{D}\}$
Locking Summary	$\mathbb{M}_2 := (\mathbb{O},\ \mathbb{S}) \to \mathbb{O}$
Return Summary	$\mathbb{M}_3 := (\mathbb{RS},\ \mathbb{LS})$

Figure 8: Three function summaries

structions, and then performs LS_{merge} and RS_{merge} to conservatively merge these states. Specifically, *may* analysis is used to determine whether a lock o could be **A**cquired or whether an isr could be **E**nabled.

Example 5.2. If the typestate from one incoming branch is $\{(o, \mathbf{A}),\ (isr, \bot)\}$ and that from the opposite branch is $\{(o, \mathbf{R}),\ (isr, \mathbf{E})\}$, the merged results would be $\{(o, \mathbf{A}),\ (isr, \mathbf{E})\}$, indicating lock o could be acquired and isr could be enabled at the branches confluence.

5.2.2 Function Summary Construction

Figure 8 shows the definition of three function summaries constructed along the data-flow analysis.

- **Preemption Unit Summary** ($\mathbb{M}_1$): This summary forms the units of interrupt preemption. Each summary item represents a critical section, denoted by a lock object and a lock statement $(o,\ s)$. The item also maintains a set of conservative interrupt states, representing the set of ISRs that could preempt this critical section. To do so, we record *must*-**D**isabled and *may*-**E**abled state of each already analyzed ISR.
- **Lock Acquisition Summary** ($\mathbb{M}_2$): This summary represents lock acquisitions inside a function. Each summary item is also denoted by a lock object and a lock statement $(o,\ s)$. Different from $\mathbb{M}_1$, the item in $\mathbb{M}_2$ records a set of locks that *must* have been **R**eleased before this lock acquisition.
- **Return State Summary** ($\mathbb{M}_3$): This summary caches the lockset and interrupt states at the end of a function, which can be inlined by caller functions to update the typestate at call sites.

Summary Construction. Algorithm 1 shows how to construct the three summaries. First, for each encountered lock-acquisition API call $s : lock(o)$, we construct a new item of $\mathbb{M}_1$, recording the set of ISRs that *must* have been **D**isabled and cannot preempt the current critical section (Line 3). Similarly, we construct a new item of $\mathbb{M}_2$, recording the set of locks *must* have been **R**eleased before lock o is acquired (Line 4). Second, for each ISR-enabling API call $s : enable\_irq(isr)$ (including those inlined from callee's summary), we first retrieve the set of locks $(o,\ s')$ guarding s, and update corresponding items of $\mathbb{M}_1$ to mark that *isr may* be **E**nabled inside the critical sections (Lines 5-7). Finally, at the function return,

Algorithm 1: Function Summary Construction

```
1 Function SummaryConstruction($s$):
2     case $s : lock(o)$ do
3         $\mathbb{M}_1[(o,\ s)] := \{(isr,\ \boldsymbol{D}) : (isr,\ \boldsymbol{D}) \in \mathbb{RS}[s]\}$;
4         $\mathbb{M}_2[(o,\ s)] := \{o' : (o',\ \boldsymbol{R}) \in \mathbb{LS}[s]\}$;
5     case $s : enable\_irq(isr)$ do
6         foreach $o : (o,\ \boldsymbol{A}) \in \mathbb{LS}[s]$, acquired at $s'$ do
7             $\mathbb{M}_1[(o,\ s')] \cup (isr,\ \boldsymbol{E})$
8     case $s : return$ do
9         $\mathbb{M}_3 := (\mathbb{RS}[s],\ \mathbb{LS}[s])$
```

Algorithm 2: Function Summary Inlining

```
1 Function SummaryInlining($s : call\ bar()$):
2     Retrieve $\mathbb{M}_1'$, $\mathbb{M}_2'$ and $\mathbb{M}_3'$ from $bar$
3     foreach $isr : (isr,\ \boldsymbol{D}) \in \mathbb{RS}[s]$ do
4         foreach $(o',\ s') : (isr,\ \boldsymbol{E}) \notin \mathbb{M}_1'[(o',\ s')]$ do
5             $\mathbb{M}_1'[(o',\ s')] \cup (isr,\ \boldsymbol{D})$
6     foreach $o : (o,\ \boldsymbol{R}) \in \mathbb{LS}[s],\ (o',\ s') \in \mathbb{M}_2'$ do
7         $\mathbb{M}_2'[(o',\ s')] \cup \{o\}$
8     Append $\mathbb{M}_1'$, $\mathbb{M}_2'$ to $\mathbb{M}_1$, $\mathbb{M}_2$
9     Update $\mathbb{RS}$, $\mathbb{LS}$ by $\mathbb{M}_3'$
```

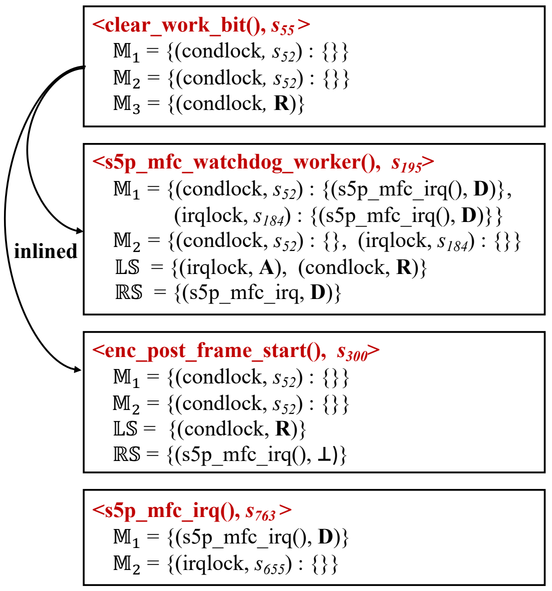
$\mathbb{LS}$ and $\mathbb{RS}$ would be cached as $\mathbb{M}_3$ (Lines 8-9), used by callers to update typestates at call sites.

5.2.3 Function Summary Inlining.

At a call site of an already analyzed function, constructed summaries inside the callee function can be inlined by the caller for context-sensitive inter-procedural analysis. Algorithm 2 shows the detail of the inlining process.

Summary Inlining. Archerfish first retrieves the three summaries $\mathbb{M}_1'$, $\mathbb{M}_2'$ and $\mathbb{M}_3'$ constructed in callee function (Line 2). Before inlining the summary, we take two steps. First, if some ISRs have already been disabled at the call site and are not enabled inside the callee's preempt unit, Archerfish will update the summary item in $\mathbb{M}_1'$ by marking the state as **D**isabled. (Lines 3-5). Second, some locks have already been **R**eleased before the call site, so Archerfish needs to insert these released locks into each item of callee's $\mathbb{M}_2'$ (Lines 6-7). After these, callee's $\mathbb{M}_1'$ and $\mathbb{M}_2'$ would be inlined (Line 8), and the current lockset and interrupt states are updated by callee's Return State Summary $\mathbb{M}_3'$ (Line 9).

Example 5.3. Figure 9 shows partial analysis results of Figure 1. Once `clear_work_bit()` is analyzed, its summary $\mathbb{M}_1$ contains a preemption unit $\{(\texttt{condlock}, s_{52}) : \{\}\}$, indicating that the critical section $(\texttt{condlock}, s_{52})$ does not execute with any ISR explicitly enabled or disabled. Next, the summary is inlined by two caller



<clear_work_bit(), s_{55}>
$\mathbb{M}_1$ = {(condlock, s_{52}) : {}}
$\mathbb{M}_2$ = {(condlock, s_{52}) : {}}
$\mathbb{M}_3$ = {(condlock, **R**)}
<s5p_mfc_watchdog_worker(), s_{195}>
$\mathbb{M}_1$ = {(condlock, s_{52}) : {(s5p_mfc_irq(), **D**)}, (irqlock, s_{184}) : {(s5p_mfc_irq(), **D**)}}
$\mathbb{M}_2$ = {(condlock, s_{52}) : {}, (irqlock, s_{184}) : {}}
$\mathbb{LS}$ = {(irqlock, **A**), (condlock, **R**)}
$\mathbb{RS}$ = {(s5p_mfc_irq, **D**)}
inlined
<enc_post_frame_start(), s_{300}>
$\mathbb{M}_1$ = {(condlock, s_{52}) : {}}
$\mathbb{M}_2$ = {(condlock, s_{52}) : {}}
$\mathbb{LS}$ = {(condlock, **R**)}
$\mathbb{RS}$ = {(s5p_mfc_irq(), ⊥)}
<s5p_mfc_irq(), s_{763}>
$\mathbb{M}_1$ = {(s5p_mfc_irq(), **D**)}
$\mathbb{M}_2$ = {(irqlock, s_{655}) : {}}

Figure 9: The analysis result for code in Figure 1.

functions. For caller `s5p_mfc_watchdog_worker()`, since the hardware ISR `s5p_mfc_irq()` is already disabled at Line 184, the summary $\mathbb{M}_1$ is updated to $\{(\texttt{condlock}, s_{52}) : \{(\texttt{s5p\_mfc\_irq()}, \mathbf{D})\}\}$, indicating that all statements inside the critical section cannot be preempted by `s5p_mfc_irq()`. On the contrary, no ISR is explicitly disabled or enabled inside the caller `enc_post_frame_start()`, so the summary item $\{(\texttt{condlock}, s_{52}) : \{\}\}$ is inlined without being updated.

5.3 Interrupt Lock Graph Construction

In this stage, Archerfish proceeds to perform ILG construction. The core algorithm is shown in Algorithm 3.

Interrupt Lock Edge Construction. Interrupt lock dependencies are captured as interrupt lock edges ($\mathbf{E_I}$). As shown in `InterruptLockEdgeAnalysis()`, pairwise matching is run between each preemption unit ($\mathbb{M}_1$) of lock o inside a function f and each higher priority *isr* (Lines 2-5). If the *isr* is not disabled inside the preemption unit, for each lock acquisition ($\mathbb{M}_2$) of o' inside *isr*, an interrupt lock edge $o \rightsquigarrow o'$ is constructed (Lines 6-10).

Regular Lock Edge Construction. Regular lock dependencies are captured as regular lock edges ($\mathbf{E_S}$). As shown in `RegularLockEdgeAnalysis()`, for a *lock* statement on lock o and each already acquired lock o', a regular lock edge $o' \rightarrow o$ is constructed (Lines 12-14) ; while for a *call* statement, Archerfish performs pairwise matching between each already acquired lock o and each acquisition of lock o' inside the callee function. If o is not released before o' is acquired inside the callee, a regular lock edge $o \rightarrow o'$ is constructed (Lines 15-20).

Once ILG is constructed, we use a standard cycle-detection algorithm [29] to identify an initial set of ILCs.

Algorithm 3: ILG Construction

```
1  Function InterruptLockEdgeAnalysis():
2      foreach $f \in \mathbb{T} \cup \mathbb{R}$, $isr \in \mathbb{R}$ do
3          if isr has higher priority than $f$ then
4              Retrieve $\mathbb{M}_1$ from $f$;
5              foreach $(o,\ s) \in \mathbb{M}_1$ do
6                  if $(isr, \boldsymbol{D}) \in \mathbb{M}_1[(o,\ s)]$ then
7                      continue;
8                  Retrieve $\mathbb{M}_2{}'$ from isr;
9                  foreach $(o',\ s') \in \mathbb{M}_2{}'$ do
10                     add $o \rightsquigarrow o'$

11 Function RegularLockEdgeAnalysis($s$):
12     case $s$ : lock($o$) do
13         foreach $(o', \boldsymbol{A}) \in \mathbb{LS}[s]$ do
14             add $o' \rightarrow o$ ;
15     case $s$ : call bar() do
16         Retrieve $\mathbb{M}_2$ from bar;
17         foreach $o$ : $(o, \boldsymbol{A}) \in \mathbb{LS}[s]$ do
18             foreach $(o',\ s') \in \mathbb{M}_2$ do
19                 if $o \notin \mathbb{M}_2[(o',\ s')]$ then
20                     add $o \rightarrow o'$
```

Example 5.4. In Figure 9, by running pairwise matching between summaries $\mathbb{M}_1$ of `enc_post_frame_start()` and $\mathbb{M}_2$ of `s5p_mfc_irq()`, we can construct an interrupt lock edge *condlock* $\rightsquigarrow$ *irqlock*. By performing pairwise matching between $\mathbb{M}_2$ of `clear_work_bit()` and $\mathbb{LS}$ of `s5p_mfc_watchdog_worker()` at s_{195}, we can construct a regular lock edge *irqlock* $\rightarrow$ *condlock*. As a result, the two lock edges form an ILC, as shown in Figure 1(c).

5.4 Lazy Deadlock Cycle Validation

Finally, each detected ILC is validated in terms of both the happen-before relationship and path feasibility.

5.4.1 Interrupt Happen-Before Validation

Synchronization between threads and ISRs should be characterized to ensure a true deadlock. Specifically, since interrupt preemption cannot happen before it is registered, all statements before ISR registration cannot be preempted. To this end, we perform a static happen-before analysis to validate the feasibility of an interrupt lock edge $o \rightsquigarrow o'$ in terms of ISR registration. Specifically, we perform a forward search from the last statement of the preemption unit of o, following the context-sensitive control flow graph. Once the registration statement of the ISR acquiring o' is found during the graph traversal, the interrupt lock edge is considered invalid.

$$\Phi(\circ) = \bigwedge_{e \in E_S} \Phi_S(e) \wedge \bigwedge_{e' \in E_I} \Phi_I(e')$$

$$\Phi_S(e) = \underbrace{\bigvee_{p \in P(T_{entry},\, s)} \Phi(p)}_{(1)} \wedge \underbrace{\bigvee_{p' \in P(s,\, s')} \Phi(p')}_{(2)}$$

$$\Phi_I(e) = \forall s \in S_p, \underbrace{\bigvee_{p \in P(T_{entry},\, s)} \Phi(p)}_{(3)} \wedge \underbrace{\bigvee_{p' \in P(R_{entry},\, s')} \Phi(p')}_{(4)}$$

Figure 10: Path-feasibility constraints of an ILC

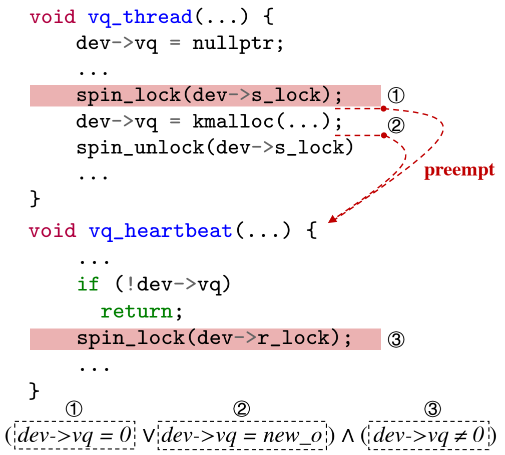
5.4.2 Interrupt Path-Feasibility Validation

Figure 10 shows the path constraints $\Phi(\circ)$ of an ILC (denoted by $\circ$), which consist of two parts: path constraints on regular lock edges, denoted as $\Phi_S(e)$ and the path constraints on interrupt lock edges, denoted as $\Phi_I(e')$. The Z3 SMT solver is used to determine path feasibility.

Regular Lock Edge. At a high level, the path constraints $\Phi_S(e)$ of a regular lock edge represent that at least one path starting from the entry point could lead to lock dependency $o \rightarrow o'$. To do so, we first collect the disjunctive path conditions of each path from the entry to the first lock acquisition site s of o, as shown in Figure 10 (1). Second, we collect the disjunctive path conditions of each path from s to the second lock acquisition site s' of o', as shown in Figure 10 (2).

Interrupt Lock Edge. The path condition $\Phi_I(e)$ of an interrupt locking edge $o \rightsquigarrow o'$ is more complicated, as the preemption at each statement inside the preemption unit with target ISR enabled (denoted by S_p) should be considered. Consequently, the disjunctive path conditions of each path from entry to each statement s in S_p are encoded, as shown in Figure 10 (3). Finally, the disjunctive path conditions of each path from ISR entry to lock acquisition site s' are encoded, as shown in Figure 10 (4).

Example 5.5. Figure 11 depicts the path conditions on the interrupt lock edge $s\_lock \rightsquigarrow r\_lock$ when the critical section of `s_lock` in the `vq_alloc()` function is preempted by the `vq_heartbeat()` ISR. With two valid preemption points within the critical section, the first part constraint (Figure 10 (3)) is formulated as the disjunctive conditions of these two preemption points (① ∨ ②). The second part constraint (Figure 10 (4)) consists of the disjunctive conditions on program paths from the entry of `vq_heartbeat()` to the lock acquisition site of `r_lock` (③). By evaluating the conjunction of these two parts using an SMT solver, the constraints are satisfied, as the second preemption point ② is feasible along the path.



```
void vq_thread(...) {
    dev->vq = nullptr;
    ...
    spin_lock(dev->s_lock);   ①
    dev->vq = kmalloc(...);   ②
    spin_unlock(dev->s_lock)
    ...
}
void vq_heartbeat(...) {
    ...
    if (!dev->vq)
      return;
    spin_lock(dev->r_lock);   ③
    ...
}
```

$$(\overset{①}{dev\text{->}vq = 0} \vee \overset{②}{dev\text{->}vq = new\_o}) \wedge (\overset{③}{dev\text{->}vq \neq 0})$$

Figure 11: An example of path-feasibility validation. The symbol *new_o* is the return value of `kmalloc()`.

6 Implementation

Archerfish is implemented on LLVM [36] (4,034 Loc) and uses Z3 [82] as its path condition solver. Several Python scripts (354 Loc) are implemented to perform network interaction with the OpenAI RESTful API [49]. For the pointer aliasing information, we use a standard pointer analysis, following the previous work [21, 48].

Lockset and Interrupt-State Analysis. Archerfish performs analysis by modeling a set of locking and interrupt-related APIs in the Linux kernel. Appendix A.2 shows the whole set of these APIs. To avoid these APIs being inlined during compilation, we attach `__attribute__((noinline))` attribute on these functions before compilation. We model several heavily used lock types in the Linux kernel, such as `spin_lock` and `rw_lock` [69], and model both the hardware and software interrupts (e.g., timer and tasklet [44]).

LLM-Assisted Specification Generation. The program data for constructing prompt queries, including the struct types [47] and function names, are extracted from compiled LLVM IR. Since the LLVM type system does not contain the name of the structure field types, we compile the Linux kernel with the "-g" option and parse LLVM metadata to extract such information. To construct prompt queries, we use RESTful APIs provided by Azure OpenAI Service and ChatGPT-4.0 model [49].

Path Conditions Solving. Path feasibility is validated with Z3 SMT solver, where each variable in the LLVM IR is modeled as a bit vector, on top of which arithmetic computation and guarded constraints can be modeled and solved. Following existing works [7, 14], the conditions solving is under-constrained, losing the bounds of external values (e.g., those derived from hardware IO operations or input arguments of root functions). As a result, false positives are possible due to the presence of under-constrained values.

7 Evaluation

This section presents the evaluation results of Archerfish, through investigating the following research questions:

- **RQ1:** How effective is Archerfish in detecting interrupt-based deadlocks? (§ 7.1)
- **RQ2:** How do the stages contribute to enhancing the effectiveness of Archerfish? (§ 7.2)

We also discuss other points, such as the false negatives and generalizability of our approach in § 7.3.

Experiment Setup. We built the source code of Linux to LLVM IR using clang-12 and evaluated the latest version Linux (v6.4) at the time of the experiment. Following the conventional setup, we used the built-in `allyesconfig` setting during compilation. To generate subsystem specifications, Archerfish relies on the Azure OpenAI Service APIs [49] and uses the ChatGPT-4.0 model for evaluation. While performing the prompting, we set the temperature as 0.1 and set Top-p as 0.2, to achieve relatively stable and accurate results. All experiments were conducted on an Ubuntu server with two 20-core Intel(R) Xeon CPU@2.20GHz and 256GB of physical memory.

Note that Archerfish is the first static tool for detecting interrupt-based deadlocks in Linux kernel. To the best of our knowledge, no other open-source tools are available that can serve as a basis for comparison. Crucially, Archerfish is specific to detect interrupt-based deadlocks that previous static methods have overlooked.

7.1 Effectiveness on Bug Detection

The Archerfish's effectiveness in detecting interrupt-based deadlocks is evaluated and summarized in Table 1.

Performance. Archerfish finished analysis of Linux v6.4 in 68.6 minutes with a peak memory of 38.3G. We inspected the time consumption of each step and found that the specification generation (Stage I), the data-flow analysis (Stage II), ILG construction (Stage III), and ILCs validation (Stage IV) consumes 31.0 mins, 28.4 mins, 1.3 mins, and 7.9 mins, respectively. The high proportion of time consumption in Stage I is mainly due to network communication with OpenAI, and the generated specifications can be reused for later analysis. These results demonstrate that Archerfish can scale up to large software like Linux within an acceptable timeframe.

False Positive Analysis. Archerfish reported 151 bugs with 75 false warnings, resulting in a 49.7% false positive rate. Specifically, we spent roughly 10 - 60 minutes examining each bug report, which depends on the complexity of the bug report and the corresponding source code. Two key reasons lead to false positives. First, the misidentification of ISRs or threads causes 34 of the false alarms. Specifically, out of these 34 false alarms,

Table 1: Analysis results on the Linux kernel v6.4

	Description	Results
Linux kernel statistics	Lines of source code	19.8M
	Number of functions	798.3K
Performance	Peak memory	38.3G
	Time spend on Stage I (Spec. generation)	31.0 mins
	Time spend on Stage II (Data-flow analysis)	28.4 mins
	Time spend on Stage III (ILG construction)	1.3 mins
	Time spend on Stage IV (ILC validation)	7.9 mins
	Total time (Stages I - IV)	68.6 mins
Precision	Validated ILCs **(detected bugs)**	151
	Real interrupt-based deadlocks **(real bugs)**	76
ISR and thread identification	Kernel threads	136,207
	Softirq ISRs by manual-model specifications	689
	Hardirq ISRs by manual-model specifications	1,226
	Softirq ISRs by LLM-assisted specifications	578
	Hardirq ISRs by LLM-assisted specifications	1,151
	Total tokens used for prompt query	752,666
ILG construction	Preempt units	309,821
	Preempt units with ISRs enabled	234,547
	Total ISRs	3,644
	ISRs containing lock acquisitions	1,449
	Lock acquisition inside ISRs	5,957
	Regular lock edges	1,507,760
	Interrupt lock edges	3,962,932
Interrupt-based deadlocks detection	Detected ILCs	353
	Invalid ILCs	202
	ILCs pruned by happen-before analysis	67
	ILCs pruned by path-feasibility validation	135
	Validated ILCs brought by hardirq preemption	52
	Validated ILCs brought by softirq preemption	99

21 are due to the case that an ISR cannot be identified and is misclassified as a kernel thread. Such misclassification leads to infeasible preemption scenarios, where the misclassified ISR is deemed able to be preempted by other ISRs. In the remaining 13 cases, a kernel thread is misidentified as an ISR due to incorrect specifications generated by LLMs, also leading to infeasible preemption scenarios. The high proportion of false alarms in this category indicates that correctly identifying ISRs is important to interrupt-based deadlock detection. Second, complex path conditions involving under-constrained values cause 25 false alarms. For example, Archerfish reports a false alarm on the function `gmc_v10_0_process_interrupt()`, which could be executed under both interrupt context and process context, depending on an under-constrained hardware status value received from IO operators. Since the actual execution path is directly controlled by the hardware status, such path constraints cannot be identified by static analysis. The remaining cases are associated with false alias relationships, which are inherent static analysis problems.

Real-World Impacts. We manually reviewed all the detected interrupt-based deadlocks and identified 76 of them as genuine and new bugs. These interrupt-based deadlocks are difficult to detect, as they have remained hidden for an average of 9.9 years. We wrote 37 patches to fix these bugs and sent the patches to the corresponding Linux kernel subsystem developers. Currently, 53 of these bugs have been confirmed by the Linux kernel developers, and most of the bug-fixing patches have al-

Table 2: Accuracy of LLM-Assisted Specifications

Hardirq		**Softirq**	
Precision	Recall	Precision	Recall
78.6	87.3	80.6	88.0

ready been merged into the mainstream Linux kernel or corresponding subsystem development branch, resulting in 46 bugs already fixed.

The detected interrupt-based deadlocks cover numerous subsystems of Linux kernel, including various device drivers, network stack, and filesystems. Linux kernel developers highly appreciated our bug-reporting and patch-submitting efforts and demonstrated high interest in our tool with comments like *"Will this tool be available for general use? It's obviously quite handy."*, *"Your experimental tool looks really promising."*, *"Interesting."*. Furthermore, two bugs inside the SCTP and TIPC network subsystems are assigned 2 CVEs. Details of these detected bugs can be found in appendix A.4.

7.2 Effectiveness on Each Stage

Next, we investigate the key stages of Archerfish.

7.2.1 ISR Identification

We study how LLMs address the first challenge by inferring subsystem ISR registration specifications automatically. As shown in Table 1, among all the 3,644 ISRs identified by Archerfish, 1,729 (47.1%) are derived from the LLM-assisted subsystem specifications. During the process, Archerfish spent 752,666 tokens in prompting with the GPT-4 model, with 1,292 APIs and 19,172 interfaces collected and used in prompting.

Out of all the identified ISRs, 1,267 are softirq ISRs, while 2,377 are hardirq ISRs. ISRs only account for a small portion (2.6%) of all routines, compared to a large number of kernel threads (136,207). Despite their relatively small number, ISRs play a crucial role and lead to many interrupt-based deadlocks.

Accuracy of LLM-Assisted Specifications. To examine the precision of the LLM-assisted specifications, we randomly sampled 150 hardirq and 150 softirq ISRs from the generated specifications and manually inspected their correctness. Additionally, to examine the recall, we collected 150 real hardirq and 150 real softirq ISRs (the ground truth) from subsystem API documentation, code comments, or manual code reviewing, and checked whether they are covered by the LLM-assisted specifications. Table 2 shows that the overall precision and recall are 79.6% and 87.6%, respectively.

Ablation Study 1: LLM-Assisted Specifications. We further performed end-to-end ablation evaluation to

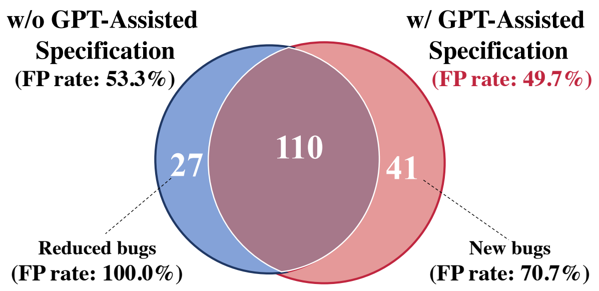


Figure 12: Bug reports of Archerfish with (denoted as $w/$) or without (denoted as w/o) the assistance of GPT.

examine the direct effect of LLM-assisted subsystem specifications on bug detection. As shown in Figure 12, with the LLM-assisted specifications, Archerfish detects 41 new bugs and misses 27 originally detected bugs, compared to Archerfish without the help of LLMs.

We manually examined all the 27 missed bugs and confirmed they are false warnings. The false warnings arise from situations of infeasible preemption, where an ISR is not correctly identified and mistakenly classified as a kernel thread, thus allowing the ISR to be preempted by other ISRs incorrectly. With the LLM-assisted subsystem specifications, more real ISRs can be identified, thus reducing such infeasible preemptions.

Out of the 41 additionally reported bugs with the help of LLM-assisted specifications, 12 are true positives. The false positive rate of these additionally reported bugs is a bit higher (70.7% vs 53.3%) because the LLM-assisted specifications contain some fake ISRs. When considering together the 27 reduced bugs brought by infeasible preemption, the overall false positive rate remains similar (49.7% vs 53.3%). In conclusion, the specifications help Archerfish capture new preemption and reduce infeasible preemption with high precision, resulting in 12 more real bugs detected while remaining comparable FP rate shown in our evaluation. A case study of a real bug can be found in appendix A.1. To sum up, the effect of LLM-assisted specifications in Stage I of Archerfish is highly effective.

7.2.2 Lockset and Interrupt-State Analysis

We first present some statistical data in data-flow analysis (Stage II). During typestate analysis, 309,821 preempt units are constructed, and 234,547 among them have at least one ISR enabled. On the other hand, 5,957 lock acquisition summaries are constructed inside 1,449 ISRs. These lock acquisitions and preempt units can introduce interrupt lock dependencies, represented by interrupt lock edges on the ILG. Specifically, a total of 3,962,932 (72.4%) interrupt lock edges and 1,507,760 (27.6%) regular lock edges are constructed. This result indirectly indicates that, despite the number of ISRs being much smaller than kernel threads, interrupt preemption is a noteworthy source of deadlocks in the Linux kernel that existing deadlock detection tools have ignored.

Ablation Study 2: Preemption Unit. Next, we study the effectiveness of the interrupt preemption unit to address the large concurrency reasoning space. Importantly, the preemption unit can significantly reduce the time costs for interrupt preemption analysis, as only critical sections need to be checked instead of all statements. To demonstrate the importance of this design, we created an ablation group called Archerfish$^-$, where each statement is considered a unique preemption point, and executed Archerfish$^-$ with the same experiment setup. Our evaluation results indicate that Archerfish$^-$ takes 166.7 minutes to complete the analysis, resulting in a time overhead that is 2.43 times greater than before. The peak memory usage is also up to 130.2G (3.39x), as we require more memory resources to maintain enormous summaries for each statement during typestate analysis. In summary, using the preemption unit can significantly reduce the overhead regarding both time and memory.

7.2.3 Lazy Circular Lock Dependency Validation

We first demonstrate the effectiveness of deadlock cycle validation (Stage IV) to reduce false positives. By performing cycle detection on the ILG, we initially detected 353 ILCs. In total, 202 (57.2%) ILCs are pruned away during validation. Specifically, 67 ILCs are pruned by the interrupt happen-before validation, and 135 ILCs are dropped due to infeasible paths.

Ablation Study 3: Lazy Validation. We next explore the effectiveness of the lazy validation strategy in improving efficiency. As shown in Table 1, the number of lock edges (5,470,692) significantly exceeds the number of finally detected ILCs (353). Consequently, it is inefficient to validate the feasibility of every edge, highlighting the need for the lazy validation strategy.

7.3 Discussion

False Negative Study. We collected the past interrupt-based deadlocks and examined whether Archerfish can find them. Specifically, we searched the commit history of the Linux kernel using keywords, including “interrupt” and “deadlock”, and examined each relevant commit message. We then identified 16 existing interrupt-based deadlocks in the past five years. These deadlocks were detected by Lockdep [73] or manual code review, 14 of which can be initially detected by Archerfish in their corresponding historical versions. We manually checked the two missed bugs and found that their critical ISRs cannot be identified with LLMs, resulting in the missed corresponding interrupt lock dependencies. By manually modeling those ISRs, Archerfish can finally detect those two bugs without inducing false negatives.

Generalizability. Archerfish aims to detect interrupt-based deadlocks on `spin_lock` and `rwlock` [69] inside the Linux kernel. However, the lockset analysis, ILG construction, and cycle validation are general to model various locks. We believe that, with further engineering effort, Archerfish can be extended to model other locking mechanisms (e.g., completion variables and waiting queues) or detect interrupt-aware deadlocks in other OS kernels [1]. We leave the extension as our future work.

8 Related Work

Concurrency bug detection in the OS Kernel is an important problem [6,8,9,24,27,80]. Among the diverse types of concurrency errors [15, 26, 37, 38, 67, 75], including data races [46, 75, 80], atomicity violation [28, 38] and deadlocks [7, 14], we focus on presenting the previous deadlock detection work in this section.

Static Deadlock Detection. Archerfish is specialized for detecting interrupt-based deadlocks and is orthogonal to existing works [7,14,15,21,34], which generally focus on deadlocks under thread interleaving and do not consider interrupt preemption. Recently, a few works were proposed for deadlock detection in specialized domains. For example, DeadWait [63] focuses on detecting deadlocks in asynchronous C# programs. Also, the work [58] proposes approaches for static communication deadlock detection in Go programs. Recently, DLOS [7] proposes a deadlock detection tool for OS kernel; however, their algorithm still does not consider interrupt preemption.

Dynamic Deadlock Detection. Dynamic tools like Lockdep [73] and ThreadSanitizer [64] can perform program instrumentation and monitor the runtime locking behaviors for deadlock detection. However, dynamic and static approaches have been separate realms for bug finding, having different merits. For thoroughly interrupt-based deadlocks detection with higher code coverage, a static bug detector has its advantages since there is no reliance on test cases or runtime environment like concurrency fuzzing [8, 11, 17, 27, 80] or controlled concurrency testing [57, 77]. As an illustration of its effectiveness, Archerfish has uncovered 76 previously unknown and long-latent interrupt-based deadlocks in the Linux kernel despite subjecting the system to stress testing and extensive fuzzing.

Finally, we are the first to use LLMs to infer interrupt specifications. Using LLMs for dynamic or static program analysis is an interesting recent trend [39,59,62,76] on it. We believe that using LLMs for specification generation would bring new inspiration for combining LLMs into security analysis and solving practical problems.

9 Conclusion

We have presented Archerfish, a static and effective approach for interrupt-based deadlock detection in the Linux kernel. Evaluation shows that Archerfish effectively analyzes the Linux kernel in around one hour, uncovering 76 previously unknown interrupt-based deadlocks, with 53 confirmed and mostly fixed (with two assigned CVE IDs), demonstrating its real-world impact.

Acknowledgment

We thank the reviewers for their valuable comments on this work. This work is supported by the PRP/004/21FX grant from the Hong Kong Innovation and Technology Commission and research grants from Huawei and TCL. Yuandao Cai is the corresponding author.

References

[1] Freebsd. https://www.freebsd.org.

[2] Infiniband. https://github.com/torvalds/linux/commit/10af303192bc5490bb39b29541ecb0ead2eff1ce.

[3] Interrupt service routines. http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch06lev1sec3.html.

[4] Netlink deadlock. https://github.com/torvalds/linux/commit/8d61f926d42045961e6b65191c09e3678d86a9cf.

[5] Ufs deadlock. https://github.com/torvalds/linux/commit/948afc69615167a3c82430f99bfd046332b89912.

[6] ABDULLA, P., ARONIS, S., JONSSON, B., AND SAGONAS, K. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2014), POPL '14, Association for Computing Machinery, p. 373–384.

[7] BAI, J., LI, T., AND HU, S. DLOS: effective static detection of deadlocks in OS kernels. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), J. Schindler and N. Zilberman, Eds., USENIX Association, pp. 367–382.

[8] BAI, J.-J., LAWALL, J., CHEN, Q.-L., AND HU, S.-M. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (USA, 2019), USENIX ATC '19, USENIX Association, p. 255–268.

[9] BAI, J.-J., LAWALL, J., AND HU, S.-M. Effective detection of sleep-in-atomic-context bugs in the linux kernel. *ACM Trans. Comput. Syst. 36*, 4 (apr 2020).

[10] BROTHERSTON, J., BRUNET, P., GOROGIANNIS, N., AND KANOVICH, M. A compositional deadlock detector for android java. In *Proceedings of ASE-36* (2021), ACM.

[11] CAI, Y., AND CHAN, W. K. Magicfuzzer: Scalable deadlock detection for large-scale applications. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), pp. 606–616.

[12] CAI, Y., JIN, Y., AND ZHANG, C. Unleashing the power of type-based call graph construction by using regional pointer information. In *33nd USENIX Security Symposium, USENIX Security 2024, PHILADELPHIA, PA, USA, August 14-16, 2024* (2024), D. Balzarotti and W. Xu, Eds., USENIX Association.

[13] CAI, Y., MENG, R., AND PALSBERG, J. Low-overhead deadlock prediction. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (New York, NY, USA, 2020), ICSE '20, Association for Computing Machinery, p. 1298–1309.

[14] CAI, Y., YAO, P., YE, C., AND ZHANG, C. Place your locks well: Understanding and detecting lock misuse bugs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023* (2023), J. A. Calandrino and C. Troncoso, Eds., USENIX Association.

[15] CAI, Y., YE, C., SHI, Q., AND ZHANG, C. Peahen: fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022* (2022), A. Roychoudhury, C. Cadar, and M. Kim, Eds., ACM, pp. 784–796.

[16] CAI, Y., AND ZHANG, C. A cocktail approach to practical call graph construction. *Proc. ACM Program. Lang. 7*, OOPSLA2 (2023).

[17] CHEN, H., GUO, S., XUE, Y., SUI, Y., ZHANG, C., LI, Y., WANG, H., AND LIU, Y. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 2325–2342.

[18] CLANG STATIC ANALYZER. https://clang-analyzer.llvm.org/.

[19] DI, P., SUI, Y., YE, D., AND XUE, J. Region-based may-happen-in-parallel analysis for c programs. In *2015 44th International Conference on Parallel Processing* (2015), pp. 889–898.

[20] DISTEFANO, D., FÄHNDRICH, M., LOGOZZO, F., AND O'HEARN, P. W. Scaling static analyses at facebook. *Commun. ACM 62*, 8 (jul 2019), 62–70.

[21] ENGLER, D., AND ASHCRAFT, K. Racerx: Effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev. 37*, 5 (oct 2003), 237–252.

[22] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for java. PLDI '02, Association for Computing Machinery, p. 234–245.

[23] FLOWER, A. Kernel thread.

[24] GONG, S., ALTINBKEN, D., FONSECA, P., AND MANIATIS, P. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 66–83.

[25] HARDEKOPF, B., AND LIN, C. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (USA, 2011), CGO '11, IEEE Computer Society, p. 289–298.

[26] HUANG, Z., GUO, S., WU, M., AND WANG, C. Understanding concurrency vulnerabilities in linux kernel, 2022.

[27] JEONG, D. R., KIM, K., SHIVAKUMAR, B., LEE, B., AND SHIN, I. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), pp. 754–768.

[28] JIN, G., SONG, L., ZHANG, W., LU, S., AND LIBLIT, B. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds.

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

[30] JOSHI, P., NAIK, M., SEN, K., AND GAY, D. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2010), FSE '10, Association for Computing Machinery, p. 327–336.

[31] JOSHI, P., PARK, C.-S., SEN, K., AND NAIK, M. A randomized dynamic program analysis technique for detecting real deadlocks. PLDI '09, Association for Computing Machinery.

[32] KAHLON, V., YANG, Y., SANKARANARAYANAN, S., AND GUPTA, A. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2007), CAV'07, Springer-Verlag, p. 226–239.

[33] KROENING, D., POETZL, D., SCHRAMMEL, P., AND WACHTER, B. Sound static deadlock analysis for c/pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE 2016, Association for Computing Machinery, p. 379–390.

[34] KROENING, D., POETZL, D., SCHRAMMEL, P., AND WACHTER, B. Sound static deadlock analysis for c/pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE '16, Association for Computing Machinery, p. 379–390.

[35] L2D2. https://pajda.fit.vutbr.cz/xmarci10/fbinfer_concurrency.

[36] LATTNER, C., AND ADVE, V. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), pp. 75–86.

[37] LI, C., CHEN, R., WANG, B., WANG, Z., YU, T., JIANG, Y., GU, B., AND YANG, M. An empirical study on concurrency bugs in interrupt-driven embedded software. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2023), ISSTA 2023, Association for Computing Machinery, p. 1345–1356.

[38] LI, C., CHEN, R., WANG, B., YU, T., GAO, D., AND YANG, M. Precise and efficient atomicity violation detection for interrupt-driven programs via staged path pruning. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2022), ISSTA 2022, Association for Computing Machinery, p. 506–518.

[39] LI, H., HAO, Y., ZHAI, Y., AND QIAN, Z. The hitchhiker's guide to program analysis: A journey with large language models, 2023.

[40] LI, T., ELLIS, C. S., LEBECK, A. R., AND SORIN, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)* (Anaheim, CA, Apr. 2005), USENIX Association.

[41] LINUX. Commit. https://github.com/torvalds/linux/commit/7fb89eca0f2ad.

[42] LINUX KERNEL LABS. Hardirq. https://linux-kernel-labs.github.io/refs/pull/189/merge/labs/interrupts.html#implementing-an-interrupt-handler.

[43] LINUX KERNEL LABS. Interrupts. https://linux-kernel-labs.github.io/refs/heads/master/lectures/interrupts.html.

[44] LINUX KERNEL LABS. Softirqs. https://linux-kernel-labs.github.io/refs/pull/189/merge/labs/deferred_work.html#softirqs.

[45] LIU, B., AND HUANG, J. D4: fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds.

[46] LIU, B., LIU, P., LI, Y., TSAI, C.-C., DA SILVA, D., AND HUANG, J. When threads meet events: Efficient and precise static race detection with origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2021), PLDI 2021, Association for Computing Machinery, p. 725–739.

[47] LU, K., AND HU, H. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery, p. 1867–1881.

[48] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds.

[49] MICROSOFT. Azure openai service. https://azure.microsoft.com/en-us/products/ai-services/openai-service, 2021. [Online; accessed 15-July-2021].

[50] MITRE-CVE. CVE-2014-1453. https://nvd.nist.gov/vuln/detail/CVE-2014-1453.

[51] MITRE-CVE. CVE-2014-8131. https://nvd.nist.gov/vuln/detail/CVE-2014-8131.

[52] MITRE-CVE. CVE-2019-14034. https://nvd.nist.gov/vuln/detail/CVE-2019-14034.

[53] MITRE-CVE. CVE-2020-10573. https://nvd.nist.gov/vuln/detail/CVE-2020-10573.

[54] MITRE-CVE. CVE-2021-41141. https://nvd.nist.gov/vuln/detail/CVE-2021-41141.

[55] MITRE-CVE. CVE-2021-41213. https://nvd.nist.gov/vuln/detail/CVE-2021-41213.

[56] MITRE-CVE. CVE-2023-0160. https://bugzilla.redhat.com/show_bug.cgi?id=2159764.

[57] MUKHERJEE, S., DELIGIANNIS, P., BISWAS, A., AND LAL, A. Learning-based controlled concurrency testing. *Proc. ACM Program. Lang. 4*, OOPSLA (nov 2020).

[58] NG, N., AND YOSHIDA, N. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction* (New York, NY, USA, 2016), CC 2016, Association for Computing Machinery, p. 174–184.

[59] PEI, K., BIEBER, D., SHI, K., SUTTON, C., AND YIN, P. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning* (23–29 Jul 2023), A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202 of *Proceedings of Machine Learning Research*, PMLR, pp. 27496–27520.

[60] PENG, H., AND PAYER, M. Usbfuzz: A framework for fuzzing usb drivers by device emulation. In *Proceedings of the 29th USENIX Conference on Security Symposium* (USA, 2020), SEC'20, USENIX Association.

[61] ROUMELIOTIS, K. I., AND TSELIKAS, N. D. Chatgpt and open-ai models: A preliminary review. *Future Internet 15*, 6 (2023).

[62] RUIJIE MENG, MARTIN MIRCHEV, M. B. A. R. Large language model guided protocol fuzzing. In *Network and Distributed System Security Symposium (NDSS), 2024* (2024).

[63] SANTHIAR, A., AND KANADE, A. Static deadlock detection for asynchronous c# programs. *SIGPLAN Not. 52*, 6 (jun 2017), 292–305.

[64] SEREBRYANY, K., AND ISKHODZHANOV, T. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, NY, USA, 2009), WBIA '09, Association for Computing Machinery, p. 62–71.

[65] SINHA, N., AND WANG, C. Staged concurrent program analysis. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2010), FSE '10, Association for Computing Machinery, p. 47–56.

[66] SUN, S., LIU, Y., ITER, D., ZHU, C., AND IYYER, M. How does in-context learning help prompt tuning?, 2023.

[67] TAN, L., ZHOU, Y., AND PADIOLEAU, Y. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)* (2011), pp. 11–20.

[68] THE LINUX KERNEL ARCHIVES. Interrupt preemption. https://www.kernel.org/doc/html/v4.18/core-api/genericirq.html.

[69] THE LINUX KERNEL ARCHIVES. Locking. https://docs.kernel.org/locking/spinlocks.html.

[70] THE LINUX KERNEL ARCHIVES. Lockup. https://docs.kernel.org/admin-guide/lockup-watchdogs.html.

[71] THE LINUX KERNEL ARCHIVES. Netdevices. https://www.kernel.org/doc/Documentation/networking/netdevices.txt.

[72] THE LINUX KERNEL ARCHIVES. Preempt-locking. https://www.kernel.org/doc/Documentation/preempt-locking.txt.

[73] THE LINUX KERNEL ARCHIVES. Lockdep. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt, 2021.

[74] TORVALDS, L. Linux kernel git repository. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[75] WANG, Y., GAO, F., WANG, L., YU, T., ZHAO, J., AND LI, X. Automatic detection, validation, and repair of race conditions in interrupt-driven embedded software. *IEEE Transactions on Software Engineering 48*, 1 (2022), 346–363.

[76] WEN, C., CAI, Y., ZHANG, B., SU, J., XU, Z., LIU, D., QIN, S., MING, Z., AND TIAN, C. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *the ACM Transactions on Knowledge Discovery from Data.* (2024).

[77] WEN, C., HE, M., WU, B., XU, Z., AND QIN, S. Controlled concurrency testing via periodical scheduling. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022* (2022), ACM, pp. 474–486.

[78] WIKIPEDIA. Critical section. https://en.wikipedia.org/wiki/Critical_section, 2021. [Online; accessed 15-July-2021].

[79] XIE, Y., AND AIKEN, A. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds.

[80] XU, M., KASHYAP, S., ZHAO, H., AND KIM, T. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), pp. 1643–1660.

[81] YUAN, M., ZHAO, B., LI, P., LIANG, J., HAN, X., LUO, X., AND ZHANG, C. Ddrace: Finding concurrency UAF vulnerabilities in linux drivers with directed fuzzing. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023* (2023), J. A. Calandrino and C. Troncoso, Eds., USENIX Association.

[82] Z3. https://github.com/Z3Prover/z3.

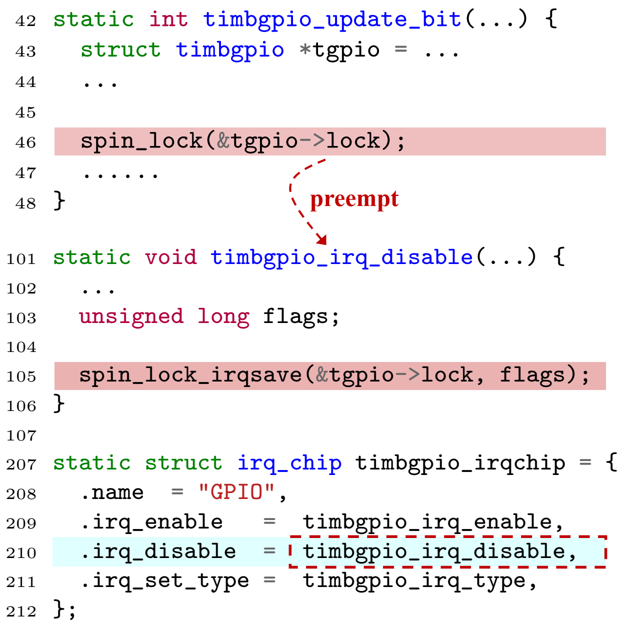
[83] ZHAN, S., AND HUANG, J. ECHO: instantaneous in situ race detection in the IDE. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds.

[84] ZHOU, J., SILVESTRO, S., LIU, H., CAI, Y., AND LIU, T. Undead: Detecting and preventing deadlocks in production software. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), pp. 729–740.

[85] ZHOU, J., YANG, H., LANGE, J., AND LIU, T. Deadlock prediction via generalized dependency. ISSTA 2022, Association for Computing Machinery.

A Appendix

A.1 Case Study of a Real Bug Detected with LLM-Assisted Specification



```
 42 static int timbgpio_update_bit(...) {
 43   struct timbgpio *tgpio = ...
 44   ...
 45
 46   spin_lock(&tgpio->lock);
 47   ......
 48 }

101 static void timbgpio_irq_disable(...) {
102   ...
103   unsigned long flags;
104
105   spin_lock_irqsave(&tgpio->lock, flags);
106 }
107
207 static struct irq_chip timbgpio_irqchip = {
208   .name   = "GPIO",
209   .irq_enable   =  timbgpio_irq_enable,
210   .irq_disable  =  timbgpio_irq_disable,
211   .irq_set_type =  timbgpio_irq_type,
212 };
```

Figure 13: A new interrupt-based deadlock detected after applying LLM-assisted specification.

Figure 13 shows one of the newly detected bugs after applying LLM-assisted specification. Since the function `timbgpio_update_bit()` acquires `tgpio→lock` under process context at Line 46, `timbgpio_irq_disable()` could preempt the execution and acquire the lock again under interrupt context at Line 105. In such a scenario, the CPU would spin on the lock infinitely, resulting in a system hang. The key to detecting this bug is the expert knowledge that `timbgpio_irq_diable()` could be executed under the interrupt context, which is ignored by manual modeling. As `timbgpio_irq_diable()` is registered via the interface `irq_chip→irq_disable`, ChatGPT-4.0 can infer that this callback interface is executed during interrupt handling, enabling Archerfish to detect this self-cyclic interrupt-based deadlock.

A.2 Details of APIs used in Static Lockset and Interrupt-State analysis

Table 3 shows the list of APIs modeled for the lockset and interrupt-state analysis. Next, we illustrate some details of the implementation to model these APIs.

- **Lock/Unlock:** We model three kinds of commonly used locks inside the interrupt context, including `spin_lock`, `read_lock`, and `write_lock`. One special case in modeling is that read_lock operations allow concurrent accesses, whereas write_lock operations require exclusive accesses. As a result, two read_lock acquisitions do not introduce lock dependency, and we model these special cases in our tool.
- **Hardirq Enable/Disable:** There are several special cases in the API modeling. First, different APIs could disable/enable different sets of ISRs. The APIs `*_lock/unlock_irq()` and `*_local_irq_disable/enable()` operate on all the hardirq ISRs in the system, while APIs `enable/disable_irq()` only operate on a designated hardirq specified by an irq line number. As a result, we perform different operations on different APIs to model these differences. Second, APIs end with `_irqsave()` or `_irqrestore()` suffix would store/load the interrupt state into/from a local flag variable. Thus, we perform a local data-flow analysis to track the interrupt state stored inside the flag variable.
- **Softirq Enable/Disable:** Compared with hardirq, the modeling for softirq is relatively simple. Specifically, each one of these APIs would disable/enable all the softirq ISRs in the system.

A.3 Details of Manually Modeled ISR Registration APIs in the Linux Kernel

Table 4 shows some standard APIs used by all the subsystems in Linux. Since there are well-defined documentations [42, 44], we can manually model them in the implementation.

- **Hardirq:** Standard APIs such as request_irq() are used for hardware interrupt handler registration, and they are well-defined in the documentation of Linux Kernel [42]. Thus, we can manually model these commonly used hardware ISR registration APIs.
- **Softirq:** The timer, tasklet, and polling are well-known types of software interrupt handlers in the Linux kernel, and corresponding registration APIs are well-defined in the Linux Kernel documentation [44]. We also manually model the corresponding APIs.

For subsystem APIs or interfaces that are not well-defined, we resort to LLM-assisted specifications.

Table 3: List of manually modeled APIs for the lockset and interrupt-state analysis.

Category	**APIs**
Lock	spin/read/write_lock() spin/read/write_lock_irq() spin/read/write_lock_irqsave() spin/read/write_lock_bh()
Unlock	spin/read/write_unlock() spin/read/write_unlock_irq() spin/read/write_unlock_irqrestore() spin/read/write_unlock_bh()

Hardirq Enable	spin/read/write_unlock_irq()
	spin/read/write_unlock_irqrestore()
	enable_irq_nosync()
	local_irq_enable(), enable_irq()

Hardirq Disable	spin/read/write_lock_irq()
	spin/read/write_lock_irqsave()
	disable_irq_nosync()
	local_irq_disable(), disable_irq()

Softirq Enable	spin/read/write_unlock_bh() local_bh_enable()
Softirq Disable	spin/read/write_lock_bh() local_bh_disable()

Table 4: Manually modeled APIs for ISR registration.

Category	**APIs**

Hardirq Registration	devm_request_irq()
	devm_request_threaded_irq()
	request_irq(), request_threaded_irq()
	request_percpu_irq()

Softirq Registration	tasklet_init(), timer_setupt()
	irq_poll_init(), open_softirq()

A.4 Detected Deadlocks

Table 5 presents a list of all the bugs detected. Two bugs detected inside SCTP and TIPC subsystems are assigned CVE-2024-0639 and CVE-2024-0641, respectively.

Table 5: List of 76 interrupt-based deadlocks detected by Archerfish on Linux kernel v6.4. Each row displays the subsystem name, the preempted function that introduces the deadlock, whether the bug is detected with the help of LLM-assisted specifications, and the status of the bug. ● denotes the corresponding bug-fixing patch is already merged into Linux, ◐ denotes developers already confirmed the bug by adding "Acked-by" or "Reviewed-by" tags under the patch and prepared to accept the patch, and ○ denotes the patch is still under review.

System	Function	LLM?	Status
i2c	iproc_i2c_rd_reg()	✘	●
i2c	iproc_i2c_wr_reg()	✘	●
atm	console_show()	✘	●
atm	pclose()	✘	●
gpio	timbgpio_update_bit()	✔	●
dma	sun6i_dma_pause()	✘	○
dma	sun6i_dma_terminate_all()	✘	○
dma	pd_tx_submit()	✘	○
dma	pdc_desc_get()	✘	○
dma	pdc_desc_put()	✘	○
dma	pd_issue_pending()	✘	○
dma	pdc_desc_get()	✘	○
dma	milbeaut_hdmac_chan_config()	✘	◐
dma	milbeaut_hdmac_chan_pause()	✘	◐
dma	milbeaut_hdmac_chan_resume()	✘	◐
dma	plx_dma_process_desc()	✘	◐
dma	xgene_dma_cleanup_descriptors()	✘	◐
dma	k3_dma_terminate_all()	✘	○
dma	k3_dma_transfer_pause()	✘	○
dma	mtk_hsdma_free_active_desc()	✘	○
staging	hostif_sme_work()	✔	●
staging	rtsx_exclusive_enter_ss()	✘	○
char	kcs_bmc_handle_event()	✘	●
IB	read_mod_write()	✔	●
IB	ib_device_get_netdev()	✔	○
media	enc_post_frame_start()	✘	●
media	dvb_dmxdev_ts_callback()	✘	○
media	dvb_dmxdev_section_callback()	✘	○
mfd	pm8xxx_config_irq()	✔	●
mfd	pm8xxx_irq_set_type()	✔	●
misc	bcm_vk_reset()	✘	●
misc	bcm_vk_blk_drv_access()	✘	●
misc	bcm_vk_get_ctx()	✘	●
misc	bcm_vk_free_ctx()	✘	●
isdn	_hfcpci_softirq()	✘	●
watchdog	s3c2410wdt_stop()	✘	●
watchdog	s3c2410wdt_keepalive()	✘	●
watchdog	s3c2410wdt_start()	✘	●
tty	imx_uart_dma_rx_callback()	✘	◐
tty	slgt_compat_ioctl()	✘	●
scsi	mvs_slot_complete()	✔	○
scsi	mvs_slot_complete()	✔	○
scsi	qedi_cpu_offline()	✘	●
scsi	megaraid_reset_handler()	✘	○
scsi	qedi_tmf_resp_work()	✘	○
scsi	ips_eh_abort()	✘	○
scsi	hisi_sas_release_task()	✘	○
scsi	hisi_sas_slot_task_free()	✘	○
scsi	hisi_sas_task_deliver()	✘	○
scsi	hisi_sas_slot_index_free()	✘	●
scsi	hisi_sas_slot_index_alloc()	✘	●
scsi	hisi_sas_alloc_dev()	✘	●
scsi	fcoe_ctlr_flogi_send()	✔	●
scsi	fcoe_ctlr_flogi_retry()	✔	●
scsi	fcoe_ctlr_announce()	✔	●
scsi	fcoe_ctlr_els_send()	✔	●
scsi	mvs_find_dev_mvi()	✘	○
ocfs2	o2quo_make_decision()	✘	●
ocfs2	o2quo_hb_up()	✘	●
ocfs2	o2quo_hb_down()	✘	●
ocfs2	o2quo_hb_still_up()	✘	●
ocfs2	o2quo_conn_up()	✘	●
ocfs2	o2quo_conn_err()	✘	●
ocfs2	o2net_debug_add_nst()	✘	●
ocfs2	o2net_debug_del_nst()	✘	●
ocfs2	nst_seq_start()	✘	●
ocfs2	nst_seq_next()	✘	●
ocfs2	nst_seq_show()	✘	●
ocfs2	o2net_debug_add_sc()	✘	●
ocfs2	o2net_debug_del_sc()	✘	●
ocfs2	sc_seq_start()	✘	●
ocfs2	sc_seq_next()	✘	●
ocfs2	sc_seq_show()	✘	●
sctp	sctp_auto_asconf_init()	✘	●
ax25	ax25_check_dama_slave()	✘	◐
tipc	tipc_crypto_key_revoke()	✘	●
Total			**76**