Detecting Kernel Memory Bugs through Inconsistent Memory Management Intention Inferences

Dinghao Liu, Zhipeng Lu, and Shouling Ji, *Zhejiang University;* Kangjie Lu, *University of Minnesota;* Jianhai Chen and Zhenguang Liu, *Zhejiang University;* Dexin Liu, *Peking University;* Renyi Cai, *Alibaba Cloud Computing Co., Ltd;* Qinming He, *Zhejiang University*

https://www.usenix.org/conference/usenixsecurity24/presentation/liu-dinghao-detecting

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

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Detecting Kernel Memory Bugs through Inconsistent Memory Management Intention Inferences

Dinghao Liu[1], Zhipeng Lu[1], Shouling Ji[1], Kangjie Lu[2], Jianhai Chen[1], Zhenguang Liu[1], Dexin Liu[3], Renyi Cai[4], and Qinming He[1]

[1]Zhejiang University, [2]University of Minnesota, [3]Peking University, [4]Alibaba Cloud Computing Co., Ltd

E-mails: {dinghao.liu, alexious, sji}@zju.edu.cn, kjlu@umn.edu,
chenjh919@zju.edu.cn, liuzhenguang2008@gmail.com, dxliu@pku.edu.cn, renyi.cry@alibaba-inc.com, hqm@zju.edu.cn

Abstract

Modern operating system kernels, typically written in low-level languages such as C and C++, are tasked with managing extensive memory resources. Memory-related errors, such as memory leak and memory corruption, are common occurrences and constantly introduced. Traditional detection methods often rely on taint analysis, which suffer from scalability issues (i.e., path explosion) when applied to complex OS kernels. Recent research has pivoted towards leveraging techniques like function pairing or similarity analysis to overcome this challenge. These approaches identify memory errors by referencing code that is either frequently used or semantically similar. However, these techniques have limitations when applied to customized code, which may lack a sufficient corpus of code snippets to facilitate effective function pairing or similarity analysis. This deficiency hinders their applicability in kernel analysis where unique or proprietary code is prevalent.

In this paper, we propose a novel methodology for detecting memory bugs based on inconsistent memory management intentions (`IMMI`). Our insight is that many memory bugs, despite their varied manifestations, stem from a common underlying issue: the ambiguity in ownership and lifecycle management of memory objects, especially when these objects are passed across various functions. Memory bugs emerge when the memory management strategies of the caller and callee functions misalign for a given memory object. `IMMI` aims to model and clarify these inconsistent intentions, thereby mitigating the prevalence of such bugs. Our methodology offers two primary advantages over existing techniques: (1) It utilizes a fine-grained memory management model that obviates the need for extensive data-flow tracking, and (2) it does not rely on similarity analysis or the identification of function pairs, making it highly effective in the context of customized code. To enhance the capabilities of `IMMI`, we have integrated a large language model (LLM) to assist in the interpretation of implicit kernel resource management mechanisms. We have implemented `IMMI` and evaluated it against the Linux kernel. `IMMI` effectively found 80 new memory bugs (including 23 memory corruptions and 57 memory leaks) with 35% false discovery rate. Most of them are missed by the state-of-the-art memory bug detection tools.

Shouling Ji and Qinming He are the co-corresponding authors.

1 Introduction

The operating system (OS) kernel constitutes a foundational component of modern computing systems, serving as the critical intermediary between hardware and user-level applications. Within the kernel, memory resources are ubiquitously employed to maintain data structures, manage I/O buffers, and facilitate inter-process communication, thereby highlighting their essentiality in kernel functionality. However, the complexity inherent in memory management introduces a multitude of memory-related bugs such as memory leaks, use-after-free (UAF), and double-free. These defects not only compromise system stability and performance by leading to resource exhaustion and unpredictable behavior, but also pose severe security threats. Exploitation of such vulnerabilities can result in unauthorized access, privilege escalation, and ultimately, system compromise [21, 35, 37, 39]. Given the universality of memory objects in the kernel space and the potential for widespread harm, it is imperative to address these memory bugs with rigorous and systematic approaches.

The most direct way to detect the aforementioned memory bugs is to statically track the data-flows of memory objects, which is commonly employed in the analysis of user-space applications [4, 7, 38, 42]. For instance, a memory leak is identified when a memory object remains unreleased upon program termination, whereas use-after-free (UAF) or double-free bugs arise when a memory object is accessed or deallocated subsequent to its initial release without any intervening reinitialization. However, the application of such a straightforward approach to OS kernels is impeded by significant scalability challenges. The vast codebase (e.g., the latest Linux kernel comprises approximately 28 million source code lines) introduces an overwhelming number of execution paths,

culminating in path and state explosion problems.

To mitigate this issue, practical kernel analysis tools instead utilize *function pairs* for memory bug detection [2, 16, 17, 34]. Specifically, OS kernels commonly use abundant memory management (MM) functions (e.g., `kmalloc` and `kfree`) to orchestrate the lifecycle of dynamically allocated memory objects. These MM functions are intended to operate in concert, reflecting the allocation and subsequent deallocation stages inherent to the lifecycle of memory objects. Tools like K-Meld [9] and Hector [26] identify memory leaks by scrutinizing the absence of the corresponding deallocation functions in these pairs. SinkFinder [3] and NLP_EYE [30] leverage MM functions to simplify the data-flow analysis in memory corruption detection. HERO [34] is specifically tailored to identify bugs that emerge from the improper sequencing of function pair usage. Some tools, on the other hand, employ similarity analysis to address the scalability challenges in kernel analysis. For example, IPPO [15] and NDI [46] detect memory bugs by identifying inconsistent memory operations across program paths that are semantically analogous.

These approaches have demonstrated their effectiveness in identifying memory bugs within OS kernels. However, they still suffer from several limitations. Firstly, there is an inherent assumption that the implementation of function pairs is intrinsically correct. This presumption does not always hold true within the complex landscape of OS kernels. In instances where a function pair harbors internal defects, such as a deallocation function failing to fully relinquish the memory resources allocated by its counterpart, the presence of a bug persists despite the ostensibly correct usage of the function pairs. Secondly, these approaches heavily rely on the availability of existing *correct* code snippets to facilitate effective MM function pairing or to perform similarity analysis. Tools such as HERO [34] and ErrDoc [29] identify function pairs within error handling paths by examining commonly paired functions. Similarly, IPPO [15], NDI [46], MLEE [31], and Hector [26] require at least one intra-procedural path that includes the anticipated memory release operation for accurate bug detection. Given the extensive amount of specialized code in OS kernels, many function pairs or code paths are unique, lacking appropriate or analogous references for comparison.

Existing research indicates that memory bugs frequently manifest within error handling paths [11, 12, 18, 29, 34, 44], where programs attempt to release previously allocated memory resources to revert to prior states. Ideally, there should be a consensus among all functions involved regarding the responsibility for deallocating a memory object upon a particular failure. Our key observation is that a multitude of memory bugs, despite their diverse symptoms, originate from a fundamental issue: the lack of clarity in ownership and lifecycle management of memory objects when premature release is necessitated. This is particularly problematic when memory objects are shared across multiple functions. For instance, if both caller and callee functions presume responsibility for managing a memory object during error handling, a double-free or UAF could happen, ultimately resulting in a memory corruption. Conversely, if neither assumes this responsibility, the memory remains occupied, leading to a memory leak. In essence, there exists *inconsistent intentions* regarding the management of memory objects.

Based on this insight, we introduce Inconsistent Memory Management Intentions (`IMMI`) as a heuristic for memory bug detection. We present an object-based method, augmented by Large Language Models (LLMs), to deduce MM intentions. The co-analysis leverages both the logical structure of the program and the natural language elements embedded within the code (e.g., code comments and implicit MM mechanisms). This enables precise inference of MM intentions within kernel-space at a fine granularity. Additionally, we propose several code slicing and summarization techniques to balance the precision and efficiency of `IMMI`. `IMMI` offers several distinct advantages over existing methods. Firstly, `IMMI`'s inference of MM intentions requires the analysis of only a limited set of code paths and call traces, thereby obviating the need for extensive data-flow analysis across the kernel. Secondly, `IMMI` operates independently of frequently used or semantically similar code snippets, thus maintaining efficacy in analyzing customized code. Additionally, `IMMI` does not rely on function pairs to identify bugs, avoiding the constraints associated with such methods.

We implement `IMMI` based on LLVM and apply it to the Linux kernel. Remarkably, `IMMI` completes its analysis of the entire kernel within 35 minutes, demonstrating its efficiency and scalability. As of the submission of this paper, `IMMI` has identified 80 new bugs, including 57 memory leak bugs, 6 UAF bugs, 16 double-free bugs and 1 null-pointer-dereference bug. Most of them have been confirmed or fixed by community maintainers with our patches. `IMMI` exhibits a false discovery rate of 35%. The incorporation of LLM effectively eliminates 43% false positives reported by `IMMI`, while only incurring an additional 4% in false negatives. We plan to open-source `IMMI` to facilitate bug detection in more projects. In summary, we make the following contributions.

- **A new approach for memory bug detection.** We introduce `IMMI`, a heuristic-based method that identifies memory bugs by analyzing inconsistencies in MM intentions. Compared to existing approaches, `IMMI` does not require long-term data-flow tracking, or frequently used or similar code snippets, thus is effective in analyzing customized code.

- **New techniques for MM intention inference.** We propose multiple techniques in implementing `IMMI`. Firstly, we propose an object-based analysis framework designed to infer MM intentions with fine granularity. Secondly, we introduce an object-based slicing methodology and a novel demand-driven function summarization technique, to enhance the efficacy of our analysis. Thirdly, we incorporate

```c
/* net/smc/smc_er.c */
int smc_wr_alloc_link_mem(struct smc_link *link)
{
    link->wr_tx_bufs = kcalloc(...);
    if (!link->wr_tx_bufs)
        goto no_mem;
    link->wr_rx_bufs = kcalloc(...);
    if (!link->wr_rx_bufs)
        goto no_mem_wr_tx_bufs;
    link->wr_tx_ibs = kcalloc(...);
    if (!link->wr_tx_ibs)
        goto no_mem_wr_rx_bufs;
    ...
    return 0;
    ...
no_mem_wr_rx_bufs:
    kfree(link->wr_rx_bufs);
no_mem_wr_tx_bufs:
    kfree(link->wr_tx_bufs);
no_mem:
    return -ENOMEM;
}
```

Figure 1: Example of callee-based management.

LLM into our framework to augment the capabilities of IMMI significantly.

- **New bugs in real-world OS kernel.** We have found 80 new memory bugs in the widely-used Linux kernel with the help of IMMI. We have reported these bugs, and most of them have been confirmed or fixed by working with the community maintainers. This results substantiate the efficacy of IMMI as a tool for enhancing OS kernel security.

2 Background and Study

2.1 MM Strategies of OS Kernels

OS kernels are responsible for managing substantial memory resources. In the event of an error, it is imperative for kernel functions to perform cleanup operations, reverting the system to its pre-error state (i.e., *error handling*). Through an extensive examination of the Linux kernel code, we have identified two primary categories of error-induced memory management strategies: callee-based management and caller-based management.

Callee-based management. Figure 1 shows a typical example of callee-based management in the Linux kernel. Specifically, `smc_wr_alloc_link_mem` is responsible for allocating multiple dynamic memory objects by calling `kzalloc`. In the event of an allocation failure, the function employs a structured jump to designated labels, ensuring the comprehensive deallocation of all previously allocated memory. Consequently, its caller can safely assume that, in the event of a failure in `smc_wr_alloc_link_mem`, no partial allocations remain. This eliminates the necessity for the caller to perform additional cleanup, allowing it to concentrate solely on the resources allocated prior to the call.

Caller-based management. Contrary to callee-based management, certain paradigms shift the onus of resource deallocation to the callers, as illustrated in Figure 2. In this example, `tm6000_alloc_urb_buffers` also allocates multiple memory resources using `kmalloc_array`. However, should an error transpire, it merely returns an error code without performing any cleanup. As a result, the onus falls upon the caller (`tm6000_prepare_isoc` in this example) to execute the appropriate release function (`tm6000_free_urb_buffers`) to free the allocated memory when `tm6000_alloc_urb_buffers` fails. While this approach may simplify the function's control flows by reducing the complexity of jump labels, it concurrently burdens the callers with augmented resource management duties.

```c
/* drivers/media/usb/tm6000/tm6000-video.c */
static int tm6000_alloc_urb_buffers(struct tm6000_core *dev)
{
    ...
    dev->urb_buffer = kmalloc_array(...);
    if (!dev->urb_buffer)
        return -ENOMEM;

    dev->urb_dma = kmalloc_array(...);
    if (!dev->urb_dma)
        return -ENOMEM;
    ...
    if (!dev->urb_buffer[i]) {
        tm6000_err("unable to allocate ... buffer %i\n",
                dev->urb_size, i);
        return -ENOMEM;
    }
    ...
    return 0;
}

static int tm6000_prepare_isoc(struct tm6000_core *dev)
{
    ...
    if (tm6000_alloc_urb_buffers(dev) < 0) {
        tm6000_err("cannot allocate memory for urb buffers\n");

        /* call free, as some buffers might have been allocated */
        tm6000_free_urb_buffers(dev);
        ...
        return -ENOMEM;
    }
    ...
}
```

Figure 2: Example of caller-based management.

2.2 Impact of Inconsisitent MM Intentions

Both MM strategies are extensively employed to facilitate comprehensive error handling. Nevertheless, when disparate functions exhibit inconsistent MM intentions for a particular memory object, the potential for memory bugs increases.

Memory corruption. When a callee function employs callee-based management while its caller adopts caller-based management, there exists a potential for a memory object to be accessed and subsequently deallocated following its initial release. Such scenarios frequently precipitate UAF or double-free vulnerabilities, which can lead to system crashes. Furthermore, these forms of memory corruption can be exploited to orchestrate sophisticated attacks, ultimately compromising the integrity of the system and allowing adversaries to seize control [6, 21].

Memory leak. Contrary to previous scenarios, another inconsistency arises when a callee function utilizes caller-based management while its caller adopts callee-based management. This inconsistency can result in a memory object being neglected, potentially culminating in a memory leak. Such leaks are deemed security critical within OS kernels, as they may precipitate denial-of-service (DoS) attacks, thereby rendering the entire system inoperative [9, 31].

2.3 Causes of Inconsisitent MM Intentions

In this section, we summarize three main causes of inconsistent MM intentions based on our empirical analysis of the Linux kernel and corresponding patches.

Complex logic in error handling. One of the predominant factors contributing to MM inconsistencies is the intricate error handling logic, which is known to be error-prone. A single kernel function may be responsible for managing multiple memory resources, each necessitating distinct MM strategies. Additionally, the complexity is exacerbated by the widespread sharing and transfer of memory resources among different modules and layers of abstraction. Such interactions necessitate collaboration among developers with disparate coding practices on managing the same memory objects, potentially resulting in discrepancies in the understanding and implementation of MM strategies.

Imperfect code updating. The OS kernel is a dynamic entity, subject to continuous evolution through frequent updates, with a multitude of patches being integrated regularly. These updates often originate from a diverse group of developers, including those who may not have been involved in the original development or maintenance of the codebase (e.g., security analysts who provide bug fixing patches). Contributors who lack a deep understanding of the existing code logic may inadvertently make assumptions about MM strategies that diverge from the intended design, thereby introducing potential vulnerabilities into the system.

Implicit OS MM mechanisms. Modern OS kernels have developed advanced MM mechanisms to simplify resource management for kernel developers. For instance, the Linux kernel employs reference counting, indirect calls, and compiler-assisted techniques to facilitate *automated memory resource management*, thereby obviating the need for explicit resource release operations. When memory resources governed by these mechanisms are intermingled with conventional resources, or when transitioning resources to these mechanisms, the potential for developer error escalates significantly.

2.4 Motivation of IMMI

Figure 3 illustrates a memory leak bug identified by IMMI in the Linux kernel. In this example, the memory object `qdev->lrg_buf` (line 5) allocated in `ql_alloc_buffer_queues` is not released, leading to a

```
1  /* drivers/net/ethernet/qlogic/qla3xxx.c */
2  static int ql_alloc_buffer_queues(struct ql3_adapter *qdev)
3  {
4      ...
5      qdev->lrg_buf = kmalloc_array(...);
6      if (qdev->lrg_buf == NULL)
7          return -ENOMEM;
8      ...
9      if (qdev->lrg_buf_q_alloc_virt_addr == NULL) {
10         netdev_err(qdev->ndev, "lBufQ failed\n");
11         return -ENOMEM;
12     }
13     ...
14     return 0;
15 }
16
17 static int ql_alloc_mem_resources(struct ql3_adapter *qdev)
18 {
19     ...
20     if (ql_alloc_net_req_rsp_queues(qdev) != 0) {
21         netdev_err(qdev->ndev, ...);
22         goto err_req_rsp;
23     }
24
25     if (ql_alloc_buffer_queues(qdev) != 0) {
26         netdev_err(qdev->ndev, ...);
27         goto err_buffer_queues;
28     }
29
30     if (ql_alloc_small_buffers(qdev) != 0) {
31         netdev_err(qdev->ndev, ...);
32         goto err_small_buffers;
33     }
34     ...
35     return 0;
36     ...
37 err_small_buffers:
38     ql_free_buffer_queues(qdev);
39 err_buffer_queues:
40     ql_free_net_req_rsp_queues(qdev);
41 err_req_rsp:
42     ...
43     return -ENOMEM;
44 }
```

Figure 3: A memory leak bug found by IMMI in the Linux kernel.

memory leak if an error occurs subsequent to resource allocation, as indicated by the error code returned at line 11. Existing approaches for detecting memory leaks typically involve tracking data-flows associated with a specific memory object [38, 41, 42] or referring to existing code snippets to deduce proper usage patterns [15–17, 43]. However, exhaustively tracing all potential paths to identify a definitive path that neglects to release memory across the entire kernel is impractical. In practice, many tools focus solely on intra-procedural contexts [9, 15, 23], presuming that callers will invariably manage errors and perform necessary cleanup correctly, which is frequently invalidated. Moreover, in the presented example, there is a lack of a reference for resource deallocation (e.g., a release operation in alternative error-handling paths of `ql_alloc_buffer_queues`), which would result in its omission by current memory bug detection methodologies.

When examining this bug through the lens of MM intentions, it becomes apparent that the absence of a release operation implies an expectation that the caller of `ql_alloc_buffer_queues` (line 17) is responsible for clean-

ing up `qdev->lrg_buf` upon failure. However, the caller function does not address this failure, instead deferring the cleanup of associated memory to a subsequent failure scenario (i.e., invoking `ql_free_buffer_queues` at line 38 to deallocate `qdev->lrg_buf` when `ql_alloc_small_buffers` fails). This indicates an inconsistency between the assumptions of the callee function and its caller regarding responsibility for memory management, leading to the observed memory leak. Our analysis suggests that a more holistic approach, one that considers the intentions inherent in MM practices, is necessary to effectively detect and prevent such memory-related bugs.

3 Overview

3.1 Challenges in MM Intention Inference

The design of `IMMI` is inspired by the insight that many kernel memory bugs come from the inconsistency of MM intentions. Though the idea is intuitive, there are several technical challenges.

How to infer MM intentions? MM intention inference necessitates a sophisticated comprehension of kernel memory operations and the ability to identify indicative patterns of memory utilization intentions. This task involves dissecting complex logic that dictates kernel behavior, which is non-trivial. Additionally, the implicit memory management mechanisms embedded within OS kernels are not only pervasive but also intricate, demanding a substantial depth of domain-specific knowledge for precise interpretation. As a result, these mechanisms frequently escape detection by conventional bug detection tools. Furthermore, MM intentions could be concealed within code comments, necessitating a level of natural language understanding that typically exceeds the capabilities of existing static program analysis techniques.

To address this challenge, we introduce a novel approach that synergizes behavior-based MM intention analysis with enhancements derived from large language models (LLMs). Our approach involves analyzing and extracting MM intentions from error handling logic, while concurrently leveraging LLMs to interpret pertinent code comments and implicit kernel MM mechanisms. `IMMI` aims to bridge the gap between the explicit code and the implicit intentions of memory management, thereby improving the accuracy and efficacy of kernel memory bug detection.

How to balance precision and efficiency? Achieving a high level of precision in MM analysis requires exhaustive and meticulous scrutiny, which could be both resource-intensive and protracted. For instance, identifying the memory leak in Figure 3 necessitates an inter-procedural, path-sensitive, and field-sensitive analysis approach, which is particularly burdensome when examining large-scale OS kernels. On the other hand, compromising on precision would result in a significant increase in both false positives and false negatives.

To address this challenge, we introduce several techniques aimed at bolstering the efficiency and scalability of our static analysis, while maintaining a high level of precision. Specifically, we propose an object-based program slicing method to confine the scope of our analysis. We also present a novel approach for summarizing memory operations that optimizes inter-procedural analysis. Diverging from conventional function-level summary generation, our method segments summaries into discrete units and reconstructs them on-demand. This technique promises to deliver accurate guidance on memory operations without the need for costly inter-procedural alias analysis.

3.2 The `IMMI` Framework

Figure 4 illustrates the workflow of `IMMI`, which consists of three phases. In the first phase, `IMMI` accepts LLVM Intermediate Representation (IR) files as input. `IMMI` constructs control-flow graphs (CFGs) for individual functions and a global call graph for the entire system. `IMMI` also implements a specialized alias analysis to determine aliasing relationships within functions.

In the second phase, `IMMI` first constructs memory operation summaries to facilitate following MM intention inference. Following this step, `IMMI` proceeds to extract memory objects in the target program and generates program slices centered based on these objects. By analyzing these slices in conjunction with the memory operation summaries, `IMMI` infers the MM intentions at the point of allocation within the respective functions. If `IMMI` detects divergent MM intentions for a single memory object within a function, it classifies this as a bug (i.e., *intro-inconsistency*), reports it immediately, and halts further analysis.

In the third phase, `IMMI` performs a backward inconsistency analysis for memory bug detection. This phase initiates with a verification step to ascertain whether a memory object has been propagated to the caller functions. If propagation is confirmed, `IMMI` meticulously analyzes the MM intentions for the caller functions in relation to the memory object. If an inconsistency arises between the MM intentions deduced in the callees and those in the callers, `IMMI` flags this as a potential memory bug (i.e., *inter-inconsistency*). `IMMI` persistently monitors the memory object across successive calling functions if the immediate caller does not deduce a definitive MM intention but rather delegates the MM responsibility to subsequent callers, thereby ensuring a comprehensive and thorough analysis.

4 System Design

4.1 Alias Analysis

To enhance the analysis of data-flows associated with memory objects, we employed *path-based alias analysis* [14], which is based on the *alias graph*. This data structure comprises nodes

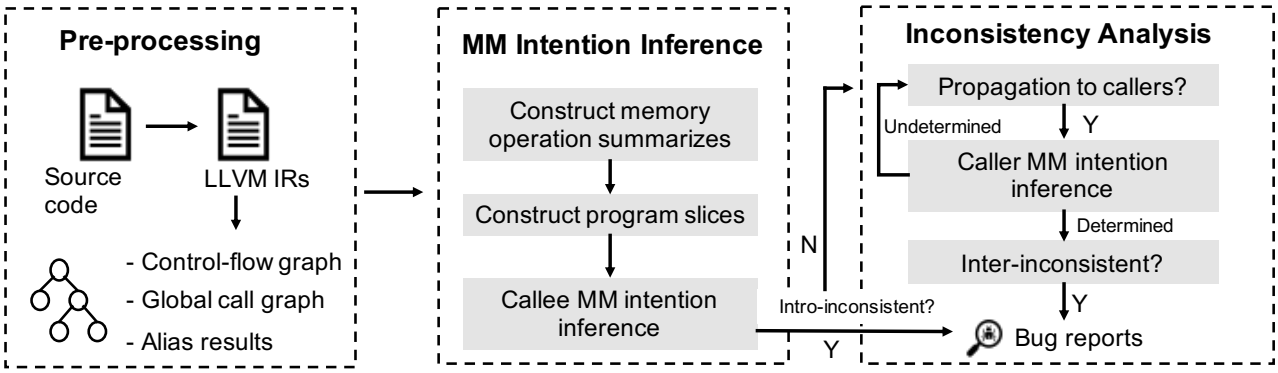

Figure 4: The overview of IMMI.

representing alias sets with edges that illustrate the relationships between pointers of discrete alias sets, such as pointer dereferencing and struct field accesses. A significant benefit of this approach is its capability to record pointer relationships comprehensively during the analysis. The approach is desiged to be inter-procedural, flow-sensitive, and field-sensitive.

To augment the efficiency and accuracy of our analysis, we reformed the alias updating algorithm to a flow-insensitive version, confining its use to intra-procedural contexts. The incorporation of inter-procedural information is achieved through a demand-driven summarization method, which will be expounded upon subsequently. Our approach limits alias analysis to the processes of summary computation and analysis of specific code slices, averting the demands of an extensive, full-program alias analysis.

Following the completion of alias analysis, the data-flow relationships among distinct pointers are succinctly recorded within the alias graph. It is pertinent to acknowledge that for array types, the indices associated with element access frequently vary and are non-static, presenting difficulties for static analysis. Therefore, our alias analysis conservatively assumes that elements within the same array act as aliases.

4.2 Memory Operation Summarization

Program summarization is an important technique in static analysis for streamlining inter-procedural analysis. In this paper, we focus on two primary types of memory operations within our summarization framework: resource release and resource resetting. Resource release is an essential element for inferring MM intentions. On the other hand, resource resetting significantly impacts the determination of potential bugs. For instance, reassigning a pointer that denotes a memory object typically signifies the end of that object's lifetime. Additionally, nullifying a pointer post-release, a practice known as *defensive programming*, ensures that subsequent deallocations of the same memory object are not misconstrued as double-free errors, given that deallocating a null pointer in the kernel is deemed safe.

Challenges. Current summarization techniques employed for OS kernel analysis are either excessively coarse-grained or prohibitively expensive. For instance, numerous memory bug detection tools operate under the assumption that memory allocations and deallocations have a one-to-one correspondence, an assumption frequently invalidated by one-to-many and many-to-many relationships. Goshawk [22] introduces a structure-aware and object-centric approach to memory summarization. Nonetheless, this strategy incurs significant overhead. Kernel developers typically implement memory management in a layered fashion, leading to extensive call chains. Consequently, memory summaries, especially for higher-level functions such as driver probe routines, tend to represent an excessive amount of memory resources. On the other hand, the generation of summaries that are not subsequently utilized in bug detection results in considerable waste of computational and storage resources.

Our solution: demand-driven summarization. To address the aforementioned challenges, we present a demand-driven memory operation summarization technique. Our approach involves decomposing memory operation summaries into discrete pairwise interactions, which are then cataloged within a global map. Each entry in this map meticulously records the intricate details of these interactions, which is structure-aware. When subsequent analysis necessitates a query of memory operations for a specific function, IMMI dynamically reconstructs the comprehensive summarization from these pairwise entries. This method allows for precise and targeted analysis without incurring the prohibitive costs associated with traditional summarization techniques

Definition 1. *In this work, a discrete memory operation summary is defined as a tuple:* $S = \{F, Arg, L_O\}$*, where* F *denotes the function encapsulating the summary,* Arg *represents the function's argument associated with the targeted memory objects, and* L_O *is a list that chronicles the memory operations executed within* F*. More precisely, each element in* L_O *is characterized by a tuple:* $\{O_{type}, O_{value}\}$*, with* O_{type} *specifies the operation type, which for the scope of this paper, is confined to either 'Release' or 'Resetting'. For 'Release' operations,*

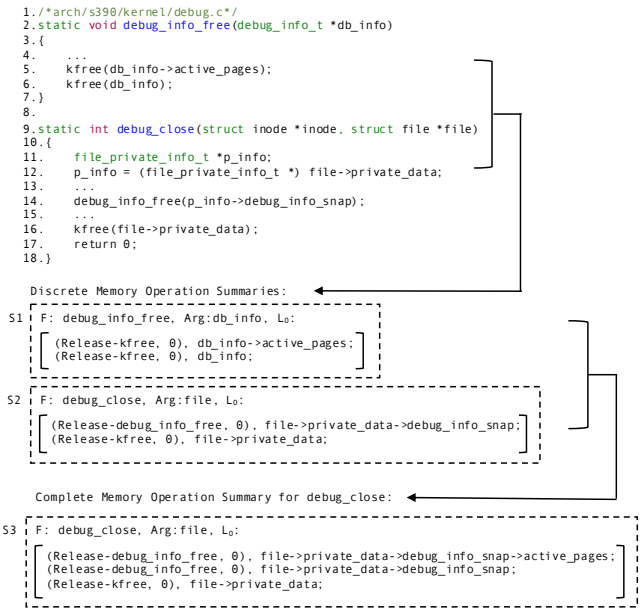

Figure 5: Example of demand-driven summarization.

Algorithm 1: Release summary generation

```
1  SummaryGen(List_Release, F);
   Input: List_Release: Release API list;
          F: Function to analyze;
2  CSet_analyzed ← R_S ← ∅ (Init once on the first call);
3  foreach Call_inst in F do
4      if Call_inst ∈ List_Release then
5          C_index ← GetReleaseIndex(Call_inst);
6          RecurAnalysis(F, Call_inst, C_index);
7      end
8  end
9  Function RecurAnalysis(F, Call_inst, C_index):
10     if Call_inst ∈ CSet_analyzed then
11         return;
12     end
13     CSet_analyzed ← {Call_inst} ∪ CSet_analyzed ;
14     M_O ← Call_inst.GET_OPERAND(C_index) ;
15     if M_O is derived from arguments of F then
16         F_argidx ← GetIndex(F, M_O) ;
17         V_arg ← F.GET_ARG_OPERAND(F_argidx) ;
18         S_R ← GetRelationString(V_arg, M_O) ;
19         Sum ← BuildSum(F, F_argidx, Call_inst, C_index, S_R) ;
20         R_S ← {Sum} ∪ R_S ;
21         foreach Caller_inst of F do
22             F_caller ← GetHostFunc(Caller_inst) ;
23             RecurAnalysis(F_caller, Caller_inst, F_argidx);
24         end
25     end
```

O_{type} further includes the release API name and the index corresponding to the resource parameter. The term O_{value} describes the memory object involved, represented as a string that articulates the structured relationship between the memory object and the function's arguments.

Figure 5 presents an example of memory operation summarization. In this example, `IMMI` produces two distinct summaries of memory release operations for the functions `debug_info_free` and `debug_close`, denoted as S1 and S2, respectively. Within `debug_info_free`, a pair of release invocations (`kfree`) are recorded in S1. Given that `kfree` accepts a singular parameter, the index number in L_O is consequently noted as 0. This function is subsequently invoked by `debug_close` at line 14, with its memory release operations encapsulated in S2. Both S1 and S2 accurately capture the structure-aware relationships between the released memories and the function arguments.

Summary generation. `IMMI` generates release summaries from a set of official kernel release APIs, as detailed in Table 6. The goal of `IMMI` in this process is to identify all function calls that directly or indirectly release memory objects (i.e., release wrappers). Algorithm 1 outlines the process for creating these summaries. Specifically, `IMMI` examines each function within the target system and employs `SummaryGen` to produce individual release summaries. This component accepts two inputs: a predefined list of general release APIs ($List_{Release}$) and the function under analysis (F). The resulting summaries are stored in R_S. The procedure begins by scanning the call instructions within F (line 3). Upon identifying a release call, `IMMI` performs a recursive analysis using `RecurAnalysis` (line 6). During this process, `IMMI` extracts the memory object being released from the release API (line 14) and determines if it originates from an argument of F. If so, `IMMI` extracts this argument (lines 16-17) and examines the derivation relationship between the argument and the memory object (i.e., O_{value}). This analysis is conducted via a backward graph traversal on the alias graph of F (line 18). With all pertinent information collected, `IMMI` constructs and records the summary entity for the current memory object (lines 19-20). At this stage, it is inferred that the released memory objects are propagated through the callers of F, prompting a recursive analysis of these callers to assemble comprehensive release summaries (line 21-24).

Memory resetting summaries are generated in a manner akin to release summaries. In this process, `IMMI` extracts memory objects from release APIs and conducts a further analysis of store instructions that reset these memory objects. To enhance efficiency, `IMMI` maintains a record of the analyzed instructions, thereby avoiding redundant analysis (line 10-13).

Reconstructing complete summary on-demand. The memory operation summaries previously generated are distinct but not exhaustive, failing to encapsulate all memory operations within a given function. For instance, summary S2 in Figure 5 merely indicates that the callee `debug_info_free` has released some memories, lacking comprehensive details. To derive an all-encompassing summary for `debug_close`, it is imperative to amalgamate S2 with S1, which delineates the

specific memory operations performed by `debug_info_free`. When querying the memory operation summaries of a function, `IMMI` first extracts the function's direct summary (e.g., S2 for `debug_close`). Subsequently, `IMMI` examines the memory operation list (L_O) to determine if any listed functions possess additional summaries. If affirmative, `IMMI` recursively aggregates these summaries to formulate a complete summary, concurrently updating the O_{value} throughout the process. For example, by combining the `db_info->ccactive_pages` in S1 and `file->private_data->debug_info_snap` in S2, `IMMI` could precisely reconstruct the actual release memory object is `file->private_data->debug_info_snap->active_pages`. The fully integrated release summary for `debug_close` is depicted as S3 in Figure 5.

4.3 Object-based Code Slicing

The goal of code slicing is to refine the analysis scope, thereby enhancing efficiency and scalability. To this end, `IMMI` initially identifies the error handling paths for each function, subsequently extracting the relevant memory objects contained therein. Ultimately, `IMMI` confines the analysis to those error handling paths that are reachable to the identified memory objects.

Identifying error handling paths. The MM intention analysis in this paper primarily concentrates on error handling mechanisms, necessitating the accurate identification of error handling paths. Current methodologies for this task can be broadly classified into two predominant approaches: the first employs backward data-flow analysis, commencing from return instructions that yield error codes [15, 20]; while the second approach engages in forward data-flow analysis, initiating from function calls that could potentially propagate errors [34]. Our assessment reveals that the former approach often overlooks numerous error handling paths, particularly those associated with function calls, whereas the latter may erroneously classify non-error handling paths as such. To address these shortcomings, `IMMI` synergizes these two strategies, integrating their strengths to enhance the precision and coverage of error path detection. Specifically, `IMMI` initially applies the backward analysis to enumerate error paths within each function and to annotate call instructions that affect branching conditions. It then uses forward analysis to follow these annotated call instructions, capturing the paths that address their failures and excluding irrelevant paths. This integrated approach ensures a more robust and accurate identification of error handling paths.

Extracting memory objects. OS kernels conventionally manage memory resources through established MM application programming interfaces (APIs) [1]. Within this framework, memory objects are typically denoted by pointers. Similar to prior methodologies, our approach extracts memory objects by capturing the pointers returned from the standard kernel memory allocation APIs, as detailed in Table 6 in the Appendix. Additionally, it is commonplace for kernel developers to employ these APIs as foundational elements for crafting bespoke MM allocation functions. To accommodate such practices, our analysis extends to tracking the propagation of memory objects allocated by standard MM APIs to subsequent callers. If these objects are passed forward as return values, we incorporate them into our analysis targets. This comprehensive tracking ensures the integrity and completeness of our memory object analysis scope, thereby enhancing the reliability of our subsequent analysis.

Code slicing. In the context of MM inconsistent analysis, which often spans across multiple functions, it is imperative that the memory objects of interest represent shared resources. Upon obtaining a memory object, `IMMI` tracks its subsequent usage to pinpoint the instruction that permits access to the object for other functions (e.g., a store instruction that commits it to a function's argument). `IMMI` then scrutinizes the collected error handling paths and eliminates those that are not reachable by this instruction (e.g., error paths preceding the memory allocation). This elimination is conducted through a Control Flow Graph (CFG) reachability analysis.

Additionally, `IMMI` filters out error paths that reset the memory object subsequent to its release, by examining the memory operation summaries collected in §4.2. This step is essential as it excludes instances of *safe inconsistent MM*. For example, setting a memory object to NULL post-deallocation renders subsequent deallocations by the callers inconsequential and bug-free, as freeing a NULL pointer is generally benign in kernel contexts. The code segments that remain after this filtration process constitute the slices that lay the groundwork for the subsequent inference of MM intentions.

4.4 LLM-assisted MM Intention Inference

Callee MM intention inference. `IMMI` conducts an object-oriented analysis to infer the memory management (MM) intentions. Utilizing the analysis scope (i.e., the code slices derived in §4.3), alongside summaries of memory operations, `IMMI` deduces the MM intentions for the object within the function responsible for its allocation (denoted as F_{obj} for brevity). These intentions are categorized into three distinct types:

- **Callee-based management.** In scenarios where all error paths within the code slices ensure the release of the memory object, the callers of F_{obj} are not required to manage the memory object upon failure of F_{obj}.
- **Caller-based management.** Conversely, if none of the error paths within the code slices release the memory object, it is expected that the callers of F_{obj} will handle the cleanup of the memory objects in the event of F_{obj}'s failure.
- **Undetermined management.** This category arises when some error paths within the code slices release the memory

```
1  /* drivers/dax/bus.c */
2  static int devm_register_dax_mapping(struct dev_dax *dev_dax, ...)
3  {
4      ...
5      mapping = kzalloc(sizeof(*mapping), GFP_KERNEL);
6      if (!mapping)
7          return -ENOMEM;
8      ...
9      mapping->id = ida_alloc(&dev_dax->ida, GFP_KERNEL);
10     if (mapping->id < 0) {
11         kfree(mapping);
12         return -ENOMEM;
13     }
14     ...
15     rc = device_add(dev);
16     if (rc) {
17         put_device(dev); //Reference counting
18         return rc;
19     }
20
21     rc = devm_add_action_or_reset(..., unregister_dax_mapping,
22             dev); //Indirect release call
23     if (rc)
24         return rc;
25     return 0;
26 }
```

Figure 6: Example of implicit memory management mechanisms in the Linux kernel.

```
1  /* net/netfilter/x_table */
2  static int xt_jumpstack_alloc(struct xt_table_info *i)
3  {
4      ...
5      if (size > PAGE_SIZE)
6          i->jumpstack = kvzalloc(size, GFP_KERNEL);
7      else
8          i->jumpstack = kzalloc(size, GFP_KERNEL);
9      if (i->jumpstack == NULL)
10         return -ENOMEM;
11     ...
12     i->jumpstack[cpu] = kvmalloc_node(size, GFP_KERNEL,
13             cpu_to_node(cpu));
14     if (i->jumpstack[cpu] == NULL)
15         /*
16         * Freeing will be done later on by the callers. The
17         * chain is: xt_replace_table -> __do_replace ->
18         * do_replace -> xt_free_table_info.
19         */
20         return -ENOMEM;
21
22     return 0;
23 }
```

Figure 7: Example of valuable code comments in the Linux kernel.

object, while others do not. Such inconsistency typically indicates a bug. If the caller attempts to release the memory again, a double-free error may occur. Conversely, inaction by the caller can lead to a memory leak along certain paths. Consequently, `IMMI` reports this scenario as exhibiting intra-inconsistent MM intentions and ceases further analysis.

LLM integration. The complexity of kernel memory management mechanisms poses additional challenges for accurately deducing MM operations. Figure 6 shows such an example in the Linux kernel. A heap memory allocation for the variable `mapping` occurs at line 5, with a corresponding deallocation on the error path at line 11. However, the error paths returning at lines 18 and 24 lack explicit deallocation for this resource. Specifically, the error path at line 18 employs reference counting to manage the resource, resulting in an automatic deallocation when `put_device` is invoked. Similarly, the function `devm_add_action_or_reset` at line 21 registers a callback function, `unregister_dax_mapping`, for release (an indirect call), which is triggered upon its failure. Traditional analysis techniques often fail to account for these implicit resource releases, leading to a substantial number of false positives. We also observed that code comments often encapsulate critical insights regarding memory management, as exemplified in Figure 7. Specifically, the annotations spanning lines 15-18 suggest that the responsibility for deallocating the memory allocated to `i->jumpstack` falls to the calling function in the event of a failure signaled at line 20. Discerning these memory release patterns poses a significant challenge for purely static analysis methodologies due to the necessity of tracing complex call hierarchies.

Recent advancements in large language models (LLMs) have demonstrated their proficiency in various security tasks [5, 13, 24, 25, 27]. To tackle the challenges previously mentioned, we utilize the capabilities of LLMs to scrutinize the implicit MM strategies and extract insightful code annotations within the target systems. For each memory object, should `IMMI` detect any sliced path from prior static analysis that fails to deallocate the object, it crafts a tailored LLM prompt to investigate the possibility of an implicit memory release pertaining to that object. To fully leverage the LLM's potential, we decompose the analysis into three distinct phases: targeting the memory object, discerning error handling paths, and examining the relevant code to determine whether the memory object is governed by the calling functions or is subject to automatic deallocation. `IMMI` incorporates the function's source code, where the memory object is instantiated, into the prompt input. The LLM is then instructed to encapsulate its findings in a separate line, from which `IMMI` extracts the LLM's conclusions to augment subsequent analyses. Should the LLM uncover such information, `IMMI` will update its analysis results for the object accordingly. We provide a prompt message example for the code in Figure 6 in Table 5 in the Appendix, where the LLM successfully identifies the implicit memory release mechanisms through reference counting and an indirect call.

4.5 Inconsistency-based Bug Detection

Caller MM intention inference. In the preceding phase, a shared memory object was classified according to its management strategy: either callee-based or caller-based. `IMMI` extends this analysis by examining the callers to deduce the MM intentions for this object. Specifically, `IMMI` begins by identifying the error-handling path associated with the caller instruction and ascertains whether this path deallocates the memory object. If deallocation occurs, the caller function

is designated as employing caller-based management for the object. Conversely, if no release is detected, IMMI proceeds to aggregate all error handling paths that follow the successful execution of the caller instruction. Should any of these paths involve the release of the memory object, the caller function is then categorized under callee-based management. In the absence of these conditions, IMMI conducts a deeper investigation to determine whether the memory object is propagated further along the call chain. When such transfer is detected, IMMI recursively applies the aforementioned analysis until the MM intentions for the memory object can be conclusively identified.

Memory bug detection. In the process of inferring the caller's MM intentions, should there be inconsistency between the caller's and callee's MM intentions for a given memory object, IMMI halts further analysis and flags it as a memory bug. Specifically, when the callee operates under caller-based management, while the caller adheres to a callee-based strategy, IMMI identifies this as a potential memory leak. This occurs because neither party assumes responsibility for deallocating memory upon a particular failure scenario. Conversely, when both the caller and callee attempt to manage the same memory object, IMMI recognizes this as a double-free bug.

We use the memory leak bug in Figure 3 as an example to illustrate the workflow of IMMI. During the callee MM intention inference phase, IMMI discerns that the memory object `qdev->lrg_buf` is allocated at line 5. However, IMMI observes that subsequent error handling paths fail to deallocate it. Consequently, IMMI designates the function `ql_alloc_buffer_queues` as adhering to caller-based management for this object. Upon inspecting the caller function `ql_alloc_mem_resources`, IMMI evaluates its MM intention by analyzing the error paths following the failure of `ql_alloc_buffer_queues` (lines 26-27 and 39-43), noting the absence of deallocation for the memory object. Further analysis of the error paths after `ql_alloc_buffer_queues` has successfully executed reveals a jump to `err_small_buffers`, where the memory is appropriately released at line 38. IMMI thus determines that `ql_alloc_mem_resources` employs callee-based management for the object. This finding is in direct conflict with the handling method of the callee `ql_alloc_buffer_queues`. As a result, IMMI reports a memory leak bug due to this inconsistency.

5 IMMI Implementation

We have implemented IMMI based on LLVM with 5.3K lines of C++ code. We choose ChatGPT-4 (accessed through ChatGPT-4-1106-preview) as our target LLM and implement the LLM query with 200 lines of python code. We record the LLM query results and bug reports in a local MySQL database to facilitate further bug verification.

Pre-processing. To mitigate the risk of path explosion, we unroll loops by transforming the control flow of loop constructs into conditional (`if`) statements. This technique is a common practice in static analysis, as evidenced by its application in various works [20, 32, 33, 43]. We employed multi-layer type analysis [19] for analyzing indirect calls. Nevertheless, our findings indicated that numerous indirect calls involving nested struct types were inaccurately resolved. To address this, we introduced an additional data-flow analysis specifically tailored for these indirect calls to extract more precise nested type information. This refinement led to a reduction of approximately 23% in superflous indirect call targets.

Code exploration settings. In the current implementation, we confine the inter-procedural MM intention analysis for each shared memory object to a maximum of four procedural transitions. This constraint is informed by empirical evidence suggesting that extending the call-chain analysis beyond this threshold yields a negligible increase in bug detection, while incurring a significant computational overhead. Moreover, we have instituted an early termination protocol: if a memory object is encapsulated within a structure, the analysis is promptly concluded upon the deallocation of the enclosing structure. In this case, IMMI deems the memory object as irretrievable and immediately flags it as a potential memory leak.

LLM settings. The outputs of LLMs exhibit a form of instability commonly referred to as *hallucination* [40, 45]. To mitigate this issue, we query each prompt four times, considering only those responses that maintain consistent conclusions in at least three instances as valid. To optimize analysis time and reduce costs, we deploy LLMs exclusively for analyzing the intentions of memory management in cases where IMMI has identified inconsistencies during the static analysis phase. We have also observed that constraining LLM outputs significantly affects its efficacy in our code analysis task. Consequently, we permit the LLM to initially present intermediate analysis results, subsequently synthesizing these into a definitive outcome.

Post-processing. IMMI sometimes produces multiple bug reports that, despite originating from the same root cause, exhibit distinct call chains. This situation often arises when a callee function responsible for managing a shared memory object is invoked by multiple callers. IMMI identifies and reports a bug for each unique caller that demonstrates inconsistent memory management intentions relative to the callee. In our analysis, we have observed that these instances are generally resolvable with a single patch and, therefore, should be considered as a single bug. Consequently, for clarity and accuracy in our statistical analysis, IMMI only retains a single report from such groups of related bug reports.

6 Evaluation

We conducted an evaluation of IMMI on a Linux server running Ubuntu 20.04.1, equipped with 126GB of RAM and an Intel

Table 1: Scalability of IMMI.

Bitcode Loading	ICall Analysis	Summarization	Bug Analysis	LLM I/O	Total
1m 11s	3m 58s	2m 15s	1m 59s	25m 49s	35m 12s

Table 2: Bug detection statistics of IMMI on the Linux kernel. The SA indicates the number of bugs reported by pure static analysis, while the SA+LLM specifies the number identified when static analysis is combined with the large language model (GPT-4).

Category	Reported Bugs		Real Bugs	Precision
	SA	SA+LLM		
Intro-inconsistency	63	52	35	67.3%
Inter-inconsistency	95	71	45	63.4%
Total	158	123	80	65.0%

Xeon Silver 4316 CPU at 2.30GHz. We evaluated the bug detection efficiency of IMMI against the Linux kernel 5.18. To ensure comprehensive module coverage, we compiled the kernel using the `allyesconfig`, resulting in a total of 21,438 LLVM bitcode files.

6.1 Analysis Performance

IMMI completed the analysis of the entire Linux kernel within 35 minutes, as shown in Table 1. Specifically, the static analysis component of IMMI required only approximately 9 minutes, with the remaining time predominantly consumed by I/O latency during queries to the LLM. These results underscore the efficiency and scalability of IMMI in bug detection. IMMI reported a total of 123 unique bugs.

6.2 Bug Findings

We manually analyzed all bug reports generated by IMMI and finally confirmed 80 new bugs, including 57 memory leak bugs, 16 double-free bugs, 6 UAF bugs, and 1 null-pointer-dereference bug. We have reported all the bugs to the Linux community. Until the submission of this paper, 72 bugs have been confirmed or fixed. The detailed bug list is available in the Appendix.

An intriguing observation from our bug analysis is that the majority of inconsistent memory management intentions can be deduced through a relatively constrained scope of analysis. Specifically, among all identified bugs, 35 (43.8%) were detected without necessitating inter-procedural analysis, as highlighted by the unresolved handling cases in §4.4. For the remaining bugs uncovered during inter-procedural analysis, 22 (48.9%) were identified when shared memory objects were propagated across a single procedural boundary. The memory leak bug exemplified in Figure 3 typifies such a scenario. Furthermore, 14 (31.1%) of the bugs were associated with two procedural boundaries, while 9 (20.0%) involved three or more. These findings underscore the efficacy of employing inconsistent MM intention analysis as a strategy for memory bug detection.

6.3 False Positives

The false discovery rate of IMMI stands at 35%, which is actually promising for a static bug analysis tool, outperforming several comparable tools in detecting kernel memory bugs [9, 15, 34]. Leveraging capabilities of LLM for code and comment interpretation, IMMI effectively mitigates false positives arising from complex kernel memory management mechanisms and developer tactics. Upon examining the false positives, we have identified the primary contributing factors as follows:

Infeasible paths. Kernel developers frequently impose constraints to govern the executability of certain program paths. For instance, developers may employ temporary variables to determine the executability of object release sequences. While memory resetting summaries can mitigate some instances of infeasibility, they are not universally effective. These unaddressed cases account for 44% of the false positives in our analysis. To resolve this challenge, IMMI needs to integrate with symbolic execution to discern and exclude infeasible paths. We defer the implementation of this enhancement to future work.

Imprecise error handling path analysis. Error handling mechanisms within operating system kernels are inherently intricate, frequently encompassing numerous layers of verification and a multitude of conditional statements. Despite the incorporation of both forward and backward analyses into our methodology, certain paths elude the scope of our model. This limitation leads to the occasional omission of error handling paths in our identification process, which contributes 26% of the false positives.

Imprecise alias analysis. Despite IMMI's implementation of a field-sensitive alias analysis, the approach falls short in accurately distinguishing between individual array elements when they are accessed via indeterminate indices within loops. Consequently, IMMI erroneously identifies distinct array elements as aliases, contributing to aliasing imprecision. This particular limitation is responsible for approximately 12% of the false positives reported by IMMI.

Other causes. Several less common factors also contribute to the false positives. These encompass limitations in the handling of specialized program logics that exceed the capabilities of our static analysis framework, even when augmented with large language models. Additionally, the complexity of kernel macros and deficiencies in the implementation of the static analysis component of IMMI are also responsible for false positives. Collectively, these issues account for 18% of the false positives observed.

6.4 False Negatives

Since there is no benchmark for false negative analysis, we have devised a test set by deliberately injecting MM intention inconsistencies into the Linux kernel. Specifically, we randomly selected 20 shared memory objects initially managed with caller-based management and altered them to adopt callee-based management. Conversely, we selected additional 20 memory objects that were originally callee-based and transformed their management to caller-based. This intentional perturbation resulted in the introduction of 20 double-free bugs and 20 memory leak bugs. IMMI successfully identified all 20 double-free bugs and 17 of the memory leaks. The three undetected memory leaks can be attributed to instances where the call chains surpassed our predefined threshold, thus evading our analysis.

It is important to note that the efficacy of IMMI's bug detection mechanism is contingent upon the manifestation of inconsistent MM intentions across different functions pertaining to the same memory object. In scenarios where bugs are confined to local resources within a single function, the detection capabilities of IMMI would diminish.

6.5 Performance of LLM

Overall performance. In the process of detecting bugs, IMMI constructed 158 unique query tasks aimed at inferring MM intentions through LLM. Each query was executed four times, consistent with the methodology outlined in §5. Out of these queries, IMMI garnered 128 (81.0%) valid responses. The average I/O latency for each LLM prompt was approximately 2 seconds. Throughout our analysis, the LLM demonstrated commendable stability in processing the tasks presented, considering the complexity of the analysis.

To evaluate the contribution of the LLM to the bug detection capabilities of IMMI, we deactivated the LLM component and reassessed the outcomes using only static analysis. Subsequently, the number of bug reports surged to 158. Upon manual inspection of these reports, we identified three additional true bugs and 32 extra false positives in comparison to the previous results. These findings suggest that the LLM is responsible for approximately 4% of false negatives in bug detection, yet it effectively reduces the false positives produced by IMMI by 43%. Without the LLM's assistance, the overall false discovery rate of IMMI escalates to 47.5%. The majority of these additional false positives stem from the Linux kernel's implicit memory management mechanisms, such as reference counting and release callback functions. Furthermore, LLM mitigates some false positives that arise from inaccurate error path identification. The results underscore the efficacy of IMMI in resolving intricate scenarios that pose challenges to traditional static analysis, demonstrating its value in enhancing system security.

False negatives. We checked the false negatives introduced by LLM, where we found all of them were caused by erroneous assumption in inferring code functionalities. Figure 8 illustrates a representative case, where a memory object qedf->global_queues is instantiated at line 4. All subsequent error handling paths in this function lead to the mem_alloc_failure label, triggering the invocation of qedf_free_global_queues for resource deallocation. Contrary to the LLM's assumption that this release function would adequately dispose of all allocated resources within qedf_alloc_global_queues, it fails to release the aforementioned object. Notably, such function naming conventions could also mislead human developers, potentially leading to a similar misjudgment with the LLM. This is a pitfall for many methods that rely on keyword-based function pairing as well. To address this issue, one approach could involve enriching the LLM queries with additional context, such as the release summaries produced by IMMI, and adopting a recursive query strategy. We identify this as an avenue for future work.

```
1  static int qedf_alloc_global_queues(struct qedf_ctx *qedf)
2  {
3      ...
4      qedf->global_queues = kzalloc(...);
5      if (!qedf->global_queues) {
6          QEDF_ERR(&(qedf->dbg_ctx), "Unable to allocate global "
7                "queues array ptr memory\n");
8          return -ENOMEM;
9      }
10     ...
11     status = qedf_alloc_bdq(qedf);
12     if (status) {
13         QEDF_ERR(&qedf->dbg_ctx, "Unable to allocate bdq.\n");
14         goto mem_alloc_failure;
15     }
16     ...
17     return 0;
18
19 mem_alloc_failure:
20     qedf_free_global_queues(qedf);
21     return status;
22 }
```

Figure 8: Example of false negative introduced by LLM.

Comparison with different LLMs. In this section, we provide a comparison of different LLMs for bug detection within IMMI framework, including GPT-4, GPT-3.5, and Llama-3-70B, as shown in Table 3. Smaller models, such as Llama-3-8B, exhibited difficulties in comprehending the analysis task and produced unpredictable outputs in our preliminary testing, and therefore are not included in the comparison. The prompts for all LLM evaluations are consistent with those used in our previous assessment of GPT-4.

Although GPT-3.5 and Llama-3 demonstrate an understanding of the analysis task, they do not consistently meet every detailed requirement. For instance, Llama-3 often overlooks the criterion that only error paths following the object allocation should be considered. GPT-3.5 struggles to maintain focus on a specific memory object when the error handling paths include memory releases for multiple objects. Moreover, their accuracy significantly diminishes in the context of ex-

Table 3: Comparison with different LLMs. The TP, FP, TN, and FN indicate true positive, false positive, true negative, and false negative, respectively.

LLM	Valid Reports	TP	FP	TN	FN
GPT-4	128 (81.0%)	67	26	32	3
GPT-3.5	140 (88.6%)	41	33	36	30
Llama-3	144 (91.1%)	58	47	19	20

tensive functions and indirect calls. As a result, their utility in aiding bug detection for this paper is suboptimal. Notably, GPT-3.5 and Llama-3 generate nearly as many false negatives as the false positives they reduced. The IMMI framework necessitates an LLM with a more robust reasoning capacity for the static analysis tasks. Models such as GPT-3.5 would necessitate further fine-tuning or prompt engineering to achieve the level of precision necessary for our analysis, a process that may exceed the scope of the current paper.

6.6 Comparison with Existing Tools

This section compares IMMI with four established tools: IPPO [15], HERO [34], MLEE [31], and Goshawk [22]. Each demonstrates proficiency in detecting kernel memory bugs and targets distinct facets of bug detection methodology. We have gathered data on analysis time and precision from the corresponding papers for each tool's evaluation of the Linux kernel and presented a high-level comparison in Table 4. Given that the kernel versions and experimental environments vary across these studies, the metrics provided should be considered indicative of their general capabilities. Overall, IMMI exhibits an effective balance between performance, precision, and the capacity to uncover bugs, with a minimal overlap in bug detection compared to the existing tools, indicating that IMMI could serve as a beneficial complement to them. Below, a detailed analysis of each comparison tool is presented.

Comparison with similarity analysis. IPPO [15] aims to identify bugs stemming from omitted security operations by examining discrepancies within semantically similar path pairs. Among the memory bugs detected by IMMI, IPPO could pinpoint 28 of them, and all of which were memory leak bugs arising during the intra-inconsistency analysis phase of IMMI. Owing to its design, IPPO is confined to the analysis of intra-procedural similar paths, which accounts for its failure to detect the majority of bugs uncovered by IMMI. Furthermore, IPPO is plagued by a low precision of 36.5%, a consequence of intricate code logic and its inability to analyze implicit kernel memory management mechanisms. In contrast, IMMI effectively mitigates these issues by leveraging LLM in understanding code semantics.

Comparison with function pairing. In this subsection, we compare IMMI with HERO [34], the state-of-the-art pairing-based bug detection tool. HERO excels in identifying function pairs through their co-usage patterns and could pinpoint bugs arising from redundant, missing, or disordered paired functions. It is capable of detecting both memory leaks and memory corruption issues. HERO was only able to identify 24 bugs uncovered by IMMI, including 23 memory leak bugs and one double-free bug. The majority of bugs uncovered by IMMI did not involve commonly used function pairs. In fact, aside from standard kernel memory management APIs such as kmalloc and kfree, the function pairs implicated in the bugs detected by IMMI are typically utilized only once or twice throughout the entire kernel. Consequently, the paucity of usage instances prevents HERO from inferring them as function pairs.

Comparison with memory leak detection tool. Given that a substantial proportion of the memory bugs identified by IMMI pertain to memory leaks, we compare IMMI against MLEE [31], which utilizes four weighted rules to detect kernel memory leaks bugs. The combined weight of these rules is 1, and each rule contributes either a score of 1 or 0 to the assessment. If the total weighted score reaches 0.5, MLEE classifies the case as a memory leak. Due to the lack of access to MLEE's source code and the undisclosed rule weights in its research publication, we initially chose to allocate equal weights, assigning 0.25 to each rule on the presumption that MLEE would report a leak upon the concurrent satisfaction of at least two rules. However, we determined that two particular rules are excessively broad[1]. Even bug-free kernel code often meets both criteria, leading to numerous false positives. Given that MLEE's false discovery rate is reported as only 18%, we fine-tuned the weight of these two generic rules to 0.24 and elevated the others to 0.26 to better reflect MLEE's actual performance. Furthermore, our evaluation conservatively presumes MLEE's flawless execution of inter-procedural analysis for memory deallocation and an errorless approach in its alias and liveness analyses.

In our evaluation, MLEE was able to identify 26 memory leak bugs found by IMMI, with 25 exhibiting intra-procedural inconsistencies. The empirical rules demonstrated limited effectiveness in detecting memory leaks that occur across function boundaries, highlighting a weakness in inter-procedural context analysis. Additionally, MLEE recognizes caller-based memory management as a bug pattern, which can lead to misclassification of legitimate memory management strategies as leaks.

Comparison with memory corruption detection tool. We also compare IMMI with a kernel memory corruption detection tool, where we choose Goshawk [22] as our comparison target. It employs a memory operation synopsis technique to streamline the tracking of data flows, thereby facilitating the detection of memory corruptions such as UAF. Upon deploy-

[1] One rule stipulates that if a memory object is allocated within a kernel function, it typically necessitates deallocation along the following error paths of that function. Another rule asserts that when multiple memory objects are allocated within a kernel function, all objects allocated prior to an allocation failure must be deallocated.

Table 4: Comparison with existing tools.

Tool	Methodology	Supported Bug Types	Analysis Time	Precision	Overlap with IMMI
IPPO	Similarity analysis	Memleak, memory corruption, ref-count leak, deadlock, missing check	2h	37%	28
HERO	Function pairing	Memleak, memory corruption, ref-count leak, deadlock	11h	52%	24
MLEE	Rules checking	Memleak	30 min	82%	26
Goshawk	Memory operation synopsis	Memory corruption	7h	63%	0
IMMI	MM intention analysis	Memleak, memory corruption	34 min	65%	80

ing Goshawk on the Linux kernel, we observed that there is no overlap between the bugs detected by Goshawk and those identified by IMMI. A meticulous manual examination of the double-free bugs reported by Goshawk revealed a high incidence of false positives, with all 34 cases being incorrectly flagged. Similar to IPPO, Goshawk's memory operation synopsis fails to account for implicit kernel memory management mechanisms like reference counting, which contributes to this high rate of false positives.

7 Related Work

Memory bug detection. As we evaluated in §6, various previous works try to detect memory bugs in different ways. IPPO [15] detects inconsistencies between similar path-pairs, positing that a memory bug is present if two similar paths exhibit divergent error-handling behaviors. MLEE [31] formulates four rules derived from empirical observations, specifically targeting memory leaks within early-exit paths. HERO [34] identifies function pairs and checks for disordered follower functions including redundant and inadequate ones. FREEWILL [10] detects UAF bugs by identifying inaccuracies in reference counting. These methodologies rely on predefined rules or the extraction of patterns (e.g., function pairs), which, while effective to a degree, may not encompass the full spectrum of memory bug manifestations. Our work aims to bridge this gap by analyzing the high-level memory management intentions.

LLM-assisted program analysis. The advent of Large Language Models (LLMs) has marked a significant milestone in the domain of program analysis, offering novel methodologies for understanding and manipulating code. GPTScan [28] harnesses the capabilities of GPT as a multifaceted code comprehension instrument, adept at identifying critical variables and deconstructing complex bug-finding rules into more tractable properties. ChatRepair [36] employs LLMs to iteratively generate patches based on test failure data, thereby advancing the frontier of automated program repair. FuzzGPT [8] integrates LLMs with a repository of historical bug-inducing programs and anomalous edge cases to cultivate the capacity for autonomously generating fuzzing inputs. LLMs have demonstrated proficiency in assimilating high-level semantic patterns and interfacing effectively with the inputs and outputs of conventional analysis tools. Despite these advancements, the independent deployment of LLMs for bug detection in complex programs remains an elusive goal. Our work explores the potential of LLMs to augment the intricate task of kernel bug detection, suggesting a promising avenue for leveraging these models to enhance the security of critical systems.

8 Conclusion

In this paper, we present IMMI, a novel system designed to identify memory bugs by analyzing the inconsistencies in memory management intentions. IMMI employs advanced techniques such as code slicing and demand-driven memory operation summaries to optimize both efficiency and effectiveness in bug detection. Furthermore, we integrate IMMI with a LLM to augment its code interpretation capabilities. When applied to the Linux kernel, IMMI successfully uncovered 80 new memory bugs, comprising 57 memory leaks and 23 memory corruptions, while maintaining a false discovery rate of only 35%. Our evaluation demonstrates that IMMI is both scalable and proficient at detecting Linux kernel memory bugs.

9 Acknowledgment

We sincerely appreciate our shepherd and all the anonymous reviewers for their insightful comments on our work. This work was partly supported by the Key R&D Program of Zhejiang Province (2022C01086). This work was also partly supported by the Alibaba Cloud Computing. Kangjie Lu was supported in part by the NSF awards CNS2045478, CNS-2106771, CNS-2154989, and CNS-2247434. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of NSF.

References

[1] 2024. Linux memory management APIs. https://www.kernel.org/doc/html/latest/core-api/mm-api.html

[2] Jia-Ju Bai, Yu-Ping Wang, Hu-Qiu Liu, and Shi-Min Hu. 2016. Mining and checking paired functions in device drivers using characteristic fault injection. *Information and Software Technology* 73 (2016), 122–133.

[3] Pan Bian, Bin Liang, Jianjun Huang, Wenchang Shi, Xidong Wang, and Jian Zhang. 2020. SinkFinder: harvesting hundreds of unknown interesting function pairs with just one seed. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1101–1113.

[4] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 133–143.

[5] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* (2023).

[6] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. 1–5.

[7] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2005. Defeating memory corruption attacks via pointer taintedness detection. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE, 378–387.

[8] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. arXiv:2304.02014 [cs.SE]

[9] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *The 2021 Annual Network and Distributed System Security Symposium (NDSS'21)*.

[10] Liang He, Hong Hu, Purui Su, Yan Cai, and Zhenkai Liang. 2022. FREEWILL: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries. In *Proceedings of the 31st USENIX Security Symposium, Security 2022 (Proceedings of the 31st USENIX Security Symposium, Security 2022)*. USENIX Association, 2497–2512. Publisher Copyright: © USENIX Security Symposium, Security 2022.All rights reserved.; 31st USENIX Security Symposium, Security 2022 ; Conference date: 10-08-2022 Through 12-08-2022.

[11] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically detecting error handling bugs using error specifications. In *25th USENIX Security Symposium (USENIX Security 16)*. 345–362.

[12] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, Ji Wang, Xiaodong Liu, and Yunhuai Liu. 2019. Detecting error-handling bugs without error specification input. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 213–225.

[13] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. *arXiv preprint arXiv:2311.16169* (2023).

[14] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–872.

[15] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. 2021. Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1627–1644.

[16] Huqiu Liu, Yuping Wang, Lingbo Jiang, and Shimin Hu. 2014. PF-Miner: A new paired functions mining method for Android kernel in error paths. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 33–42.

[17] Hu-Qiu Liu, Jia-Ju Bai, Yu-Ping Wang, Zhe Bian, and Shi-Min Hu. 2015. Pairminer: mining for paired functions in Kernel extensions. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 93–101.

[18] Hu-Qiu Liu, Jia-Ju Bai, Yu-Ping Wang, and Shi-Min Hu. 2014. BP-Miner: mining paired functions from the binary code of drivers for error handling. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 1. IEEE, 415–422.

[19] Kangjie Lu and Hong Hu. 2019. 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*. 1867–1881.

[20] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[21] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. 2017. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying.. In *NDSS*.

[22] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. 2022. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2096–2113.

[23] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 531–544.

[24] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[25] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[26] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. 2013. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–12.

[27] Tanmay Singla, Dharun Anandayuvaraj, Kelechi G Kalu, Taylor R Schorlemmer, and James C Davis. 2023. An empirical study on using large language models to analyze software supply chain security failures. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. 5–15.

[28] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2023. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. arXiv:2308.03314 [cs.CR]

[29] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 752–762.

[30] Jianqiang Wang, Siqi Ma, Yuanyuan Zhang, Juanru Li, Zheyu Ma, Long Mai, Tiancheng Chen, and Dawu Gu. 2019. NLP-EYE: Detecting Memory Corruptions via Semantic-Aware Memory Operation Function Identification. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 309–321.

[31] Wenwen Wang. 2021. MLEE: Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernels. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 31–45.

[32] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[33] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*.

[34] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and detecting disordered error handling with precise function pairing. In *30th USENIX Security Symposium (USENIX Security 21)*. 2041–2058.

[35] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 781–797.

[36] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv:2304.00385 [cs.SE]

[37] Jidong Xiao, Hai Huang, and Haining Wang. 2015. Kernel data attack is a realistic security threat. In *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 135–154.

[38] Yichen Xie and Alex Aiken. 2005. Context-and path-sensitive memory leak detection. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 115–125.

[39] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 414–425.

[40] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is Inevitable: An Innate Limitation of Large Language Models. *arXiv preprint arXiv:2401.11817* (2024).

[41] Zhenbo Xu, Jian Zhang, and Zhongxing Xu. 2011. Memory leak detection based on memory state transition graph. In *2011 18th Asia-Pacific Software Engineering Conference*. IEEE, 33–40.

[42] Zhenbo Xu, Jian Zhang, and Zhongxing Xu. 2015. Melton: a practical and precise memory leak detection tool for C programs. *Frontiers of Computer Science* 9 (2015), 34–54.

[43] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 363–378.

[44] Dongyang Zhan, Xiangzhan Yu, Hongli Zhang, and Lin Ye. 2022. ErrHunter: Detecting Error-Handling Bugs in the Linux Kernel Through Systematic Static Analysis. *IEEE Transactions on Software Engineering* 49, 2 (2022), 684–698.

[45] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the ai ocean: A survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).

[46] Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. 2022. Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3253–3267.

A Appendix

Table 5: LLM prompt example. The red-highlighted words within the prompt message are generated by the static analysis of `IMMI`.

Description	Prompt messgaes
Prompt message	You are now a program static analysis expert. Your following analysis is based on the following function: [code of devm_register_dax_mapping] We define "error path" in a function as follows: A sequence of basic blocks that finally returns a non-zero number or null pointer. Note that if a call returns error but the path finally does not return a negative number or null pointer, this path is not an error path. A heap memory "mapping" is allocated through "mapping = kzalloc(sizeof(*mapping), GFP_KERNEL);" Please identify all of the LATER error paths after the allocation. Then, please analyze whether all of these paths have freed the heap memory. Pay attention to the implicit kernel memory release operations. Your final conclusion should be a seperate line like: [Conclusion: Answer], "Answer" should only be "yes" or "no".
Reply from LLM	Analyzing the function "devm_register_dax_mapping", we can identify the error paths that occur after the allocation of heap memory for "mapping" with "kzalloc". We need to check if the allocated memory is properly freed on each error path. 1. The first error path is when "ida_alloc" fails (returns a negative number). In this case, the allocated memory for "mapping" is freed with "kfree(mapping)" before returning "-ENOMEM". 2. The second error path is when "device_add" fails (returns a non-zero number). In this case, "put_device(dev)" is called, which will eventually lead to the release of the device and the associated memory. Since "dev" points to "mapping->dev", and "mapping" is embedded within the "struct device", the memory for "mapping" will be freed when the device is put. 3. The third error path is when "devm_add_action_or_reset" fails (returns a non-zero number). In this case, the function returns "rc" directly. However, "devm_add_action_or_reset" is designed to automatically clean up resources by executing the provided action ("unregister_dax_mapping") if the device managed by "devm" is removed or if the driver is unloaded. This means that if "devm_add_action_or_reset" fails, it will not add the cleanup action, but since "device_add" has already succeeded at this point, the device's release function will be called when the device is removed, which should take care of freeing the "mapping" memory. Based on the above analysis, all error paths after the allocation of "mapping" have mechanisms to free the allocated heap memory, either explicitly with "kfree" or implicitly through the device management functions. [Conclusion: yes]

Table 6: Memory management APIs.

API description	API list
Memory allocation	kcalloc, kzalloc, kmalloc malloc, vmalloc, vzalloc kmallo_array, kvmalloc_array kzalloc_node, kmemdup, calloc
Memory deallocation	free, kfree, kvfree vfree, kmem_cache_free

Table 7: List of bugs (1-39) detected by `IMMI` in Linux kernel. The S, C, A, and F in the Status column indicate submitted, confirmed, accepted, and fixed by other developers in the latest version, respectively.

Buggy function	Type	Status
niu_alloc_channels	Memleak	F
kfd_mem_dmamap_userptr	Memleak	F
cpufreq_dbs_governor_init	Memleak	F
aldebaran_tables_init	Memleak	A
smu7_hwmgr_backend_init	Memleak	A
amdgpu_parse_extended_-power_table	Double-free	A
arfs_create_groups	Double-free	A
gssx_dec_option_array	Memleak	A
mlx5e_fs_tt_redirect_any_create	Memleak	F
fs_udp_create_groups	Memleak	A
fs_any_create_groups	Memleak	A
accel_fs_tcp_create_groups	Memleak	F
sumo_parse_power_table	Memleak	A
trinity_parse_power_table	Memleak	A
rvu_npa_register_reporters	UAF	A
rvu_nix_register_reporters	UAF	A
budget_av_attach	Memleak	A
_r8712_init_xmit_priv	Memleak	F
submit_urbs	Memleak	F
load_video_binaries	Null-pointer-dereference	C
cx231xx_init_isoc	Memleak	A
irtoy_tx	Memleak	A
dvb_create_media_entity	Double-free	F
megasas_init_fw	Memleak	S
qla2x00_mem_alloc	Memleak	S
efct_hw_setup_io	Memleak	S
__drm_universal_plane_init	Memleak	F
vmw_gmrid_man_get_node	Memleak	A
lima_heap_alloc	Memleak	A
nv50_wndw_new_	Memleak	S
btrfs_get_dev_args_from_path	Memleak	F
selinux_add_opt	Double-free	F
alloc_wbufs	Memleak	F
btt_freelist_init	Memleak	C
btt_rtt_init	Memleak	C
btt_maplocks_init	Memleak	C
ccp_init_dm_workarea	Memleak	A
fjes_hw_setup	Memleak	A
go7007_load_encoder	Memleak	A

Table 8: List of bugs (40-80) detected by `IMMI` in Linux kernel. The S, C, A, and F in the Status column indicate submitted, confirmed, accepted, and fixed by other developers in the latest version, respectively.

Buggy function	Type	Status
hfs_find_init	Memleak	C
i40e_init_recovery_mode	Memleak	C
iio_device_register_eventset	Memleak	F
iio_device_register_sysfs	Memleak	A
init_credit_return	Memleak	A
intel_gvt_init_vgpu_types	Memleak	F
ip_setup_cork	Memleak	A
lpfc_nvmet_setup_io_context	Memleak	S
open_card_ubr0	Memleak	A
otx2_sq_init	Memleak	A
psm_init_power_state_table	Memleak	F
ql_alloc_buffer_queues	Memleak	A
radeon_vm_init	Memleak	A
rsi_coex_attach	Memleak	F
rtl8188eu_init_recv_priv	Memleak	F
sja1105_setup_devlink_regions	Memleak	F
vimc_sen_add	Memleak	C
vivid_create_instance	Memleak	C
v4l2_m2m_register_entity	Memleak	A
wm_adsp_buffer_populate	Memleak	A
beiscsi_init_wrb_handle	Memleak	A
atl1e_setup_ring_resources	Memleak	A
r8712_init_drv_sw	Memleak	F
_r8712_init_xmit_priv	Memleak	F
lbs_allocate_cmd_buffer	Memleak	A
megasas_alloc_cmdlist_fusion	UAF	S
qed_ilt_shadow_alloc	UAF	A
qla2x00_mem_alloc	Double-free	S
r600_parse_extended_power_table	Double-free	F
kfd_gtt_sa_allocate	Double-free	F
nitrox_mbox_init	Double-free	F
si_parse_power_table	Double-free	A
si_dpm_init	Double-free	F
kv_parse_power_table	UAF	A
efx_probe_filters	Double-free	A
mdp4_plane_init	Double-free	C
dvb_register_device	UAF	A
nand_scan_tail	Double-free	A
uncore_type_init	Double-free	A
base_alloc_rdpq_dma_pool	Double-free	S
bnxt_init_tc	Double-free	A