浙江大學 Zhejiang University | University of Minnesota Driven to Discover® | 北京大學 Peking University

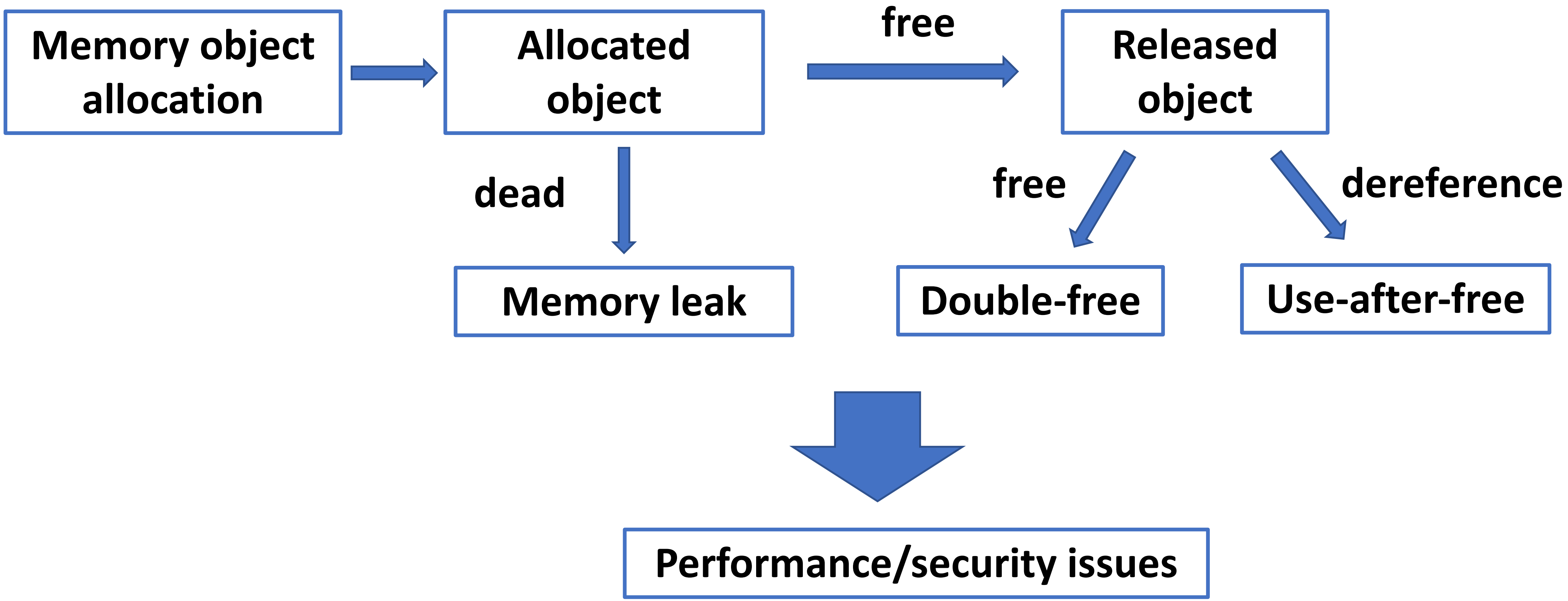
Detecting Kernel Memory Bugs through Inconsistent Memory Management Intention Inferences

Dinghao Liu Zhipeng Lu Shouling Ji Kangjie Lu
Jianhai Chen Zhenguang Liu Dexin Liu Renyi Cai Qinming He

USENIX Security 2024

Background

- Kernel memory bugs



Kernel Memory Bug Detection

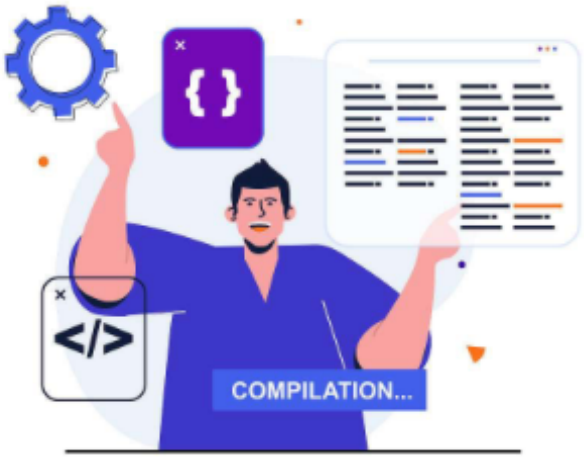
- **Data-flow analysis**
  - Keep tracking the lifecycle of a memory object.

  - Path explosion for kernel analysis

- **Function pairing**
  - The memory management function pairs are intended to operate in concert.

- **Similarity analysis**
  - Memory operations should be consistent within similar paths.

  - Both rely on CORRECT code snippets



Insight

- Memory bugs frequently manifest within error handling paths, where programs attempt to release allocated memory resources.

- A multitude of memory bugs originate from a fundamental issue: the unclarity in ownership and lifecycle management of memory objects when premature release is necessitated.

There exists inconsistent intentions

regarding the management of memory objects.

Memory Management Strategies

➢ Callee-based management

- Functions who allocate heap memories do the cleanup on failure
- The callers need not to take care of the cleanup
- The most widely-used strategy of memory management

```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;
}
```

Memory Management Strategies

➢ Caller-based management

- The caller functions do the cleanup on failure of callees
- The error handling of callee functions would be simple and clear
- This strategy is also popular

```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;
	}
	...
}
```

Inconsistent MM Intentions

Impact analysis

- **Memory corruption**: the callee function adopts callee-based management, while the caller function adopts caller-based management.
- **Memory leak**: the callee function adopts caller-based management, while the caller function adopts callee-based management.

Causes of inconsistent MM intentions

- Complex error handling logic.
- Imperfect code updating.
- Implicit OS MM mechanisms.

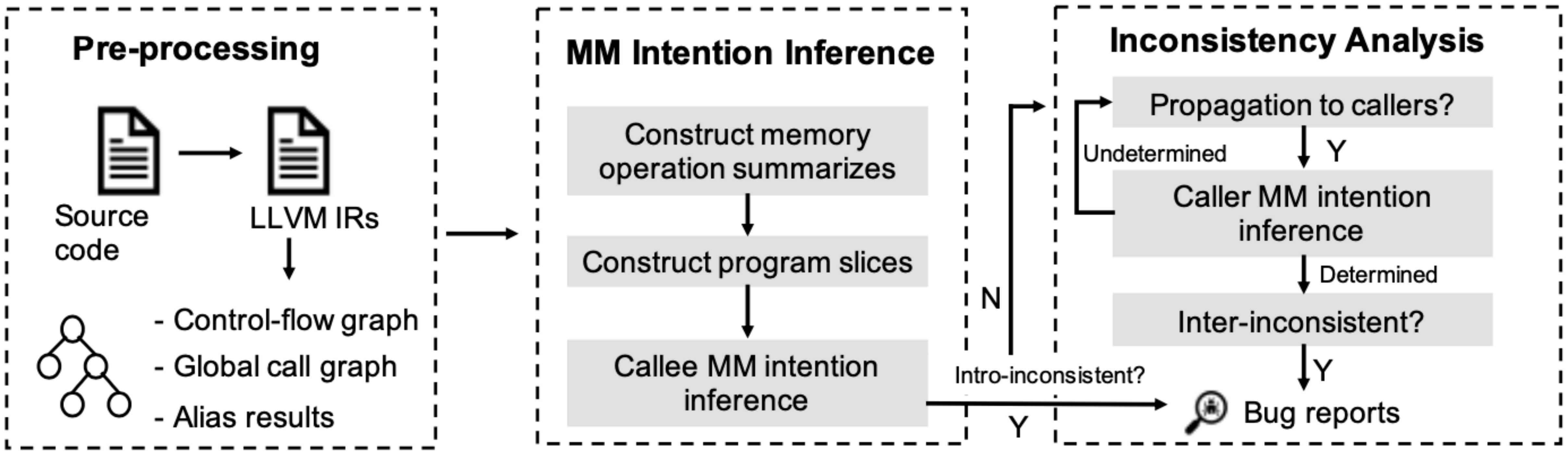
Challenges & Solutions

- **Challenge 1: How to infer MM intentions?**
  - Using static program analysis to infer explicit MM intentions.
  - Using large language model to better understating implicit MM intentions.
- **Challenge 2: How to balance precision and efficiency?**
  - Demand-driven memory operation summarization
  - Object-based code slicing

System Design

Overview

IMMI (Inconsistent Memory Management Intentions)



Memory Operation Summarization

- **Target: Simplify the inter-procedural analysis**

Kernel developers typically implement memory management in a layered fashion

```
debug_close(struct inode *inode, struct file *file)

file_private_info_t *p_info;
p_info = (file_private_info_t *) file->private_data;
…
debug_info_free(p_info->debug_info_snap);
…
kfree(file->private_data);
…
```

```
debug_info_free(debug_info_t *db_info)

…
debug_areas_free(db_info);
kfree(db_info->active_pages);
kfree(db_info);
…
```

```
debug_areas_free(debug_info_t *db_info)

…
kfree(db_info->areas);
…
```

Memory Operation Summarization

- **Techniques: Demand-driven summarization**

First generate function level summaries

Find callers

```
debug_info_free(debug_info_t *db_info)
…
kfree(db_info->active_pages);
kfree(db_info);
…
```

```
F: debug_info_free, Arg:db_info, L0:
[ (Release-kfree, 0), db_info->active_pages;
  (Release-kfree, 0), db_info; ]
```

```
debug_close(struct inode *inode, struct file *file)
file_private_info_t *p_info;
p_info = (file_private_info_t *) file->private_data;
…
debug_info_free(p_info->debug_info_snap);
…
kfree(file->private_data);
…
```

```
F: debug_close, Arg:file, L0:
[ (Release-debug_info_free, 0),
  file->private_data->debug_info_snap;
  (Release-kfree, 0), file->private_data; ]
```

Memory Operation Summarization

- **Techniques: Demand-driven summarization**

  Reconstruct complete summaries on-demand

F: debug_close, Arg:file, L_0:

(Release-debug_info_free, 0), file->private_data->debug_info_snap;
(Release-kfree, 0), file->private_data;

Find a customized release API, check its summary

F: debug_info_free, Arg:db_info, L_0:

(Release-kfree, 0), db_info->active_pages;
(Release-kfree, 0), db_info;

F: debug_close, Arg:file, L_0:

(Release-debug_info_free, 0), file->private_data->debug_info_snap->active_pages;
(Release-debug_info_free, 0), file->private_data->debug_info_snap;
(Release-kfree, 0), file->private_data;

Object-based Code Slicing

- **Target: Confine the analysis scope and enhance efficiency**

  - **Extracting memory object**

    \- Extract memory object through official memory allocation APIs.

  - **Identifying error handling paths**

    \- Combining the forward and backward data-flow analysis.

  - **Code slicing**

    \- Tracking the usages of the memory object.

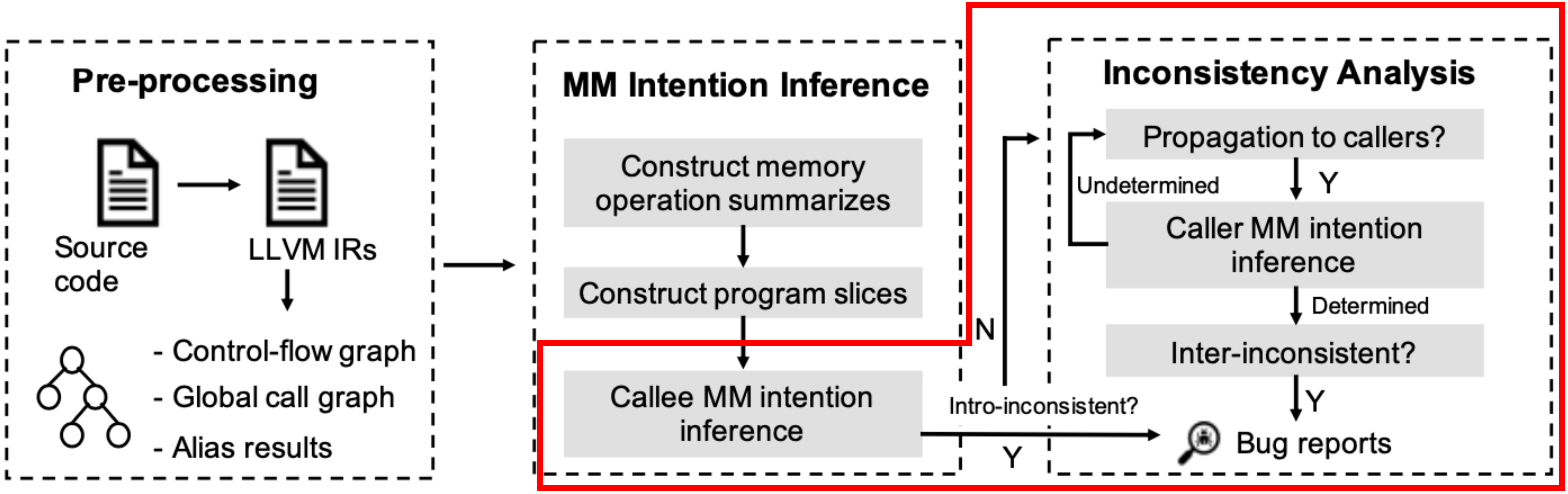
    \- Finding the instruction that permits access to the object for other functions

    \- Eliminating paths that are not reachable by this instruction.

    \- Eliminating paths that reset the memory object after its release.

Overview

IMMI (Inconsistent Memory Management Intentions)



MM Intention Inference

➢ Callee MM intention inference

Upon a memory object is allocated, IMMI analyzes the host function:

- **Callee-based management**

  - All error paths within the code slices ensure the release of the memory objects.

- **Caller-based management**

  - None of the error paths within the code slices release the memory object.

- **Undetermined management**

  - Some error paths release the memory object, while others do not.

MM Intention Inference

➢ Callee MM intention inference

Upon a memory object is allocated, IMMI analyzes the host function:

- **Callee-based management**

  - All error paths within the code slices ensure the release of the memory objects.

- **Caller-based management**

  - None of the error paths within the code slices release the memory object.

- **Undetermined management** → **Intro-inconsistency, stop further analysis**

  - Some error paths release the memory object, while others do not.

MM Intention Inference

➢ Caller MM intention inference

When the memory object is propagated to caller functions, IMMI analyzes the callers:

- **Caller-based management**

  - The memory object is released on failure of the callee.

- **Callee-based management**

  - The memory object is not released upon failure of the callee,
  but is released in the subsequent error paths.

- **Undetermined management**

  - No release operation is detected, track the transfer to further callers.

MM Intention Inference

- **Large language model integration**

  Target 1: Improving the understanding of implicit MM mechanisms.

  Target 2: Improving the understanding of code comments.

Prompt message

- You re now a program static analysis expert. Your following analysis is based on the following function: [code given by IMMI]

- 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``[object given by IMMI]“ is allocated through ”[instruction given by IMMI]“ 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 separate line like: [Conclusion: Answer], "Answer" should only be "yes" or "no".

Case Study

A Memory Leak Bug Found by IMMI

```c
/* drivers/net/ethernet/qlogic/qla3xxx.c */
static int ql_alloc_buffer_queues(struct ql3_adapter *qdev)
{
    ...
    qdev->lrg_buf = kmalloc_array(...);
    if (qdev->lrg_buf == NULL)
        return -ENOMEM;
    ...
    if (qdev->lrg_buf_q_alloc_virt_addr == NULL) {
        netdev_err(qdev->ndev, "lBufQ failed\n");
        return -ENOMEM;
    }
    ...
    return 0;
}

static int ql_alloc_mem_resources(struct ql3_adapter *qdev)
{
    ...
    if (ql_alloc_net_req_rsp_queues(qdev) != 0) {
        netdev_err(qdev->ndev, ...);
        goto err_req_rsp;
    }

    if (ql_alloc_buffer_queues(qdev) != 0) {
        netdev_err(qdev->ndev, ...);
        goto err_buffer_queues;
    }

    if (ql_alloc_small_buffers(qdev) != 0) {
        netdev_err(qdev->ndev, ...);
        goto err_small_buffers;
    }
    ...
    return 0;
    ...
err_small_buffers:
    ql_free_buffer_queues(qdev);
err_buffer_queues:
    ql_free_net_req_rsp_queues(qdev);
err_req_rsp:
    ...
    return -ENOMEM;
}
```

Function ql_alloc_buffer_queues employs caller-based management for *qdev->lrg_buf.*

When ql_alloc_buffer_queues fails, *qdev->lrg_buf* is not freed.

When the following calls fail, *qdev->lrg_buf* is freed through ql_free_buffer_queues.

IMMI determines that this function employs callee-based management.

Evaluation

Evaluation Settings

Environment

- Use a Linux server with 126 GB RAM and an Intel Xeon Silver 4316 CPU
- Use Clang-15 to implement IMMI

Target

- The Linux kernel of v5.18

LLM settings

- Model: GPT-4 (gpt-4-1106-preview)
- Query each prompt 4 times, at least 3 responses should maintain consistent
- Let LLM present intermediate results, subsequently synthesizing them into a definitive outcome
- Act as a post-filter for static analyzer

Evaluation - IMMI

Performance

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

Bug findings

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%

80 new bugs: 57 memleak bugs, 16 double-free bugs, 6 UAF bugs, 1 null-pointer-dereference bug

Evaluation - IMMI

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	35 min	65%	80

IMMI effectively balances performance, precision, and the ability to uncover bugs.

Evaluation - LLM

Performance of LLM (GPT-4)

- Total bug reports: 158 -> 123
- Among the eliminated 35 reports, 32 are valid (false positives of IMMI)
- Overall false discovery rate of IMMI: 47.5% -> 35.0%

Comparison with different LLMs

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

Evaluation - LLM

Effectiveness of LLM (GPT-4)

```c
/* drivers/dax/bus.c */
static int devm_register_dax_mapping(struct dev_dax *dev_dax, ...)
{
    ...
    mapping = kzalloc(sizeof(*mapping), GFP_KERNEL);
    if (!mapping)
        return -ENOMEM;
    ...
    mapping->id = ida_alloc(&dev_dax->ida, GFP_KERNEL);
    if (mapping->id < 0) {
        kfree(mapping);
        return -ENOMEM;
    }
    ...
    rc = device_add(dev);
    if (rc) {
        put_device(dev);
        return rc;
    }

    rc = devm_add_action_or_reset(..., unregister_dax_mapping,
              dev);
    if (rc)
        return rc;
    return 0;
}
```

➢ Reference counting analysis

· \`device_add(dev)\` is called, and if it fails, \`put_device(dev)\` is called, which will decrement the reference count and should trigger the release of the device and its associated memory if the count reaches zero.

➢ Indirect release call analysis

· \`devm_add_action_or_reset\` is called to add a cleanup action (\`unregister_dax_mapping\`) to the device-managed resource list of \`dax_region->dev\`. If this fails, the function returns the error code, and the device-managed resource list will take care of cleaning up the resources.

Evaluation - LLM

False negative of LLM (GPT-4)

```c
static int qedf_alloc_global_queues(struct qedf_ctx *qedf)
{
    ...
    qedf->global_queues = kzalloc(...);
    if (!qedf->global_queues) {
        QEDF_ERR(&(qedf->dbg_ctx), "Unable to allocate global "
               "queues array ptr memory\n");
        return -ENOMEM;
    }
    ...
    status = qedf_alloc_bdq(qedf);
    if (status) {
        QEDF_ERR(&qedf->dbg_ctx, "Unable to allocate bdq.\n");
        goto mem_alloc_failure;
    }
    ...
    return 0;

mem_alloc_failure:
    qedf_free_global_queues(qedf);
    return status;
}
```

- **The function `qedf_free_global_queues` is not defined in the provided code snippet, but based on the naming convention and typical practices in C, we can infer that this function is responsible for freeing the resources allocated for the global queues. Since all error paths after the allocation of `qedf->global_queues` lead to this function call, we can conclude that the function is designed to free the allocated heap memory.**

qedf_free_global_queues() does not actually release ***qedf->global_queues***

Conclusion

- **Inconsistent memory management intentions could lead to many kernel memory bugs like memleaks and memory corruptions.**

- **We presented IMMI to detect kernel memory bugs.**
  - Demand-driven summarization.
  - Object-based code slicing.
  - Combining program analysis and LLM in MM intention inference.

- **We evaluated IMMI on the Linux kernel**
  - Find 80 new memory bugs.
  - IMMI could effectively detect bugs that missed by existing tools

Q&A

Dinghao Liu: dinghao.liu@zju.edu.cn

Zhipeng Lu: alexious@zju.edu.cn