K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploit against Linux Kernel

Zhengchuan Liang
Xiaochen Zou
Chengyu Song
Zhiyun Qian

Memory Error

OS kernels are major targets of attackers

Memory errors

- **Read/write** in **unintended ways**
- Out-of-bound (OOB): r/w using an oob pointer
- Use-after-free (UAF): r/w using a dangling pointer

Exploitation

- OOB: Allocate an obj at the oob location
- UAF: Reallocate an obj

Infoleak

Exploit mitigation techniques in OS kernels

- E.g., KASLR
- Efforts to **circumvent** them

Infoleak

- Disclose mem layout / content
- Achieved by **exploiting** vulnerabilities

Infoleak Approaches

Two broad categories

- Side-channel-based
  - E.g., micro-architectural side-channel
- Memory-error-based
  - By exploiting **memory errors**

Memory-error-based Infoleak

Starting point

- One memory-error (e.g., UAF or OOB).

Goal

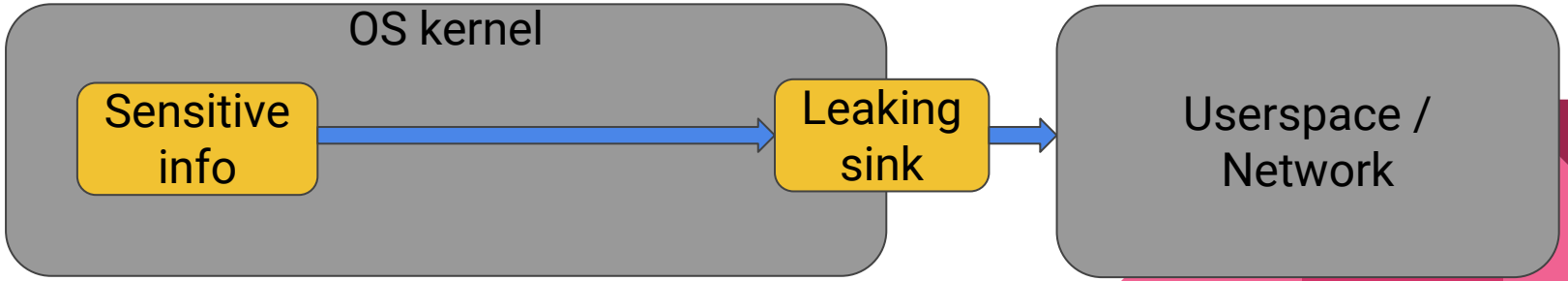
- To leak sensitive info out of the kernel

Memory-error-based Infoleak

Leverage **unintended reads and writes** to create **an infoleak data-flow**

Infoleak data-flow

- Source: sensitive information
- Sink: leaking sink



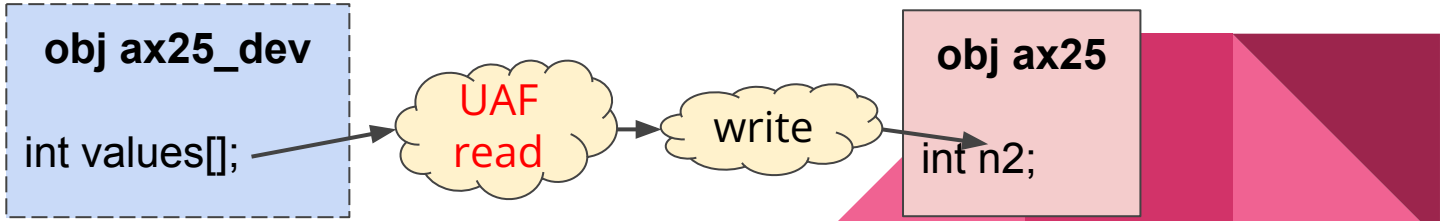
Goal: Assist the automated generation of infoleak exploits given a memory error (with PoC)

Motivating Example

(1) **UAF read error**

ax25_setsockopt()

ax25->n2 = **ax25_dev->values[N2]**;



Motivating Example

(0) Reallocate obj mbus

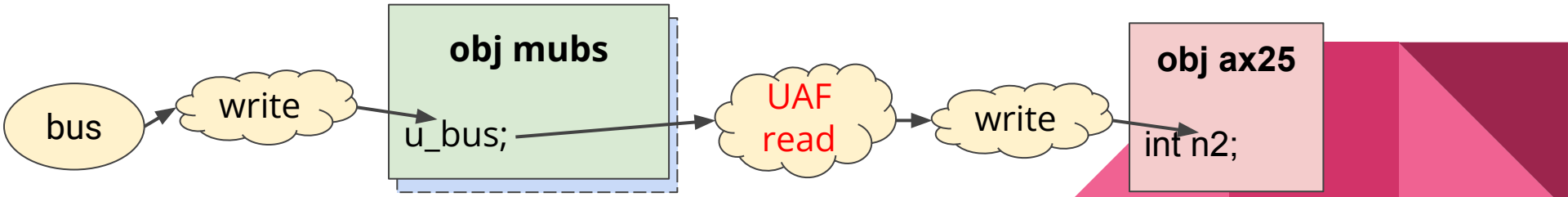
```
mon_bus_init()

mbus->u_bus = bus;
```

(1) **UAF read error**

```
ax25_setsockopt()

ax25->n2 = ax25_dev->values[N2];
```



Motivating Example

(0) Reallocate obj mbus

```
mon_bus_init()

mbus->u_bus = bus;
```

(1) **UAF read error**

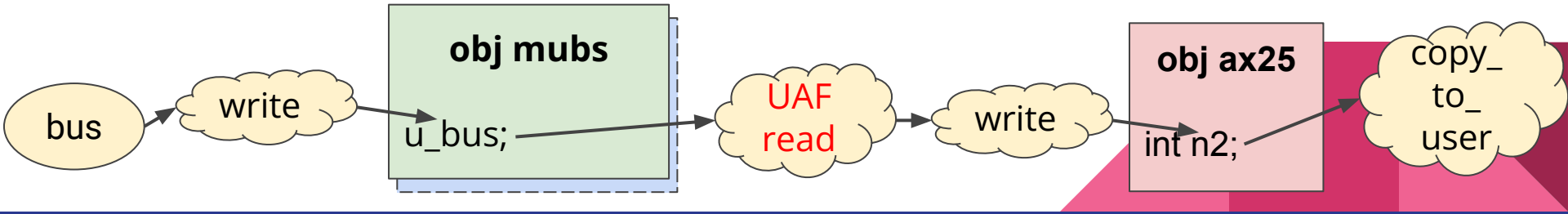
```
ax25_setsockopt()

ax25->n2 = ax25_dev->values[N2];
```

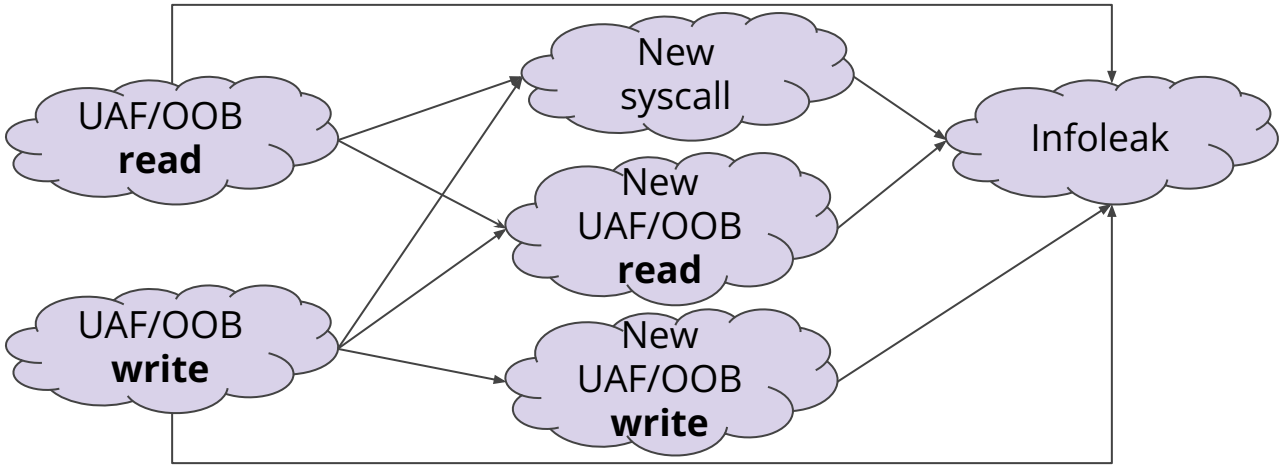
(2) Leak

```
ax25_getsockopt()

val = ax25->n2;
copy_to_user(..., &val, sizeof(int));
```



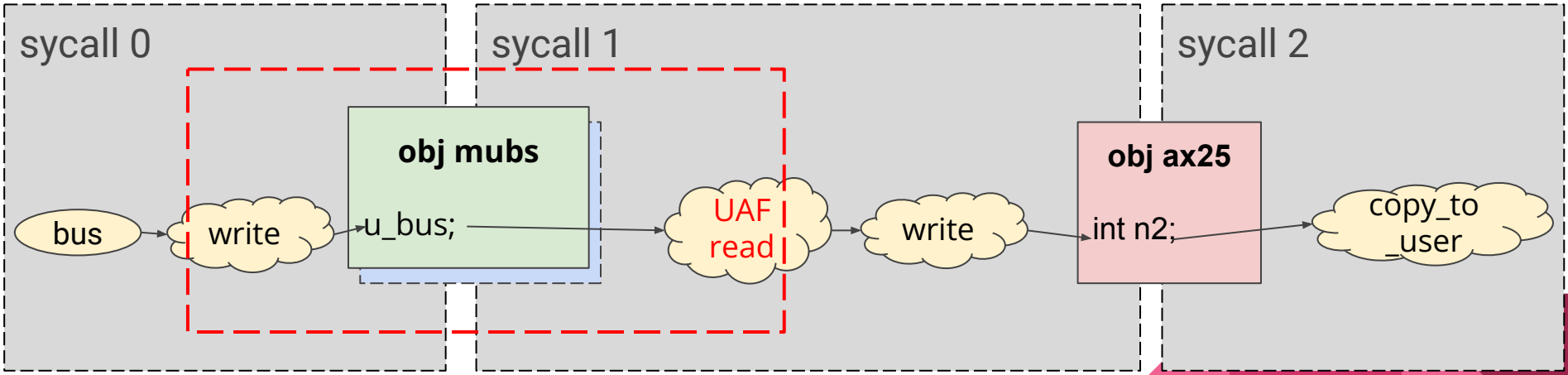
Multiple strategies
A Large Search Space



Technical Challenge 1

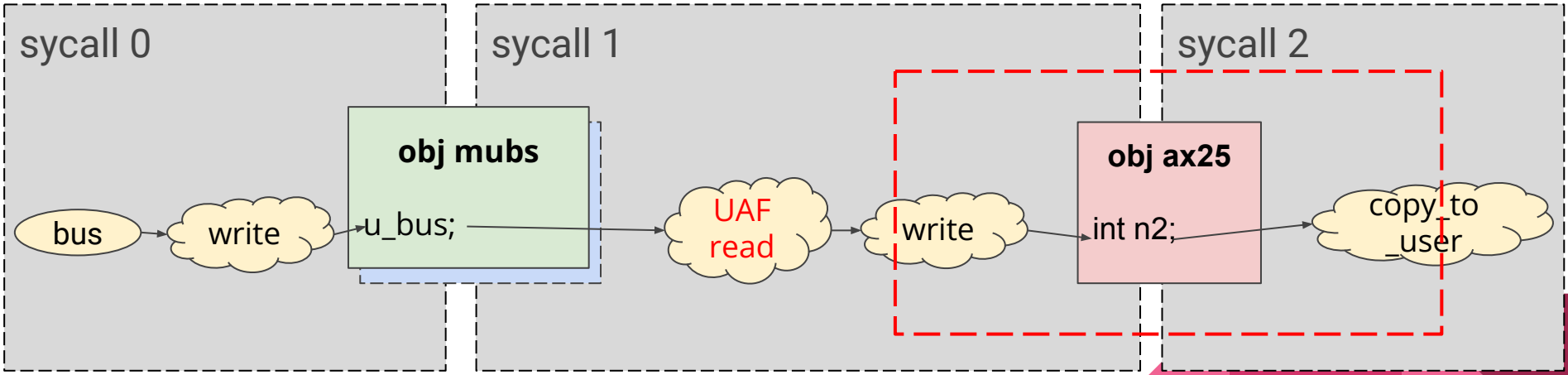
Modeling **unintended** data-flow

- Memory errors: **dereferences of invalid pointers**
- Data-flow **between memory LOAD and STORE operations**



Technical Challenge 2

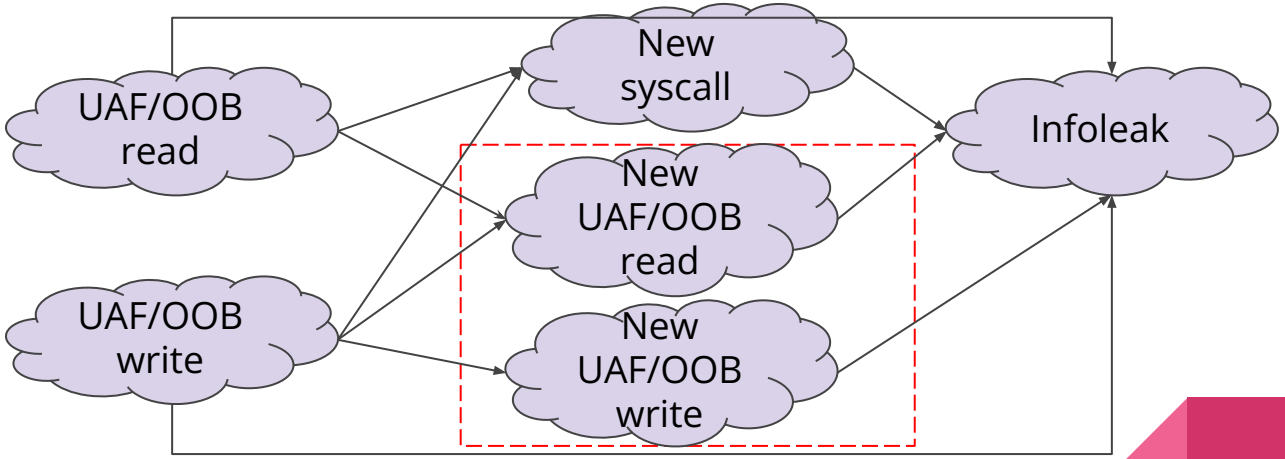
Modeling data-flow **across system calls**



Technical Challenge 3

Modeling **additional** memory errors

- A single memory error may not directly be exploitable.
- Create additional memory errors



Our Work: Graph-based Framework

A **graph-based** data-flow reasoning and search framework.

- Crafting infoleak exploits
- =>
- Searching for data-flow fragments in the **graph**

Large search space: multiple strategies to achieve infoleaks

- Handled through a **unified graph search**

Our Work: Graph-based Framework

Unique **features**

- Handling **intended** and **unintended** dataflow
- Across the **boundary of syscalls**
- Derivation of **intermediate primitives** (i.e., new memory errors)

Large search space: multiple strategies to achieve infoleaks

- Handled through a **unified graph search**

Maximize the chance of generating infoleaks

M-DFG

Nodes

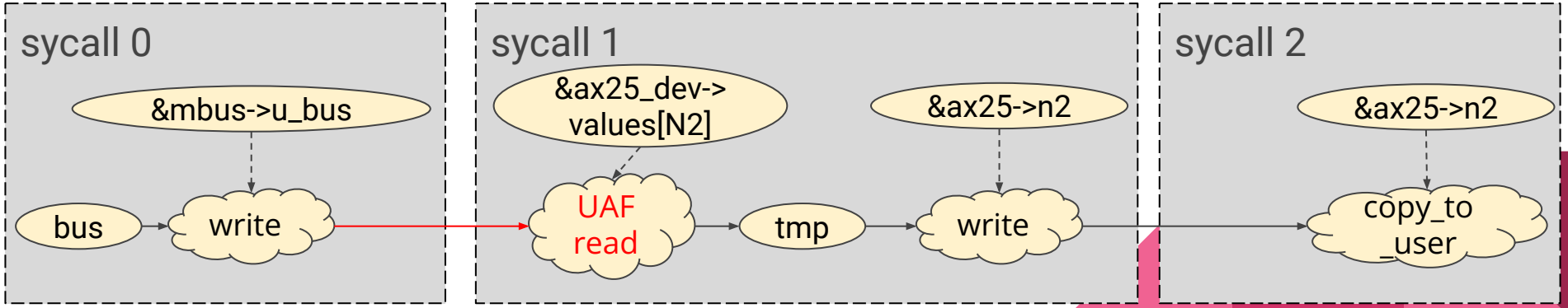
- **Variable nodes**
- **LOAD nodes**
- **STORE nodes**

Edges

- **Data edge**
  - RAW edge
- **Pointer edges**
  - Pointer variable -> LOAD/STORE

Uniqueness

- No obj node
- Pointer edge
- Unintended df



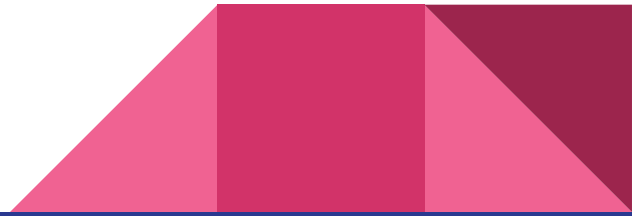
Overview

Problem scope

- Automate infoleak exploit generation
- × Control flow hijacking or end-to-end privilege escalation

Key Insight

- Additionally model **unintended data-flows** introduced by memory errors



Workflow

Input:

kernel code + a memory error (w/ PoC)

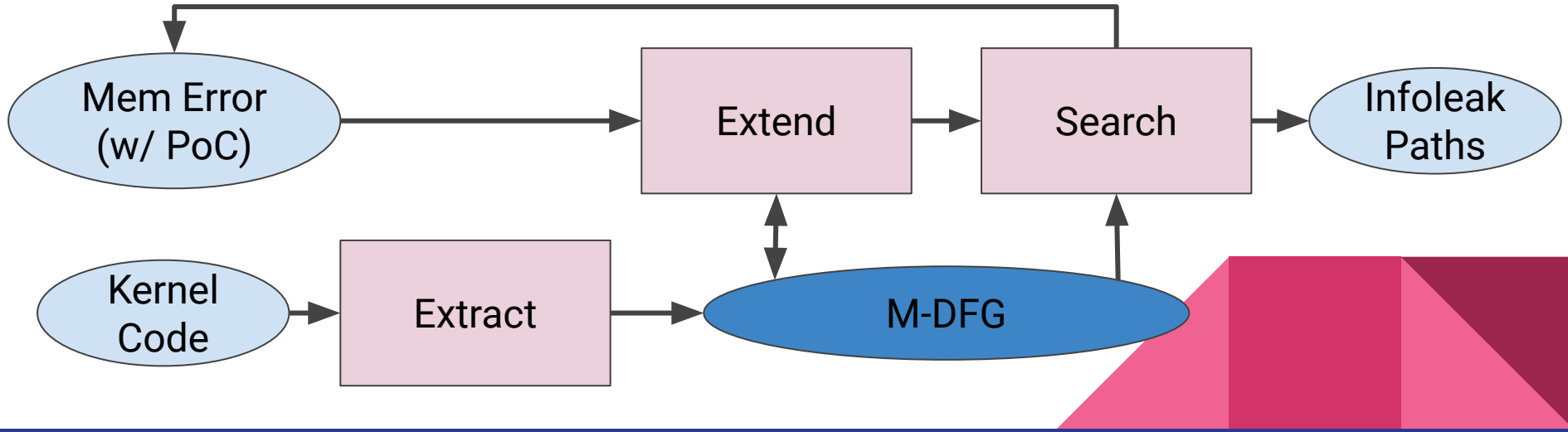
Output:

Infoleak paths

Extract intended M-DFG (Static Analysis)

Extend M-DFG

Search M-DFG (Static Analysis + Dynamic Verification)

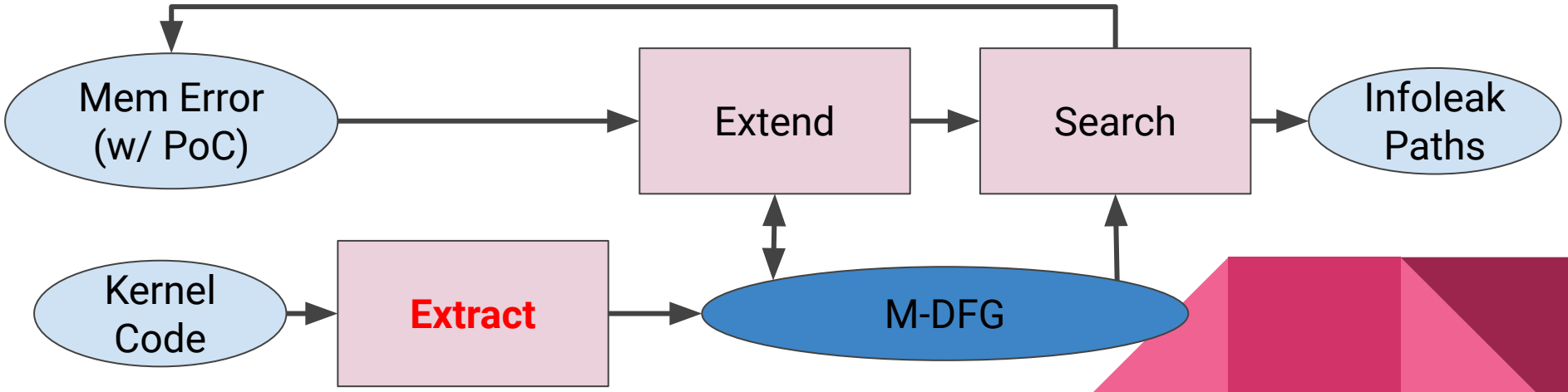


Workflow: Extract Intended M-DFG

Points-to analysis

Graph Construction

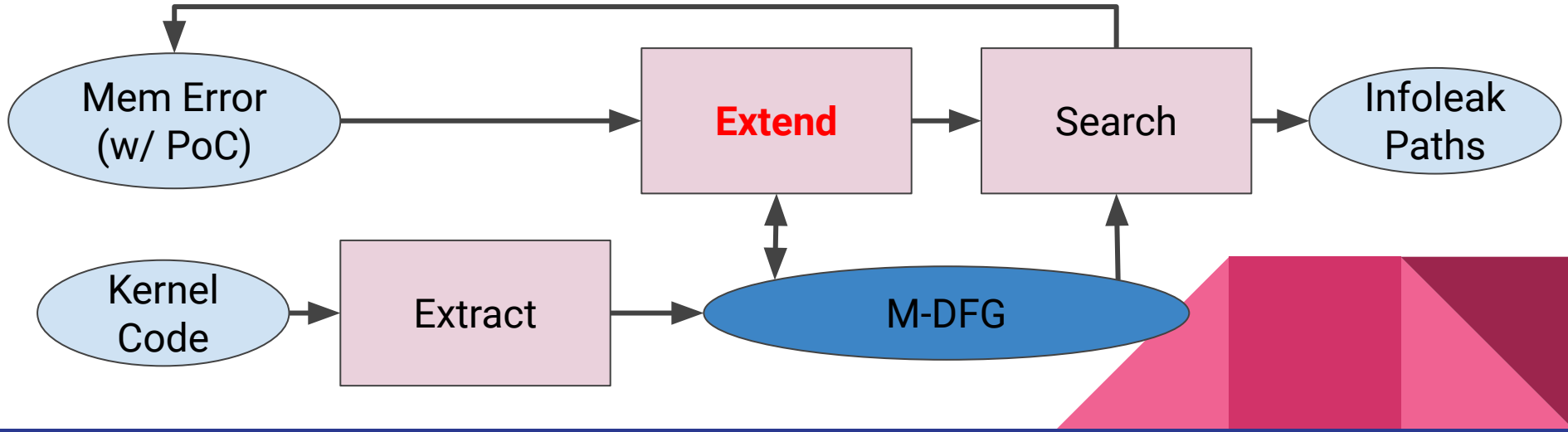
- Create **nodes** and link with **edges**
- Summary-based, Inter-procedural



Workflow: Extend M-DFG

Extend M-DFG with **unintended data-flows**

- Capability of the memory error
  - (1) Slab cache
  - (2) Offset/length



Workflow: Extend M-DFG (cont.)

RAW edge

- STORE -> LOAD
- Models **intended** and **unintended** data-flows

RAW rule to add unintended RAW edge

STORE s: *ptr = val1;

LOAD l: val2 = *ptr;

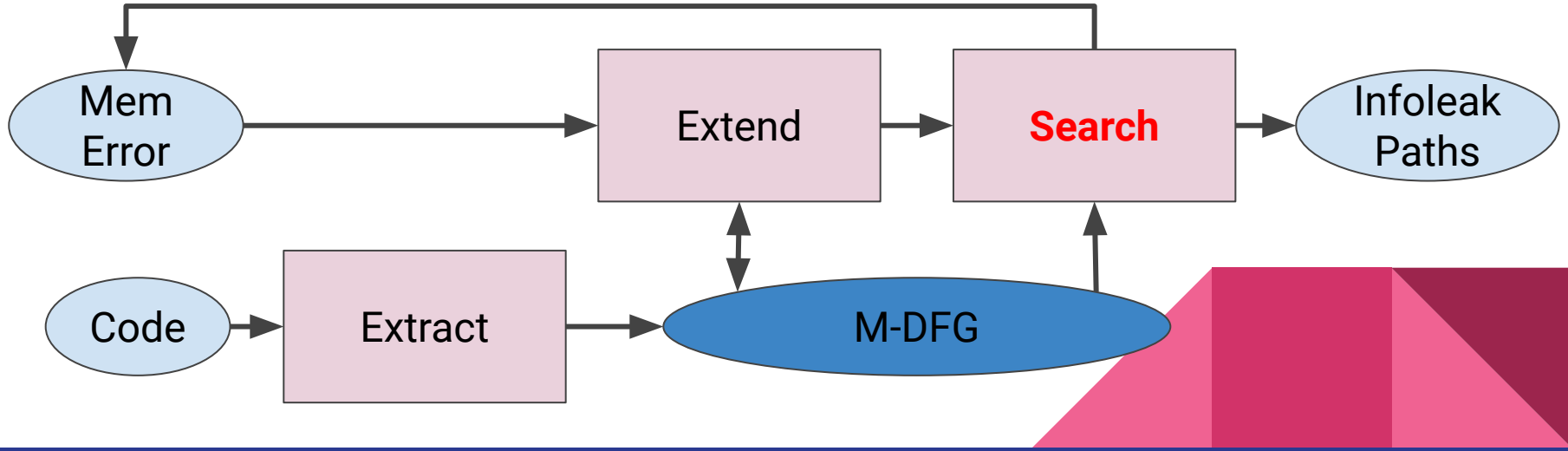
val1 -> s -> l -> val2
if both ptr alias

Objects in points-to info

Workflow: Search on M-DFG

In each iteration, extend M-DFG and do **two searches** on M-DFG

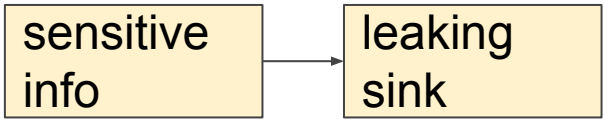
- Infoleak
- Controlled pointers (new memory errors)



Workflow: Search on M-DFG (cont.)

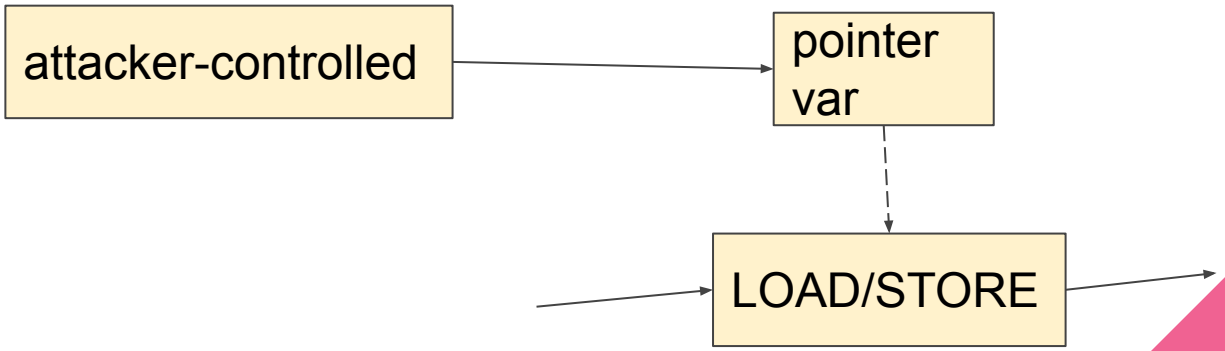
Infoleak search

- **A path in M-DFG** can transfer info to leaking sink.



New memory error search

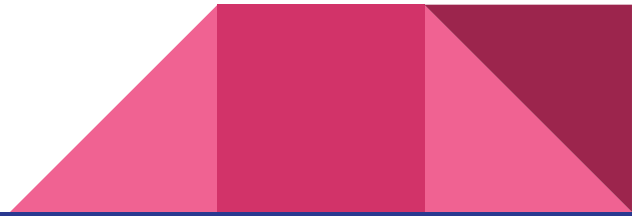
- Look for r/w **pointers** controlled by the attacker.



Workflow: Search on M-DFG (cont.)

Dynamic verification to verify each **data-flow path**

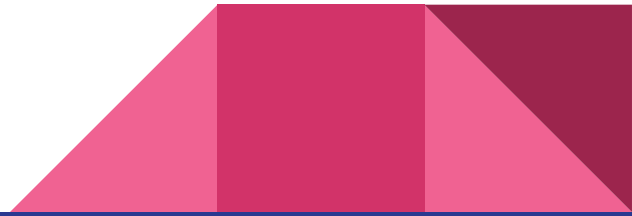
- Not all **infoleak paths** in M-DFG are valid
  - CFG
  - RAW edges
- SymExe
  - segment-by-segment



Evaluation

250 syzbot-exposed memory bugs

- K-LEAK is able to find infoleak paths in 21 bug reports
- Four kinds of **infoleak strategies**
  - R, W, R+W, R+R



Evaluation

11 CVEs

- 7 successful cases

Failure cases

- Cannot create **illegal free** primitive
- Infoleak through control-flow
- Stack memory error

Conclusion

K-LEAK automates the infoleak exploits for Linux kernel

Uncovers various **exploit strategies**

Find previously unknown **infoleaks**

Thank you!