Statically Discover Cross-Entry Use-After-Free Vulnerabilities in the Linux Kernel

Hang Zhang (*IU Bloomington*), Jangha Kim (*The Affiliated Institute of ETRI*), Chuhong Yuan (*Georgia Tech*), Zhiyun Qian (*UC Riverside*) and Taesoo Kim (*Georgia Tech*)

02/25/2025

Use-After-Free: The Classic Problem

Consistently ranked as a most **dangerous** vulnerability in CWE Top 25 list.

Prevalent in critical software (e.g., Linux kernel).

Severe **consequences** (e.g., arbitrary code execution, privilege escalation).

```
int foo(void) {
  int *ptr = malloc(sizeof(int));
  free(ptr); // FREE
  return *ptr; // USE
}
```

Use-After-Free: The Classic Problem

Consistently ranked as a most **dangerous** vulnerability in CWE Top 25 list.

Prevalent in critical software (e.g., Linux kernel).

Severe **consequences** (e.g., arbitrary code execution, privilege escalation).

```
int foo(void) {
  int *ptr = malloc(sizeof(int));
  free(ptr); // FREE
  return *ptr; // USE
}
```

Easily detectable.

UAF can be Tricky: Cross-Entry

- Use and free can happen in **different entry functions** with global variable relays.

```
int *gp, *gq;
```

```
void entry0(void) {
  int *ptr = malloc(...);
  gp = ptr;
  free(ptr); // FREE
}
```

```
void entry1(void) {
  gq = gp;
}
```

```
void entry2(void) {
  return *gq; // USE
}
```

UAF can be Tricky: Cross-Entry

- Use and free can happen in **different entry functions** with global variable relays.

```
int *gp, *gq;
```

(1) Escape

```
void entry0(void) {
  int *ptr = malloc(...);
  gp = ptr;
  free(ptr); // FREE
}
```

```
void entry1(void) {
  gq = gp;
}
```

```
void entry2(void) {
  return *gq; // USE
}
```

UAF can be Tricky: Cross-Entry

- Use and free can happen in **different entry functions** with global variable relays.

```
int *gp, *gq;
```

(1) Escape

```
void entry0(void) {
  int *ptr = malloc(...);
  gp = ptr;
  free(ptr); // FREE (2)
}
```

```
void entry1(void) {
  gq = gp;
}
```

```
void entry2(void) {
  return *gq; // USE
}
```

UAF can be Tricky: Cross-Entry

- Use and free can happen in **different entry functions** with global variable relays.

```
int *gp, *gq;
```

(1) Escape

(3) Fetch

(4) Escape

```
void entry0(void) {
  int *ptr = malloc(...);
  gp = ptr;
  free(ptr); // FREE (2)
}
```

```
void entry1(void) {
  gq = gp;
}
```

```
void entry2(void) {
  return *gq; // USE
}
```

UAF can be Tricky: Cross-Entry

- Use and free can happen in **different entry functions** with global variable relays.

```
int *gp, *gq;
```

(1) Escape
(2) FREE
(3) Fetch
(4) Escape
(5) Fetch
(6) USE

```
void entry0(void) {
  int *ptr = malloc(...);
  gp = ptr;
  free(ptr); // FREE (2)
}
```

```
void entry1(void) {
  gq = gp;
}
```

```
void entry2(void) {
  return *gq; // USE (6)
}
```

UAF can be Tricky: Cross-Entry

- Use and free can happen in **different entry functions** with global variable relays.

- *Common in codebases with multiple entry functions (e.g., the Linux kernel).*

```
int *gp, *gq;
```

```
void entry0(void) {
  int *ptr = malloc(...);
  gp = ptr;
  free(ptr); // FREE (2)
}
```

```
void entry1(void) {
  gq = gp;
}
```

```
void entry2(void) {
  return *gq; // USE (6)
}
```

(1) Escape

(3) Fetch

(4) Escape

(5) Fetch

UAF can be Tricky: Subtle Constraints

- Despite the lock protection and sanity check, UAF still happens due to **subtle flaws**.

```
void entry0(void) {
  lock(o);
  free(gp); // FREE
  unlock(o);
  gp = NULL; //SET
}
```

```
void entry1(void) {
  lock(o);
  if (gp) // CHECK
    *gp = 1; // USE
  unlock(o);
}
```

UAF can be Tricky: Subtle Constraints

- Despite the lock protection and sanity check, UAF still happens due to **subtle flaws**.

```
void entry0(void) {
  lock(o);
  free(gp); // FREE      ①
  unlock(o);
  gp = NULL; //SET
}
```

```
void entry1(void) {
  lock(o);
  if (gp) // CHECK
    *gp = 1; // USE
  unlock(o);
}
```

UAF can be Tricky: Subtle Constraints

- Despite the lock protection and sanity check, UAF still happens due to **subtle flaws**.

```
void entry0(void) {
  lock(o);
  free(gp); // FREE
  unlock(o);
  gp = NULL; //SET
}
```

①

②

```
void entry1(void) {
  lock(o);
  if (gp) // CHECK
    *gp = 1; // USE
  unlock(o);
}
```

UAF can be Tricky: Subtle Constraints

- Despite the lock protection and sanity check, UAF still happens due to **subtle flaws**.

```
void entry0(void) {
  lock(o);
  free(gp); // FREE      ①
  unlock(o);
  gp = NULL; //SET       ③
}
```

```
void entry1(void) {
  lock(o);
  if (gp) // CHECK       ②
    *gp = 1; // USE
  unlock(o);
}
```

UAF can be Tricky: Subtle Constraints

- Despite the lock protection and sanity check, UAF still happens due to **subtle flaws**.

- *Fix: Swap line 03 and 04.*

```
00  void entry0(void) {
01    lock(o);
02    free(gp); // FREE
03    unlock(o);
04    gp = NULL; //SET
05  }
```

```
06  void entry1(void) {
07    lock(o);
08    if (gp) // CHECK
09      *gp = 1; // USE
10    unlock(o);
11  }
```

Our Goal and Challenges

Statically discover cross-entry UAFs in the Linux kernel (and potentially more).

- More systematic than fuzzing.

Our Goal and Challenges

Statically discover cross-entry UAFs in the Linux kernel (and potentially more).

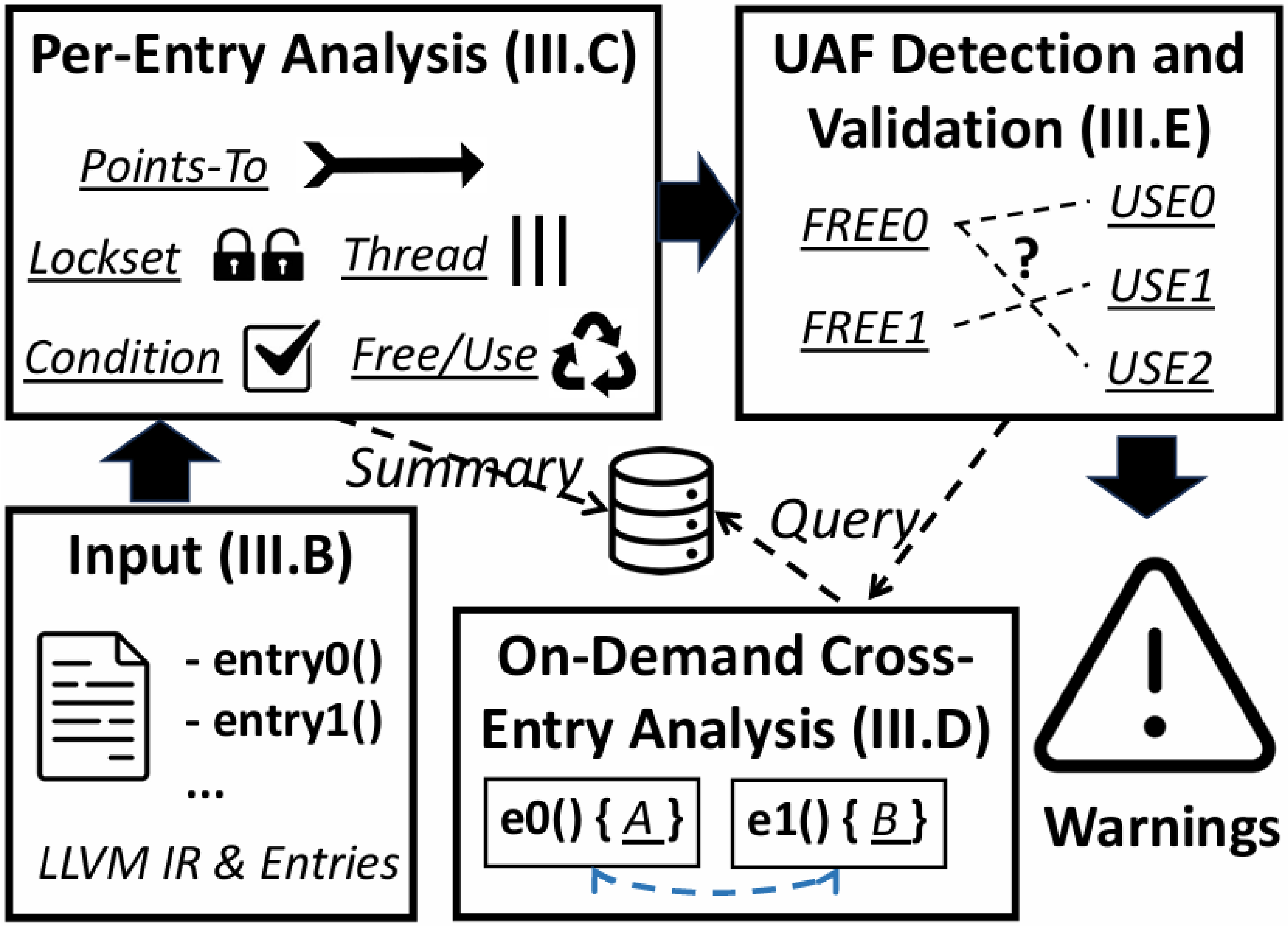
- *More systematic than fuzzing.*

Challenges

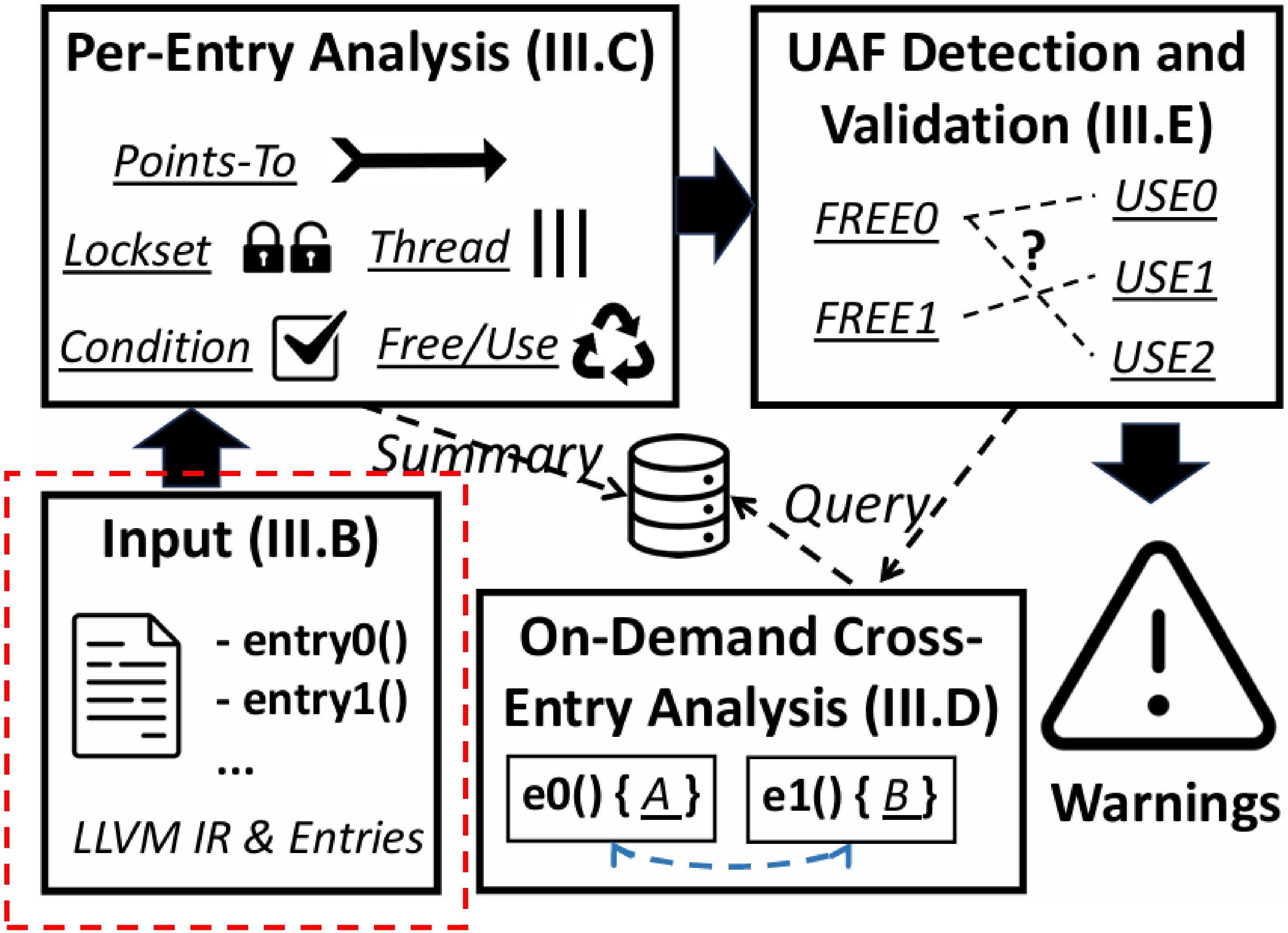
#1: precise and efficient cross-entry alias analysis (between use and free).

#2: comprehensive multi-aspect UAF validation (e.g., lock, condition check, etc.).

Our Solution: UAFX ("X" for "Xross")



Our Solution: UAFX (“X” for “Xross”)



UAFX: Input

```
@entry0 (    ) {
   %0 = load ...;
   store ...;
   %1 = gep ...;
}
@entry1 (...) {
   ...
}
@entry2 (    ) {
   ...
}

...
```

LLVM

Target Program in LLVM Bitcode

\+

Entry Functions
entry0()
entry1()
entry2()

Entry Function List

UAFX: Identify Cross-Entry UAF Candidates

- **Step 1**: Per-entry alias and escape-fetch analysis → Entry summaries

```
int *gp, *gq;
```

```
void entry0(void) {
  int *ptr = malloc(...);
  gp = ptr;
  free(ptr); // FREE
}
```

```
void entry1(void) {
  gq = gp;
}
```

```
void entry2(void) {
  return *gq; // USE
}
```

UAFX: Identify Cross-Entry UAF Candidates

- **Step 1**: Per-entry alias and escape-fetch analysis → Entry summaries

```
int *gp, *gq;
```

(1)

```
00 void entry0(void) {
01   int *ptr = malloc(...);
02   gp = ptr;
03   free(ptr); // FREE   (2)
04 }
```

```
05 void entry1(void) {
06   gq = gp;
07 }
```

```
08 void entry2(void) {
09   return *gq; // USE
10 }
```

Summary:

(1) obj@1 → (escape) → gp

(2) obj@1: freed@3

UAFX: Identify Cross-Entry UAF Candidates

- **Step 1**: Per-entry alias and escape-fetch analysis → Entry summaries

```
int *gp, *gq;
```

(1) (3) (4)

```
00 void entry0(void) {
01   int *ptr = malloc(...);
02   gp = ptr;
03   free(ptr); // FREE   (2)
04 }
```

```
05 void entry1(void) {
06   gq = gp;
07 }
```

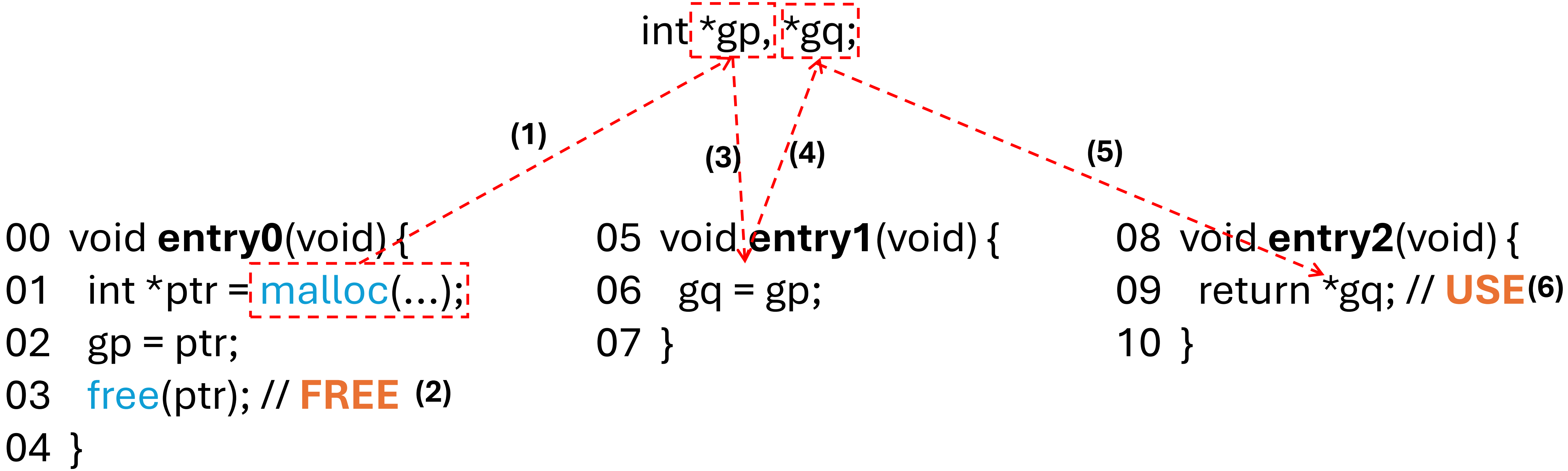
```
08 void entry2(void) {
09   return *gq; // USE
10 }
```

Summary:
(1) obj@1 → (escape) → gp
(2) obj@1: freed@3

Summary:
(3) dobj@6 ← (fetch) ← gp
(4) dobj@6 → (escape) → gq

UAFX: Identify Cross-Entry UAF Candidates

- **Step 1**: Per-entry alias and escape-fetch analysis → Entry summaries



Summary:
(1) obj@1 → (escape) → gp
(2) obj@1: freed@3

Summary:
(3) dobj@6 ← (fetch) ← gp
(4) dobj@6 → (escape) → gq

Summary:
(5) dobj@9 ← (fetch) ← gq
(6) dobj@9: used@9

UAFX: Identify Cross-Entry UAF Candidates

- **Step 1**: Per-entry alias and escape-fetch analysis → Entry summaries

- *Accurate*: *Interprocedural, flow-, context-, field-, and opportunistically path-sensitive.*

```
int *gp, *gq;
```

(1) (3) (4) (5)

```
00 void entry0(void) {
01   int *ptr = malloc(...);
02   gp = ptr;
03   free(ptr); // FREE (2)
04 }
```

Summary:
(1) obj@1 → (escape) → gp
(2) obj@1: freed@3

```
05 void entry1(void) {
06   gq = gp;
07 }
```

Summary:
(3) dobj@6 ← (fetch) ← gp
(4) dobj@6 → (escape) → gq

```
08 void entry2(void) {
09   return *gq; // USE(6)
10 }
```

Summary:
(5) dobj@9 ← (fetch) ← gq
(6) dobj@9: used@9

UAFX: Identify Cross-Entry UAF Candidates

- **Step 2**: Find cross-entry aliased use/free pairs with escape-fetch paths.

- *Efficient:* ***On-demand*** *summary query.*

```
int *gp, *gq;
```

(1) (3) (4) (5)

```
00 void entry0(void) {
01   int *ptr = malloc(...);
02   gp = ptr;
03   free(ptr); // FREE (2)
04 }
```

Summary:
(1) obj@1 → (escape) → gp
(2) obj@1: freed@3

```
05 void entry1(void) {
06   gq = gp;
07 }
```

Summary:
(3) dobj@6 ← (fetch) ← gp
(4) dobj@6 → (escape) → gq

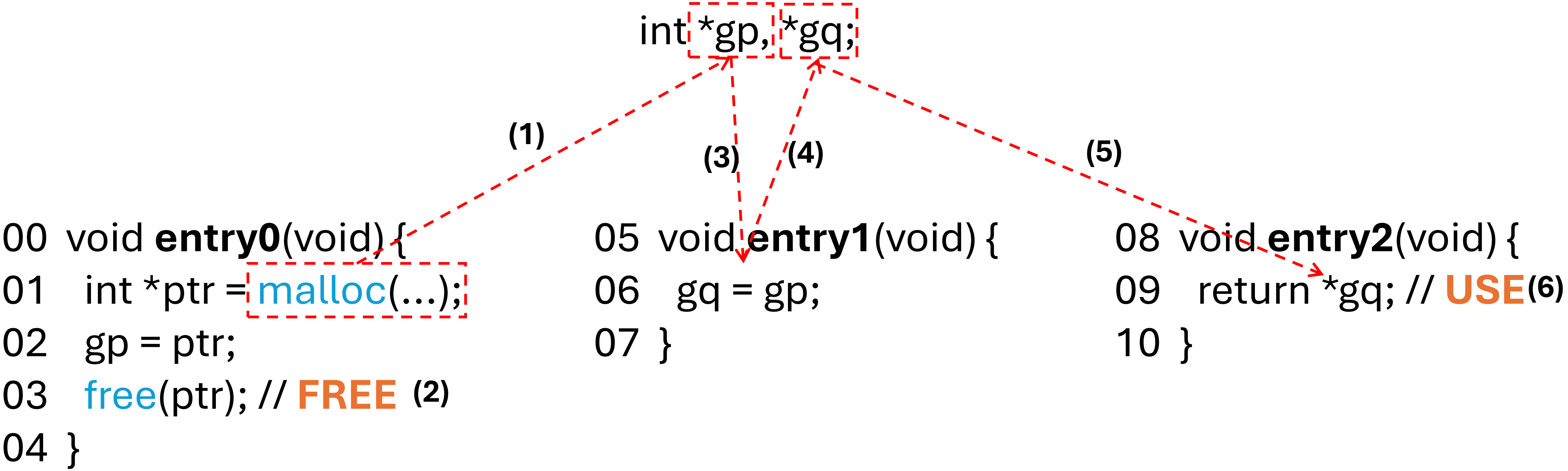
```
08 void entry2(void) {
09   return *gq; // USE (6)
10 }
```

Summary:
(5) dobj@9 ← (fetch) ← gq
(6) dobj@9: used@9

UAFX: Identify Cross-Entry UAF Candidates

- **Step 2**: Find cross-entry aliased use/free pairs with escape-fetch paths.

- *Efficient:* ***On-demand*** *summary query.*



Summary:
(1) obj@1 → (escape) → gp
(2) obj@1: freed@3

Summary:
(3) dobj@6 ← (fetch) ← gp
(4) dobj@6 → (escape) → gq

Summary:
(5) dobj@9 ← (fetch) ← gq
(6) dobj@9: used@9

UAFX: Identify Cross-Entry UAF Candidates

- **Step 2**: Find cross-entry aliased use/free pairs with escape-fetch paths.

- *Efficient:* ***On-demand*** *summary query.*

```
int *gp, *gq;
```

(1) (3) (4) (5)

```
00 void entry0(void) {
01   int *ptr = malloc(...);
02   gp = ptr;
03   free(ptr); // FREE (2)
04 }
```

Summary:
(1) obj@1 → (escape) → gp
(2) obj@1: freed@3

```
05 void entry1(void) {
06   gq = gp;
07 }
```

Summary:
(3) dobj@6 ← (fetch) ← gp
(4) dobj@6 → (escape) → gq

```
08 void entry2(void) {
09   return *gq; // USE (6)
10 }
```

Summary:
(5) dobj@9 ← (fetch) ← gq
(6) dobj@9: used@9

UAFX: Identify Cross-Entry UAF Candidates

- **Step 2**: Find cross-entry aliased use/free pairs with escape-fetch paths.

- *Efficient:* ***On-demand*** *summary query.*

```
int *gp, *gq;
```

(1) (3) (4) (5)

```
00 void entry0(void) {
01   int *ptr = malloc(...);
02   gp = ptr;
03   free(ptr); // FREE (2)
04 }
```

Summary:
(1) obj@1 → (escape) → gp
(2) obj@1: freed@3

```
05 void entry1(void) {
06   gq = gp;
07 }
```

Summary:
(3) dobj@6 ← (fetch) ← gp
(4) dobj@6 → (escape) → gq

```
08 void entry2(void) {
09   return *gq; // USE(6)
10 }
```

Summary:
(5) dobj@9 ← (fetch) ← gq
(6) dobj@9: used@9

UAFX: Identify Cross-Entry UAF Candidates

- **Step 2**: Find cross-entry aliased use/free pairs with escape-fetch paths.

- *Efficient:* ***On-demand*** *summary query.*

```
int *gp, *gq;
```

(1) (3) (4) (5)

```
00 void entry0(void) {
01   int *ptr = malloc(...);
02   gp = ptr;
03   free(ptr); // FREE (2)
04 }
```

Summary:
(1) obj@1 → (escape) → gp
(2) obj@1: freed@3

```
05 void entry1(void) {
06   gq = gp;
07 }
```

Summary:
(3) dobj@6 ← (fetch) ← gp
(4) dobj@6 → (escape) → gq

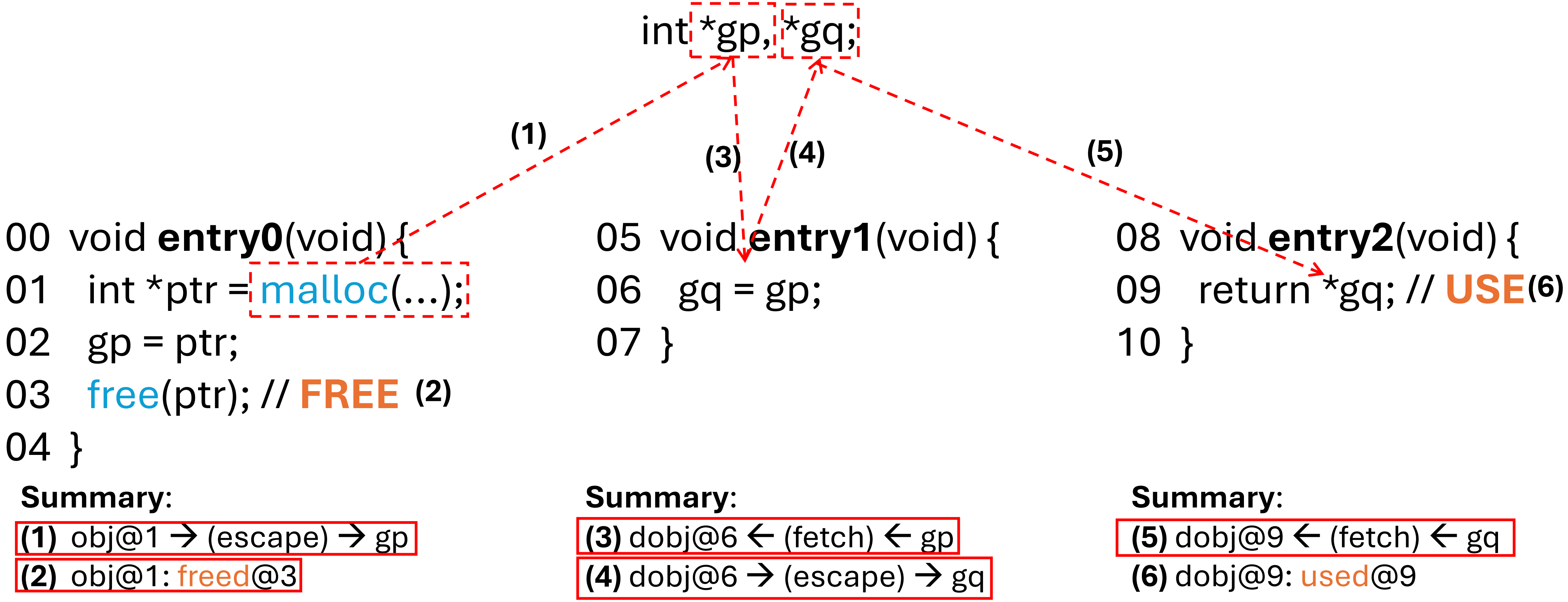
```
08 void entry2(void) {
09   return *gq; // USE (6)
10 }
```

Summary:
(5) dobj@9 ← (fetch) ← gq
(6) dobj@9: used@9

UAFX: Identify Cross-Entry UAF Candidates

- **Step 2**: Find cross-entry aliased use/free pairs with escape-fetch paths.

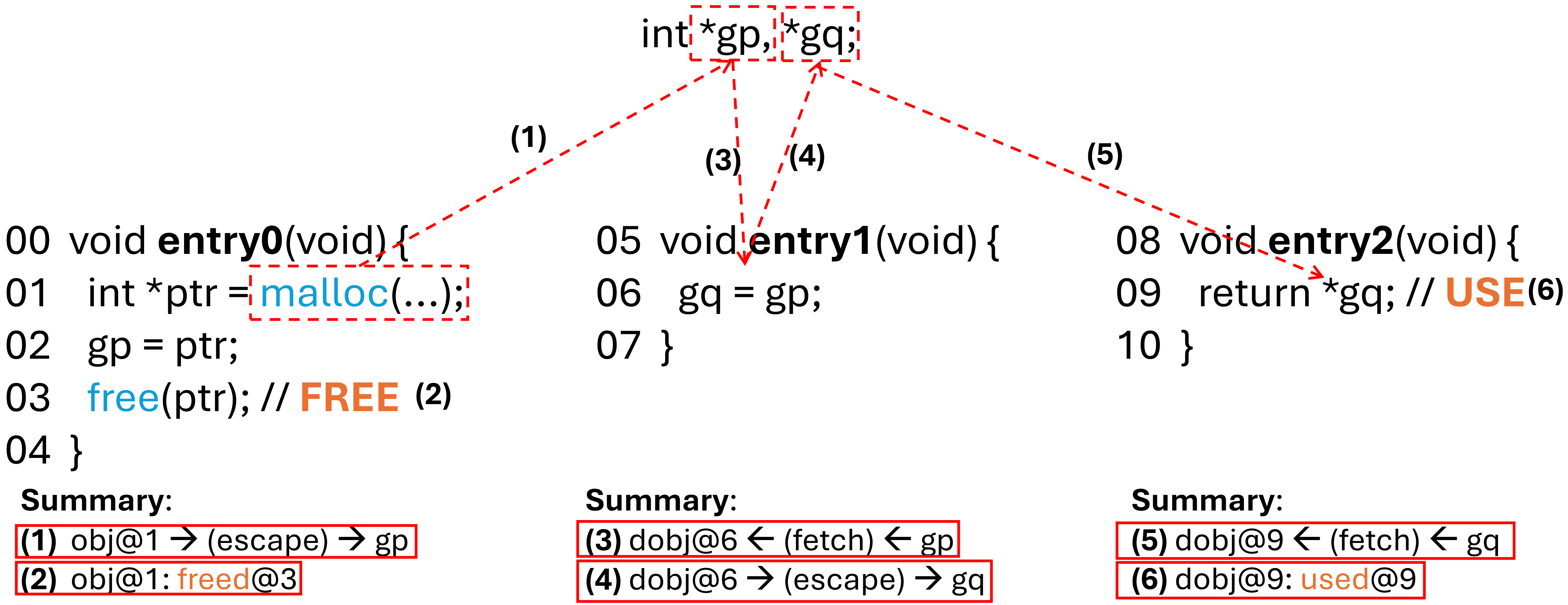
- *Efficient:* ***On-demand*** *summary query.*



UAFX: Identify Cross-Entry UAF Candidates

- **Step 2**: Find cross-entry aliased use/free pairs with escape-fetch paths.

- *Efficient:* ***On-demand*** *summary query.*



UAFX: Partial-Order-Based UAF Validation

```
void entry0(void) {
  lock(o);
  free(gp); // FREE
  unlock(o);
  gp = NULL; //SET
}
```

```
void entry1(void) {
  lock(o);
  if (gp) // CHECK
    *gp = 1; // USE
  unlock(o);
}
```

- **Step 1**: Identify relevant statements (e.g., lock/unlock, condition set/check) and perform the **cross-entry** match (e.g., lock objects, condition variables).

- *Per-entry summary + on-demand cross-entry query.*

UAFX: Partial-Order-Based UAF Validation

```
00 void entry0(void) {
01   lock(o);
02   free(gp); // FREE
03   unlock(o);
04   gp = NULL; //SET
05 }
```

```
06 void entry1(void) {
07   lock(o);
08   if (gp) // CHECK
09     *gp = 1; // USE
10   unlock(o);
11 }
```

Aliased lock pairs:
L01 (lock) - L03 (unlock),
L07 (lock) - L10 (unlock)

- **Step 1**: Identify relevant statements (e.g., lock/unlock, condition set/check) and perform the **cross-entry** match (e.g., lock objects, condition variables).

- *Per-entry summary + on-demand cross-entry query.*

UAFX: Partial-Order-Based UAF Validation

```
00  void entry0(void) {
01    lock(o);
02    free(gp); // FREE
03    unlock(o);
04    gp = NULL; //SET
05  }
```

```
06  void entry1(void) {
07    lock(o);
08    if (gp) // CHECK
09      *gp = 1; // USE
10    unlock(o);
11  }
```

Aliased lock pairs:
L01 (lock) - L03 (unlock),
L07 (lock) - L10 (unlock)

Aliased cond. set/check:
L04 (set) → (kill) → L08 (check)

- **Step 1**: Identify relevant statements (e.g., lock/unlock, condition set/check) and perform the **cross-entry** match (e.g., lock objects, condition variables).

- *Per-entry summary + on-demand cross-entry query.*

UAFX: Partial-Order-Based UAF Validation

```
void entry0(void) {
  lock(o);
  free(gp); // FREE
  unlock(o);
  gp = NULL; //SET
}
```

```
void entry1(void) {
  lock(o);
  if (gp) // CHECK
    *gp = 1; // USE
  unlock(o);
}
```

- **Step 2**: Unify all necessary UAF conditions in an extensible partial-order system – solvable by a SMT solver (e.g., z3).

- Solution exists → the UAF is feasible.

UAFX: Partial-Order-Based UAF Validation

```
00  void entry0(void) {
01    lock(o);
02    free(gp); // FREE
03    unlock(o);
04    gp = NULL; //SET
05  }
```

```
06  void entry1(void) {
07    lock(o);
08    if (gp) // CHECK
09      *gp = 1; // USE
10    unlock(o);
11  }
```

< : happens before

Lock semantics:

L03 < L07 or L10 < L01

(critical sections cannot overlap)

- **Step 2**: Unify all necessary UAF conditions in an extensible partial-order system – solvable by a SMT solver (e.g., z3).

- Solution exists → the UAF is feasible.

UAFX: Partial-Order-Based UAF Validation

```
00 void entry0(void) {
01   lock(o);
02   free(gp); // FREE
03   unlock(o);
04   gp = NULL; //SET
05 }
```

```
06 void entry1(void) {
07   lock(o);
08   if (gp) // CHECK
09     *gp = 1; // USE
10   unlock(o);
11 }
```

< : happens before

Lock semantics:

L03 < L07 or L10 < L01

(critical sections cannot overlap)

Condition semantics:

L08 < L04

(to reach use site the pointer nullification cannot happen in prior)

- **Step 2**: Unify all necessary UAF conditions in an extensible partial-order system – solvable by a SMT solver (e.g., z3).

- Solution exists → the UAF is feasible.

UAFX: Partial-Order-Based UAF Validation

```
00 void entry0(void) {
01   lock(o);
02   free(gp); // FREE
03   unlock(o);
04   gp = NULL; //SET
05 }
```

```
06 void entry1(void) {
07   lock(o);
08   if (gp) // CHECK
09     *gp = 1; // USE
10   unlock(o);
11 }
```

< : happens before

Lock semantics:

L03 < L07 or L10 < L01

(critical sections cannot overlap)

Condition semantics:

L08 < L04

(to reach use site the pointer nullification cannot happen in prior)

Sequential order:

L01 < L02 < L03 < L04

L07 < L08 < L09 < L10

- **Step 2**: Unify all necessary UAF conditions in an extensible partial-order system – solvable by a SMT solver (e.g., z3).

- Solution exists → the UAF is feasible.

UAFX: Partial-Order-Based UAF Validation

```
00 void entry0(void) {
01   lock(o);
02   free(gp); // FREE
03   unlock(o);
04   gp = NULL; //SET
05 }
```

```
06 void entry1(void) {
07   lock(o);
08   if (gp) // CHECK
09     *gp = 1; // USE
10   unlock(o);
11 }
```

< : happens before

Lock semantics:

L03 < L07 or L10 < L01

(critical sections cannot overlap)

Condition semantics:

L08 < L04

(to reach use site the pointer nullification cannot happen in prior)

Sequential order:

L01 < L02 < L03 < L04

L07 < L08 < L09 < L10

......

- **Step 2**: Unify all necessary UAF conditions in an extensible partial-order system – solvable by a SMT solver (e.g., z3).

- Solution exists → the UAF is feasible.

Implementation and Evaluation

- UAFX is implemented on top of our previous LLVM-based static analyzer – SUTURE, with about 13K LOC ++ and 4K LOC --.
- UAFX will be open-sourced: https://github.com/uafx/uafx

Implementation and Evaluation

- UAFX is implemented on top of our previous LLVM-based static analyzer – SUTURE, with about 13K LOC ++ and 4K LOC --.
- UAFX will be open-sourced: https://github.com/uafx/uafx

Capability of finding subtle cross-entry UAFs

- **Dataset**: 23 Linux device driver UAFs by Syzbot.
- UAFX can find significantly **more** bugs in this dataset (15/23) than SOTA tools.

Implementation and Evaluation

- UAFX is implemented on top of our previous LLVM-based static analyzer – SUTURE, with about 13K LOC ++ and 4K LOC --.
- UAFX will be open-sourced: https://github.com/uafx/uafx

Capability of finding subtle cross-entry UAFs

- **Dataset**: 23 Linux device driver UAFs by Syzbot.
- UAFX can find significantly **more** bugs in this dataset (15/23) than SOTA tools.

Discover new UAFs

- **Subjects** : 34 Linux kernel device driver modules and 1 user-space program.
- UAFX issues **80** true positive warnings, where **37** have been confirmed (related to **10** independent UAF issues).

Implementation and Evaluation

- UAFX is implemented on top of our previous LLVM-based static analyzer – SUTURE, with about 13K LOC ++ and 4K LOC --.
- UAFX will be open-sourced: https://github.com/uafx/uafx

Accuracy

- **Reviewer-perceived false alarm rate**: around **60%** after filtering highly similar false positives.
- **False negatives**: we intentionally give up warnings involving refcnt and recursive structures (more FNs → less FPs).

Implementation and Evaluation

- UAFX is implemented on top of our previous LLVM-based static analyzer – SUTURE, with about 13K LOC ++ and 4K LOC --.
- UAFX will be open-sourced: https://github.com/uafx/uafx

Accuracy

- **Reviewer-perceived false alarm rate**: around **60%** after filtering highly similar false positives.
- **False negatives**: we intentionally give up warnings involving refcnt and recursive structures (more FNs → less FPs).

Efficiency

- Running time **varies** for different targets, ranging from seconds to 30+ hours (for a large driver).
- More expensive than other tools due to analysis complexity.

Thank You!
Q & A