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

Hang Zhang*, Jangha Kim†, Chuhong Yuan§, Zhiyun Qian‡ and Taesoo Kim§
*Indiana University Bloomington
hz64@iu.edu
†The Affiliated Institute of ETRI, ROK
jangha@nsr.re.kr
‡University of California, Riverside
zhiyunq@cs.ucr.edu
§Georgia Institute of Technology
{chyuan,taesoo}@gatech.edu

***Abstract*—Use-After-Free (UAF) is one of the most widely spread and severe memory safety issues, attracting lots of research efforts toward its automatic discovery. Existing UAF detection approaches include two major categories: dynamic and static. While dynamic methods like fuzzing can detect UAF issues with high precision, they are inherently limited in code coverage. Static approaches, on the other hand, can usually only discover simple sequential UAF cases, despite that many real-world UAF bugs involve intricate cross-entry control and data flows (*e.g.,* concurrent UAFs). Limited static tools supporting cross-entry UAF detection also suffer from inaccuracy or narrowed scope (*e.g.,* cannot handle complex codebases like the Linux kernel).**

In this paper, we propose UAFX, a static analyzer capable of discovering cross-entry UAF vulnerabilities in the Linux kernel and potentially extensible to general C programs. UAFX is powered by a novel escape-fetch-based cross-entry alias analysis, enabling it to accurately analyze the alias relationships between the use and free sites even when they scatter in different entry functions. UAFX is also equipped with a systematic UAF validation framework based on partial-order constraints, allowing it to reliably reason about multiple UAF-related code aspects (*e.g.,* locks, path conditions, threads) to filter out false alarms. Our evaluation shows that UAFX can discover new cross-entry UAF vulnerabilities in the kernel and one user-space program (80 true positive warnings), with reasonable reviewer-perceived precision (more than 40%) and performance.

I. Introduction

Use-After-Free (UAF) is a classic temporal memory safety problem, despite the long history and numerous research efforts, it is still among the most common and severe vulnerability types today (*e.g.,* consistently ranked highly in CWE top weaknesses [1]). Due to its prevalence, many previous works are attempting to automatically discover UAF bugs, which can be generally classified into dynamic and static categories based on their methodology. A dynamic approach (*e.g.,* fuzzing) executes the target program with a limited set of input and watches for the potential runtime buggy behaviors (*e.g.,* with the help of sanitizers such as ASan [2]). Though precise (*e.g.,* the reported bugs are usually true positives), dynamic approaches generally suffer from low code/state coverage due to their random exploration nature [3].

Alternatively, static approaches systematically scan the code for UAFs, achieving a higher coverage. However, existing static UAF detectors still fall short of spotting sophisticated UAF issues. Specifically, they can be classified into two categories:

First, most static UAF detectors [4], [5], [6], [7], [8], [9] only support identifying simple sequential UAF bugs, where the use happens straightly after free either within the same function (intraprocedural) or the same call chain (interprocedural). However, as shown by both our study (§V-B) and previous work [10], many real-world UAF cases involve complex control and data flows across multiple entry functions (*i.e.,* top-level functions serving as entry points, such as Linux system calls). For example, memory can be freed in one system call while accessed in another, with apparently different but indeed aliased pointers. Our study (§V-B) reveals that more than 70% of Linux device driver UAFs found by Syzkaller [11], a most popular 24/7 kernel fuzzer, are cross-entry.

Second, some works (only DCUAF [10] and Canary [12] to our best knowledge) provide support for cross-entry UAF detection, but to limited degrees. For example, DCUAF [10] employs a coarse-grained type- and field-based alias analysis between use and free sites, as well as a simplified FP filtering heuristic (*e.g.,* single-aspect lockset analysis), which can lead to significant inaccuracies (*e.g.,* use and free sites may be regarded as aliased or not, incorrectly). Canary [12] is limited to the `fork()`-based multi-threading model, however, most Linux kernel cross-entry UAF issues are based on the alias relationships centered around global variables, rendering its

*Work partially done when the leading author was a postdoctoral researcher at Georgia Tech.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240559
www.ndss-symposium.org

approach ineffective. We discuss in more detail and perform comparative evaluations on these tools in §V-D.

In this paper, we propose UAFX, a static analyzer capable of discovering complex cross-entry UAF bugs in the Linux kernel with highly accurate cross-entry reasoning. UAFX also has the potential to support general C programs. We employ two novel techniques to address two key challenges in cross-entry UAF detection, respectively:

(1) Escape-Fetch-based Cross-Entry Alias Analysis. To accurately capture the cross-entry aliases, we develop an innovative *escape-fetch* analysis to track cross-entry memory object propagation. Specifically, we model the object assignment(retrieval) to(from) shared pointers as `escape`(`fetch`) and conduct an accurate per-entry analysis (§III-C) to summarize all such `escape` and `fetch` within the entry. Then, by querying the per-entry summaries, UAFX efficiently constructs cross-entry *escape-fetch* paths on-demand (§III-D), establishing the cross-entry alias relationship between use and free sites and laying the foundation for cross-entry UAF detection.

(2) Systematic Partial-Order Constraint-based UAF Validation. Confirming the use/free alias relationship is only the first step of UAF discovery, validating the UAF issues still needs the reasoning of many different code aspects. For example, a `flag` may be set to `true` following a free site, while the use site is guarded by the check of `flag`, preventing the UAF. This could be further entangled with code aspects like lock/unlock and multi-thread mechanisms. All these aspects must be systematically reasoned about to validate the UAF, while previous works only consider a part of them, leading to inaccuracies. To address this problem, UAFX comprehensively models UAF-relevant code aspects and encodes them in a unified and extensible partial-order constraint system (§III-E), whose feasibility can be efficiently decided (*e.g.,* by SMT solvers like z3 [13]), making a systematic UAF validation framework.

We implement the prototype of UAFX and evaluate it against both the Linux kernel and a selected user-space program. UAFX successfully discovers unknown cross-entry UAF vulnerabilities (80 true positive warnings) with reasonable reviewer-perceived precision (around 40%, as detailed in §V-A) and efficiency (§V-C). We have reported our findings to developers and so far 37 warnings have been confirmed, corresponding to 10 independent underlying UAF issues. We summarize our contributions as follows:

(1) We distill two key challenges in cross-entry UAF discovery and devise innovative techniques to effectively address them.

(2) We implement our ideas in UAFX, a prototype capable of statically discovering cross-entry UAF bugs in the Linux kernel and potentially extensible to general C programs. UAFX will be open-sourced[1].

(3) UAFX is evaluated in Linux kernel and a user-space C program, successfully discovering multiple new cross-entry UAFs with practical accuracy and performance.

[1]https://github.com/uafx/uafx

II. Overview

In this section, we first illustrate the major challenges involved in cross-entry UAF detection with a motivating example (§II-A), and then provide an overview of how UAFX can overcome these challenges and identify the sophisticated UAF cases, step by step (§II-B).

A. Challenges

Fig. 1 depicts cross-entry UAF examples distilled from real-world code. There are four independently invokable entry functions (*i.e.,* `entry0() - entry3()`), `entry0()` allocates a heap buffer (line 1) and assigns it to a global pointer `g0` (line 2), the buffer is then freed via the local pointer `p` (line 6) with a global indicator `flag` set to `true` (line 4), leaving `p` and `g0` dangling. At this point, if we execute `entry1()`, followed by `entry3()`, the use site 1 at line 20 will cause a UAF, because `g1` is now aliased to the dangling pointer `p`, though they are in different entry functions. However, figuring out such a tangled cross-entry alias relationship is difficult:

Challenge 1: Cross-entry alias analysis for memory regions accessed at the free and use sites.

On the other hand, though the use site 0 (line 15) tries to dereference the same global pointer `g1` as in use site 1, no UAF happens because the global indicator set/check (line 4 and 14), together with the lock protection (line 3, 5, 13, and 16), ensures that the use site 0 cannot execute after the free site at line 6. Missing any of the aforementioned code aspects will make a true UAF. Reasoning about this subtly intertwined protection is not easy even for human auditors:

Challenge 2: Validating UAF issues requires a careful and systematic consideration of all relevant code aspects.

Ideally, the static analysis should both identify the true positive UAF at use site 1 and avoid the false alarm at use site 0 - a difficult task due to the above challenges. In §II-B, we will overview UAFX's solutions to address these challenges.

B. UAFX's Workflow

In this section, we introduce UAFX's high-level workflow for cross-entry UAF discovery, consisting of the below steps.

Step 1: Identify entry functions. First, we locate the entry functions in the target program either automatically or manually (more details in §III-B). In Fig. 1, this step identifies the four entry functions `entry0()` to `entry3()`.

Step 2: Per-entry code analysis and summarization. UAFX then individually summarizes each entry function regarding its UAF-relevant behaviors, including both the memory object *escape/fetch* and other aspects like critical regions (*e.g.,* line 3 - line 5) and condition set/check (*e.g.,* line 4 and 14). For the example in Fig. 1, the object *escape/fetch* is summarized at the top right corner. For example, a concrete memory object (*i.e.,* one with an explicit allocation site in the entry function) is created at line 1 (*i.e., c_obj@1*) and then is assigned or "escapes" to the local pointer `p` at line 1 (edge ⓪) and global pointer `g0` at line 2 (edge ①) - note the edge directions. On the other hand, whatever object pointed to by `g0` is "fetched" (edge ②) and "escapes" to `g1` (edge ③) at line 10 - we

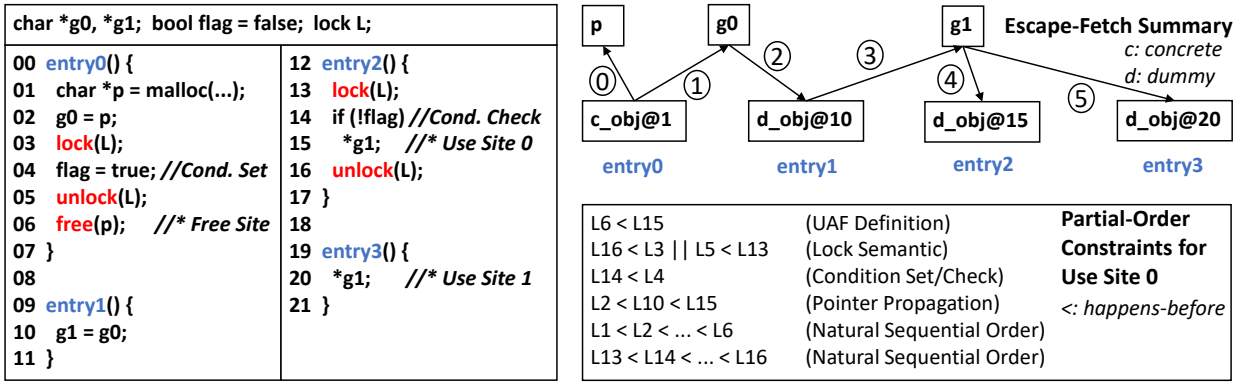

Fig. 1: Motivating UAF Examples

create a dummy object to represent it (*e.g.,* from `entry1()`'s perspective, the object's origin is unknown) in `entry1()`'s summary (*i.e., d_obj@10*). Note that "fetches" are for objects without locally known allocation sites, which is usually not the case for local pointer variables, *e.g.,* at line 2, the object pointed to by `p` is known to be locally allocated at line 1, so there is no "fetch". UAFX tracks all local pointer propagation to generate the final *escape/fetch* summary for global pointers to facilitate cross-entry analysis.

Step 3: Cross-entry recognition of aliased free and use sites. With the per-entry summary, UAFX can recognize the cross-entry aliased free and use sites - potential UAF candidates. For example, use site 1 fetches and accesses whatever object pointed to by `g1` at line 20 (*i.e., d_obj@20* fetched by edge ⑤), by querying the summary in Fig. 1, we can intuitively see that *d_obj@20* is aliased to the freed *c_obj@1* because there is an *escape-fetch* path between them: ① → ② → ③ → ⑤, where an object can first escape to a global pointer and then be fetched by the same pointer in a different entry function, recursively. This makes a potential UAF between the free site (line 6) and the use site 1 (line 20). Similarly, UAFX recognizes that the use site 0 could access the same object as the free site (*i.e.,* an *escape-fetch* path of ① → ② → ③ → ④), making another UAF candidate. These UAF candidates will be further examined in the next step.

Step 4: Systematic UAF validation. This step verifies the feasibility of UAF candidates, by considering a comprehensive set of relevant code aspects. Our basic idea is to encode various code aspects into a unified partial-order constraint system. Take the candidate UAF between the free site and use site 0 as an example, its constraints are shown at the bottom right corner in Fig. 1:

(1) By the UAF definition, the free site (line 6) must happen before the use site (line 15), encoded as a partial-order constraint of $L6 < L15$, where $<$ means "happens-before".

(2) Critical regions protected by the same lock are mutually excluded, so either line 16 executes before line 3, or line 5 before line 13 (*i.e.,* $L16 < L3 \;||\; L5 < L13$).

(3) Use site 0 can only be reached if the condition check at line 14 is satisfied ($flag == false$), which means the conflicting condition set at line 4 ($flag = true$) cannot execute beforehand, leading to the partial-order constraint of $L14 < L4$.

(4) To ensure that the free and use sites access the same object, the expected object *escape-fetch* path must be fulfilled. For example, the fetched object at the use site 0 (*d_obj@15*) must be the one previously escaped to `g1` at line 10 (*d_obj@10*), meaning that the escape (line 10) should happen before the fetch (line 15). Eventually, the *escape-fetch* path of ① → ② → ③ → ④ is encoded in the partial-order constraint of $L2 < L10 < L15$.

(5) Intuitively, the natural intra-entry-function sequential execution order should also be considered, resulting in the trivial (but necessary) constraints as shown in Fig. 1.

All the above canonical partial-order constraints constitute an extensible inequation system encoding various code aspects, whose feasibility can be decided by an SMT solver (*e.g.,* z3 [13]) that attempts to output a possible ordering of relevant statements satisfying all constraints. Note the difference between its more common usage of identifying the value constraints for program variables, as in symbolic execution. For the above example (*e.g.,* between use site 0 and free site), the solver cannot find a statement ordering satisfying all constraints, indicating an unrealistic UAF. While the UAF between use site 1 and the free site can successfully pass the validation, as expected.

III. Design

In this section, we detail the design of UAFX. We start with a set of definitions to facilitate further discussions.

A. Definitions

Def. 1 : A Concrete Object. If a memory object is explicitly allocated (*e.g.,* via `malloc()`) within an entry function, we call it a *concrete object* in the object *escape/fetch* summary of that entry function, written as c_obj.

Example. In Fig. 1, line 1 allocates a concrete object for *entry0()*, denoted as *c_obj@1*.

Def. 2 : A Dummy Object. If an entry function accesses an object allocated outside its scope (*e.g.,* dereference a global pointer), the origin of the object is unknown in the function

scope and it could potentially be one of many *concrete objects* allocated elsewhere (*e.g.,* the global pointer could be assigned different objects depending on the code executed before that entry function). UAFX uses a placeholder *dummy object* (*i.e.,* d_obj) to represent such out-of-scope objects in the per-entry function summaries.

Example. At line 20 in Fig. 1, *entry3()* accesses an object allocated outside its scope via `g1`. We use *d_obj@20* to represent this dummy object in *entry3()*'s summary.

Def. 3 : Object Escape. We say an object *escapes* if it is assigned to any global pointer within an entry function. In this case, the object is no longer confined to that entry function and can be accessed by other entry functions via global pointers. We use $obj \rightarrow p$ to note that obj escapes to pointer p.

Example. At line 2 in Fig. 1, the object pointed to by `p` (*c_obj@1*) escapes to a global pointer `g0`, denoted as $c\_obj@1 \rightarrow g0$.

Def. 4 : Object Fetch. We say an entry function *fetches* an object if for that entry, the object is a *dummy object* (Def. 2). We use $obj \leftarrow p$ to note that obj is *fetched* (*i.e.,* dereferenced) from the pointer p.

Example. At line 20 in Fig. 1, *entry3()* fetches the dummy object *d_obj@20* from `g1` for access, denoted as $d\_obj@20 \leftarrow g1$.

Def. 5 : An Escape-Fetch Path. We recursively define an *escape-fetch* path ("EFP" for short) as follows.

$$EFP = obj0 \rightarrow p0 \rightarrow obj1 \;||\; EFP \rightarrow pn \rightarrow objn$$

Basically, an EFP describes how one memory object (*i.e.,* the *source object*) appearing in one entry function could be eventually accessed in a different entry function invocation (*i.e.,* as a *destination object*) via a series of *escape* and *fetch* (*e.g.,* one entry function may first *fetch* an object and then *escape* it again). We use $Src(EFP)$ to represent the *source object* of the *EFP* and $Dst(EFP)$ for the *destination object*. Since an EFP is composed of one or more *escape-fetch* segments (*e.g.,* $obj0 \rightarrow p0 \rightarrow obj1$), each of which contains one *escape* edge and one *fetch* edge, EFP always has the same number of *escape* and *fetch* edges in total.

Example. In Fig. 1, the object allocated by *entry0()* at line 1 (*c_obj@1*) escapes at line 2 to the global pointer `g0`. It can then be fetched by *entry1()* via the same `g0` at line 10, into the dummy object *d_obj@10*. This *escape-fetch* path is written as $c\_obj@1 \rightarrow g0 \rightarrow d\_obj@10$, where $Src(EFP) = c\_obj@1$ and $Dst(EFP) = d\_obj@10$.

Def. 6 : Cross-Entry Object Aliases. We consider two memory objects $objx$ and $objy$ referenced in different entry function invocations aliased, iff they meet one of the following conditions:

$$\begin{cases} \textbf{Case 1: } \exists EFP, Src(EFP) = objx, Dst(EFP) = objy \\ \textbf{Case 2: } \exists EFP, Src(EFP) = objy, Dst(EFP) = objx \\ \textbf{Case 3: } \exists EFP_1, EFP_2 : Src(EFP_1) = Src(EFP_2), \\ Dst(EFP_1) = objx, Dst(EFP_2) = objy \end{cases}$$

Intuitively, this means that either one object is aliased to the other via an EFP (*e.g.,* they are at the two ends of an EFP), or both objects originate from the same object via two EFPs, respectively. Specifically in Case 1, we call $objy$ is a *forward alias* of $objx$ and $objx$ is a *backward alias* of $objy$, due to their relative positions in the EFP. Similar definitions also apply for Case 2.

Example. In Fig. 1, the accessed object at use site 1 (*d_obj@20*) can be equivalent to the freed object in *entry0()* (*c_obj@1*), through the *EFP*: *c_obj@1* $\rightarrow$ `g0` $\rightarrow$ *d_obj@10* $\rightarrow$ `g1` $\rightarrow$ *d_obj@20* (*i.e.,* ① $\rightarrow$ ② $\rightarrow$ ③ $\rightarrow$ ⑤), where *d_obj@20* is a *forward alias* of *c_obj@1*.

Def. 7 : A Partial-Order Constraint. We use a partial-order relationship of *happens-before* in our UAF validation, specifying the relative execution order of two program statements. The constraint of "*stmt1* must be executed before *stmt2*" is expressed as $stmt1 < stmt2$.

With the above definitions, we then describe the design of each component of UAFX in the remainder of this section. As a roadmap, the high-level architecture and workflow of UAFX are shown in Fig. 2.

B. Entry Identifier

For cross-entry UAF detection, our first step is identifying all entry functions of the program (*e.g.,* the API interfaces, the thread start points, *etc.*). Currently, we achieve this in a semi-automatic way for different target software. Specifically, we first try to automatically extract candidate entry points with the following methods:

General Call Chain Analysis. The functions serving as entry points usually have no further callers inside the program, exploiting this fact, we can perform a call chain analysis of the target program to identify those top-level callers (*e.g.,* not called by any other functions) as entry-point candidates.

Software-Specific Heuristics. Software may follow clear patterns to define entry points, which can be leveraged as domain knowledge. One example is kernel driver modules, which usually encapsulate their entry points (*e.g.,* `ioctl()`) into a `file_operations` structure. UAFX utilizes these specific heuristics to automatically locate entry functions whenever possible.

After obtaining an initial list of candidate entry functions with the above automatic methods, we then perform necessary manual inspection to filter out superfluous entry functions (*e.g.,* inaccurate call graph construction due to issues like indirect call resolution). This process takes one author about one day for all kernel modules we test in §V.

C. Per-Entry Analysis

For each identified entry function (§III-B), UAFX individually analyzes it and summarizes all its behaviors relevant to UAF detection. We describe the detailed code aspects we target in this per-entry analysis as follows.

Escape/Fetch aware Points-To Records. The most essential per-entry analysis UAFX performs is the points-to analysis (*e.g.,* which pointer variable points to which memory object). While the analysis itself is conventional in general, to support the cross-entry alias analysis, our major innovation is to make

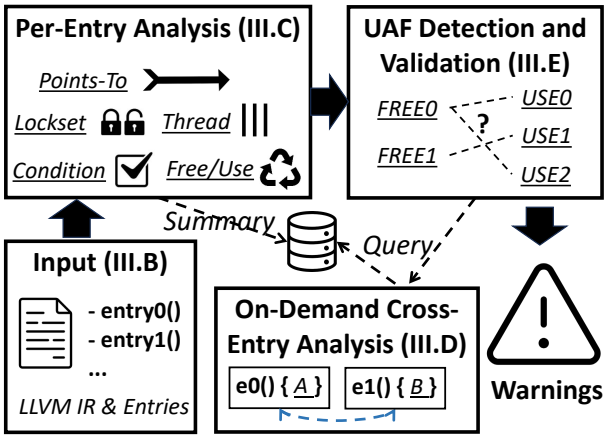

Fig. 2: The Architecture of UAFX

the points-to relationship aware of the object *fetch* (Def. 4) and *escape* (Def. 3). Specifically, besides merely recording that a pointer `p` points to the object `obj`, UAFX also labels the points-to edge with *fetch* or *escape* according to their definitions previously introduced in §III-A(Def. 4, Def. 3). This is achieved by tracking whether *obj* is allocated within the current entry function and whether `p` is a global pointer. For *fetched* object, UAFX will create a placeholder dummy object in the per-entry summary (*i.e., obj* will be a *d_obj*, see Def. 2) to accommodate its multiple possible origins. Such *fetch*/*escape* labels associated with the points-to records enable us to construct the *EFP*s (Def. 5) in the later on-demand cross-entry alias analysis phase.

It is worth noting that per-entry analysis does not mean intraprocedural analysis - the entry function can also invoke other functions, forming different call chains. UAFX, built on top of a state-of-the-art static analyzer [14], employs a standard top-down style context-sensitive interprocedural analysis (*e.g.,* function arguments and return values can be assigned with different points-to records depending on calling contexts). UAFX is also flow-, field-, and partially path-sensitive, with pointer arithmetic handling and indirect call resolution inherited from [14], more details can be found in §IV.

Lockset. UAFX performs a lockset analysis for each entry function as such information is necessary for UAF validation (*e.g.,* our motivating example in Fig. 1). The basic idea is to identify the pairs of lock and unlock functions (*e.g.,* `mutex_lock()` and `mutex_unlock()` in the Linux kernel) that operate on the same lock object (*e.g.,* `L` at line 3, 5, 13, and 16 in Fig. 1), their enclosing program segments then become competing critical regions (*e.g.,* the execution of line 3 - 5 and line 13 - 16 in Fig. 1 cannot overlap). While traditional lockset analysis also follows the above idea, we note that the accuracy of lockset analysis essentially depends on the underlying alias analysis (*e.g.,* whether different lock/unlock pairs use the same lock object). However, existing lockset analysis usually employs simple alias analysis techniques (*e.g.,* type-based), not to mention the inaccurate handling of cross-entry alias relationships (*e.g.,* imagine that in Fig. 1, `L` at line 3 and 13 might be different pointer variables but aliased to each other on the cross-entry basis).

To improve the accuracy of lockset analysis in the cross-entry scenario, we separate it into two phases: in the per-entry analysis phase, UAFX accurately identifies all the paired lock/unlock within the local entry functions, while in the later UAF validation phase (§III-E), lock objects will be further matched in a cross-entry and on-demand way (*e.g.,* supposing use and free sites are in different entry functions and both enclosed with lock/unlock, UAFX will then check whether they operate on the same lock object). This can be naturally achieved with UAFX's built-in *escape-fetch* based cross-entry alias analysis capability. Similar to points-to records, our lockset analysis is also interprocedural (*e.g.,* lock in a caller while unlock in a callee).

Condition Set/Check. Another important code aspect UAFX analyzes is the path condition set and check, for example, `flag=true` (line 4 in Fig. 1, set before free) and `if(!flag)` (line 14 in Fig. 1, check before use). It is critical to recognize the conflicting condition set and check for UAF prevention (*e.g.,* intentionally put by programmers), otherwise, false alarms will rise. In general, path feasibility can be determined with techniques like symbolic execution, however, full-fledged symbolic execution can be very expensive due to path explosion and complex constraints. Moreover, as we target cross-entry UAFs, a similar problem for "lockset" again occurs here: we have to correctly pair the set/check for aliased conditional variables (*e.g.,* the same global object field that is set/checked via different but aliased pointers) even if they scatter in different entry functions, which is also challenging for existing symbolic executors.

To address these problems, we propose a lightweight "trait analysis" that focuses on a simple but practical subset of path conditions, while being able to perform the cross-entry condition set/check matching. Specifically, UAFX only collects condition checks that compare one variable against one constant (*e.g.,* `a<1`, `p!=nullptr`, *etc.*) and condition set that assigns a constant value to a conditional (*e.g.,* `a=1`). While not complete, it is a practical design choice because many real-world conditions, including those related to UAFs, fit this simple scheme (*e.g.,* a common pattern is that the pointer will be set to *nullptr* upon free and checked before use), as also observed in the previous work [14]. Simple conditions also benefit the solving performance. Regarding the cross-entry matching, our solution here resembles that for "lockset": in the per-entry analysis, UAFX records the checked/modified conditionals within the current entry - note that this is also interprocedural (*e.g.,* condition set/check can scatter in callers and callees), while in the later UAF validation phase, all relevant condition set/check will be cross-entry matched in an on-demand way, we provide more details in §III-D.

Thread Lifecycle Events. It is common to accelerate the computation with multi-threading, however, it also introduces additional complexities for UAF detection, for example, the use and free can happen concurrently in racy threads. As a part

of efforts to support the detection of such concurrent UAFs, during the per-entry analysis, UAFX specifically recognizes and records two types of thread lifecycle events:

(1) Thread creation. These are usually function calls creating child threads, such as `pthread_create()` and `fork()`. UAFX further extracts the information of the created threads from these events, including the thread entry function and its arguments (that may point to certain objects). These thread entry functions will also be individually analyzed and summarized, similarly to the previously identified entry points in §III-B.

(2) Thread join. These events ensure that the child threads exit before a certain program location in the parent thread (*e.g.,* `pthread_join()` and `wait()`). UAFX will pair each thread exit event with the creation event according to the API specification to get a complete picture of a certain thread's lifecycle.

Free and Use Sites. Last but essential, we need to record the free and use sites and their accessed memory objects for further UAF detection. For free sites, UAFX relies on an extensible list of free functions (*e.g.,* `free()`, `kfree()`, `dealloc()`, *etc.*), each function is also configured with detailed parameter information (*e.g.,* which parameter points to the memory object) so that UAFX can record the freed memory objects. For use sites, UAFX records all the memory objects that have ever been accessed and the access locations (*e.g.,* `load`, `store`, function calls like `memcpy()`, *etc.*). We will soon describe how UAFX utilizes this information to identify UAF candidates in §III-D.

Algorithm 1: Cross-Entry Aliases Identification

Data: obj - a memory object
Result: $aliases$ - the set of aliased objects of obj

```
1  getAliases(obj):
2      aliases ← aliases ∪ fwdAliases(obj, ∅)
3      if obj is dummy
4          baliases ← bwdAliases(obj, ∅)
5          for bobj ∈ baliases
6              aliases ← aliases ∪ fwdAliases(bobj,
                ∅)
7      return aliases
8  fwdAliases(obj, cset):
9      for p ∈ escapesTo(obj)
10         for fobj ∈ fetches(p)
11             if fobj ∉ cset
12                 cset ← cset ∪ {fobj}
13                 fwdAliases(fobj, cset)
14     return cset
15 bwdAliases(obj, cset):
16     for p ∈ fetchedBy(obj)
17         for bobj ∈ escapes(p)
18             if bobj ∉ cset
19                 cset ← cset ∪ {bobj}
20                 bwdAliases(bobj, cset)
21     return cset
```

D. Demand-Driven Cross-Entry Alias Analysis

As mentioned in §II, one significant challenge in complex UAF detection is to accurately identify the cross-entry aliased use and free sites (*e.g.,* operate on the same memory objects). The same problem also applies to other relevant code aspects, such as cross-entry aliased lock objects and conditionals (§III-C). One of UAFX's major innovations is to address these problems with an on-demand cross-entry alias analysis, by querying our per-entry function summary generated previously (§III-C). In this section, we first describe how UAFX identifies aliased use and free sites and wraps them into initial UAF candidates (§III-D1), then discuss how we utilize the same backbone cross-entry analysis technique for other UAF-relevant code aspects such as lockset (§III-D2).

1) Use/Free Pairing: UAFX pairs the aliased use/free sites and generates initial UAF candidates in several steps.

Step 1: Obtain the free sites and objects. We start by collecting all the free sites observed during the per-entry analysis (§III-C), as mentioned before, our function summary also includes the objects that get freed. Our idea is to find aliased use sites for the free sites, instead of the opposite, this is because the number of free sites is much less than that of the use sites (*e.g.,* #use is about 17x more than #free in our evaluation).

Step 2: Identify aliased used objects. Given a freed object, UAFX then attempts to find all aliased accessed objects - even in a different entry function. Our algorithm is designed according to Def. 6 (*i.e.,* "Cross-Entry Object Aliases") and sketched in Algorithm 1. Upon receiving an object (obj), `getAliases()` first tries to find all its *forward* aliased objects (see Def. 6) with $fwdAliases()$ at line 2, this covers Case 1 and 2 in Def. 6. If obj is a dummy one (Def. 2), we need to further consider Case 3 in Def. 6, because only dummy objects could serve as *fetched* objects at the right end of the *EFP* (*i.e.,* $Dst(EFP)$ in Case 3). To calculate the Case 3 aliases, we first obtain all the *backward* aliases of obj at line 4, then for each *backward* alias, we collect all its *forward* aliases (line 5 - 6), which will be the Case 3 aliases of obj since they share the same EFP origin. The calculation of *forward* and *backward* aliases is relatively straightforward. Basically, we construct all possible EFPs (either direction) by recursively enumerating and connecting EFP segments. Line 8 - 21 in Algorithm 1 show the simplified logic with some details omitted (*e.g.,* configurable threshold of EFP length, which we currently set to 3 as the UAFs we have analyzed do not surpass it), where $escapesTo(obj)$ and $fetchedBy(obj)$ return the pointers to which the obj escapes and from which it is fetched, respectively. Similarly, $fetches(p)$ and $escapes(p)$ return the set of objects fetched by and escape to p, respectively. All

the above information is available in our per-entry function summaries, as described in §III-C.

Step 3: Generate UAF candidates. The previous step has generated (cross-entry) pairs of aliased freed/used objects. In this step, we wrap every aliased object pair, along with the detailed EFP establishing their alias relationship and the free/use sites with full calling context information, into a candidate UAF case. These cases will be further examined in the next phase (§III-E).

2) Support other Relevant Code Aspects.: Besides the essential use/free alias relationship, a UAF detector needs to consider a richer set of program semantics as mentioned in §III-C, many of which need cross-entry alias reasoning as well. Fortunately, UAFX's cross-entry analysis capability is versatile enough to suit these needs. We specifically support two additional types of objects/variables in our prototype: (1) lock objects, and (2) conditional variables, as both of them may be shared across entry functions and it is critical to know whether different occurrences are aliased or not (*e.g.,* whether two critical regions compete). The basic algorithm is the same as in §III-D1 but with different memory objects to start with (*e.g.,* from freed object to the lock object).

Importantly, our cross-entry alias analysis is fully demand-driven, meaning that it is only invoked when necessary for selected target objects (*e.g.,* the freed ones, the lock objects relevant to the UAF case, *etc.*) instead of all. On the other hand, even though UAFX is capable of constructing cross-entry EFPs, each entry function only needs to be analyzed once (§III-C) thanks to UAFX's summary-based reasoning. These design choices boost UAFX's efficiency while still enabling effective cross-entry UAF detection.

E. Systematic UAF Validation

The UAF candidate generation in the last phase is merely based on the alias relationship between the free and use, however, as mentioned in §II, to validate that the candidate is a real UAF there are many more aspects to be considered (*e.g.,* locks, path conditions, *etc.*). To comprehensively take multiple UAF-relevant aspects into account, our major insight is that these aspects could be uniformly translated into partial-order constraints that restrict the relative execution orders of different program statements. A UAF candidate in §III-D can be a real vulnerability iff all its partial-order constraints are satisfiable simultaneously. To fulfill this idea, UAFX first collects all key program statements associated with the considered code aspects (*e.g.,* lock/unlock statements for the "lockset" aspect), all these statements are then uniformly numbered. Next, based on the characteristics of each aspect, UAFX infers the required relative execution order between different key statements and encodes them into partial-order constraints with the statement numbers. For example, if the free site is numbered as 0 and the use as 1, UAFX generates the constraint $0 < 1$ (Def. 7) to enforce that the use must happen after free (*i.e.,* UAF definition). In the remaining of this section, we first describe all our considered code aspects and how they are translated into unified partial-order constraints, then discuss how UAFX can automatically find all aspect-related program statements that are relevant to a specific UAF candidate.

1) Code Aspects: UAFX considers the following code aspects for UAF validation.

Use and Free Sites. As mentioned, The UAF definition decides that the use must happen after free to illustrate the vulnerability. UAFX accordingly generates the partial-order constraint between the free and use statements (*e.g.,* Fig. 1, $L6 < L15$).

Lockset. Two critical regions protected by the same lock can only be sequentially executed. UAFX identifies the lock and unlock statements (*e.g.,* invocation of lock/unlock functions) of both regions and makes a constraint that one region's lock can only be executed after the other's unlock (*e.g.,* $L16 < L3||L5 < L13$ in Fig. 1).

Path Conditions. If passing a conditional check is necessary to trigger a UAF (*e.g.,* the use site is guarded by the check), then any assignment statement that could violate the condition must not be executed before the condition check (without other assignments for the same conditional in between), otherwise, the UAF will be infeasible. UAFX captures this kind of constraint by first locating the critical condition check that enables the UAF (see §III-E2 and §IV for more details) and the condition set that could affect the check (*e.g.,* enable or "kill"), and then generating the partial-order constraint to ensure the condition check will not inevitably fail (*e.g.,* $L14 < L4$ in Fig. 1).

Notably, UAFX supports reasoning about the condition set/check at the *bit-level* (*e.g.,* `if (v & 1)`, `v |= 1`), which is frequently used in some programs (*e.g.,* Linux kernel) for memory economy.

Escape-Fetch Data Flow. The cross-entry alias relationship (*e.g.,* between the use and free sites) is established by the *escape-fetch* path (or EFP), which is also essential for the UAF validity. UAFX ensures the feasibility of the EFP by enforcing that for each EFP segment (in the form of $obj0 \rightarrow p \rightarrow obj1$, see Def. 5), the *escape* ($obj0 \rightarrow p$) must happen before *fetch* ($p \rightarrow obj1$), without other assignments to p in between. Otherwise, the fetched $obj1$ may not be aliased to $obj0$, invalidating the whole EFP. The example partial-order constraint belonging to this category in Fig. 1 is $L2 < L10 < L15$.

Thread Lifecycle. Given the thread lifecycle events recognized in the per-entry analysis (§III-C), UAFX generates partial-order constraints specifying that all statements of a child thread can only execute after the thread's creation site (*e.g.,* `pthread_create()`), but before its termination site (*e.g.,* `pthread_join()`).

Intra-Entry Sequential Order. The term "cross-entry" is orthogonal to "multi-threaded" or "concurrency", as the former merely refers to the fact that the execution needs to invoke multiple entry functions, either in one thread or multiple. However, in either case, statements executed during an entry function invocation should follow the natural sequential order (micro-architecture level features like out-of-order execution

are out of our scope). For example, in Fig. 1, when invoking `entry0()` line 3 should always execute after line 2 based on their programmed order.

UAFX gathers key program statements associated with the above code aspects (*e.g.,* lock/unlock function invocations, use/free sites, *etc.* as aforementioned), and then arranges them into different entry function invocations. For those belonging to the same invocation, natural sequential order will be encoded into partial-order constraints based on their relative positions in the function (*e.g.,* $L1 < L2 < ... < L6$ in Fig. 1). More formally, we infer this order from the topological structure of the control flow graph (CFG), if in the CFG the statement A reaches B but not otherwise, we say A must execute before B in the same invocation. If A and B are mutually reachable (*e.g.,* loop), to be conservative we will not enforce the sequential order constraint between them, which may result in false alarms for fewer false negatives. We leave the better loop handling as future work.

All the above partial-order constraints make a system of inequalities (as shown in Fig. 1), which can be solved by an SMT solver (*e.g.,* z3 [13]). The UAF is only feasible when there is a solution to the system, in that case, UAFX will issue a UAF warning with the detailed execution sequence to trigger it (*i.e.,* the solution generated by the solver). Otherwise, the UAF candidate is discarded as a false alarm (*e.g.,* no solution for use site 0 in Fig. 1). Our approach has multiple benefits, including comprehensiveness (*e.g.,* support accurate aliasing across entries), efficiency (*e.g.,* SMT solvers are highly optimized for such problems), and extensibility (*e.g.,* more code aspects can be easily integrated).

2) Identify Relevant Program Statements: Given all the code aspects to consider (§III-E1), there will be many associated statements in the target program, however, to validate a specific UAF candidate, we usually only need to consider a small portion (*e.g.,* not all condition set/check matter for a certain UAF) - blindly including all statements is neither necessary nor efficient. The problem is then how we can automatically locate all relevant statements.

UAFX employs an iterative procedure to gradually discover UAF-specific key statements. The initial statement set only includes the use and free sites, then in the first iteration, all statements directly affecting the use and free sites (*e.g.,* a critical condition check guarding the use site, or surrounding lock/unlock pairs) are identified and added to the set, then in the second iteration, UAFX recursively identifies and adds the statements affecting the newly added ones in the last iteration, such iterations are repeated until no new statements can be discovered, or a configurable iteration count threshold is reached (empirically set to 3 that suffices for UAFs we have analyzed). This approach enables us to progressively include only the relevant statements to the UAF case in question, improving the validation efficiency while still comprehensively considering different code aspects listed in §III-E1.

IV. Implementation

We implement a prototype of UAFX on top of SUTURE [14], an open-source static analyzer for C programs. UAFX utilizes SUTURE's high-precision pointer analysis for LLVM IR [15], but with substantial improvements regarding *escape-fetch* based cross-entry alias analysis and systematic UAF validation (§III). Compared to SUTURE, UAFX has 12,850 lines of new code addition and 3,838 lines of deletion. UAFX will also be open-sourced to benefit the community. In the remainder of this section, we cover some noteworthy implementation details of UAFX.

Recognize Critical Condition Checks. For a specific program statement (*e.g.,* the use site), a conditional check is *critical* if it can determine whether that statement can be reached (*e.g.,* in Fig. 3, line 3 is a critical check for line 4, but not for line 7). When validating a UAF candidate, UAFX only cares about the *critical* condition checks of key program statements (§III-E2). To identify all the critical conditional checks for a certain statement, UAFX verifies whether one subsequent branch of the conditional check dominates the target statement on the CFG, interprocedurally.

```
1  char *g = ...;
2  int foo(char **p) {
3    if (...) {
4      *p = g; //FETCH
           & ESCAPE
5      return 0;
6    }
7    return 1;
8  }
```

```
9  void entry0() {
10   free(g); //FREE
11 }
12 void entry1() {
13   char *p =
         malloc(8);
14   int r = foo(&p);
15   if (r) *p; //USE
16 }
```

Fig. 3: Example of Condition Check on Return Values

Condition Check on Return Values. A specific (and common) condition check targets the return value of a callee. We show an example in Fig. 3. Assuming that `entry0()` and `entry1()` cannot interleave with each other (*e.g.,* protected by the same lock), in that case, a less precise analysis may conclude that the sequential invocation of `entry0()` and `entry1()` leads to a UAF, because after the free site at line 10 within `entry0()`, the freed object could be *fetched* by `g` and *escapes* to `p` in the callee `foo()` of `entry1()` (line 4), and then accessed later in `entry1()` at line 15. However, the UAF is actually impossible because as long as the *fetch & escape* happens at line 4, `foo()` will return 0, failing the condition check at line 15 and preventing the use site from being reached.

UAFX can correctly filter out such potential false alarms by comprehensively reasoning about multiple code aspects, including the condition check on the return value. For a conditional return value check, UAFX will dive into the callee code and figure out the locations within the callee that once reached, will inevitably lead to unsatisfying return values (*e.g.,* once line 4 is reached, the condition check at line 15 will for sure fail). Such information is then naturally integrated into our partial-order constraint system. For example, in Fig. 3,

as mentioned, line 4 must *not* be executed before line 15 to trigger the use site (*i.e.,* $L15 < L4$), but on the other hand, as the *fetch & escape* node, line 4 must happen before line 15 (*i.e.,* $L4 < L15$), immediately resulting in an unsatisfiable constraint system that invalidates the seemingly UAF case.

Function Modeling. UAFX relies on well-defined programming interfaces to extract program statements related to different code aspects, as mentioned in §III-E1. For example, the memory management interface (*e.g.,* `malloc()`, `free()`), locking primitives (*e.g.,* `mutex_lock/unlock()`), and thread lifecycle management APIs (*e.g.,* `pthread_create()`). UAFX models the semantics of these common interfaces and APIs - a standard static analysis practice. Unseen interfaces/APIs can be easily and flexibly supported by UAFX - we consider it orthogonal to our core methodology (§III).

Intentionally Unsupported Features and Constructs. There are some well-known challenging code features and constructs for static analysis (*e.g.,* recursive data structures like linked lists), for which accurate and efficient static reasoning is very difficult (*e.g.,* differentiating elements on a linked list). As a result, the common practice is to conservatively handle these features (*e.g.,* treat all list elements as aliased), however, it tends to cause lots of false alarms (*e.g.,* the free and use sites are for different list elements but wrongly treated as aliased). To mitigate this problem, UAFX first identifies UAF candidates (§III-D1) involving below features:

(1) Recursive Data Structures. UAFX will decide whether the freed/used object is an element of a recursive data structure (*e.g.,* linked lists), by inspecting whether the object pointer is loaded from another same-typed object.

(2) Reference Count Mechanisms. Reference counting is also a well-known challenge for static analyzers, as it is difficult to accurately track all the counter changes happening in different conditions and scattered at different places. UAFX recognizes the existence of reference count mechanisms in the UAF candidate, by looking at whether the free site is preceded by specific function calls with indicating names such as `xxx_put()`.

We intentionally drop UAF candidates involving the above features without further review to avoid excessive false alarms, similar to previous works [4]. This will lead to false negatives, which we try to quantify in §V-B.

V. Evaluation

Dataset. To evaluate the efficacy and efficiency of UAFX regarding cross-entry UAF discovery, we compile and extract 34 driver modules from the Linux kernel v5.17.11 as listed in Table I, all in the LLVM bitcode format suitable for UAFX's analysis. We additionally include *lrzip* [16] - a popular C program with known UAF issues according to a previous study [17] - in our test to demonstrate that UAFX could be applied to user-space C programs. To study whether UAFX can re-discover known UAF bugs and facilitate its comparison with other tools, we also collect the fixed UAF vulnerabilities found by Syzkaller [11] in the Linux kernel device drivers (publicly available on the Syzbot dashboard [18]), within the past three years of April 2024. This results in a collection of 23 UAF cases as listed in Table III. We choose this dataset because: (1) Syzkaller [11] is the most popular 24/7 Linux kernel fuzzer maintained by Google - one major source of the newly discovered kernel vulnerabilities in recent years, and (2) Syzkaller reports typically come with detailed bug-triggering PoC code and call stack traces, enabling us to reliably locate

No.*	Path	Bitcode Size	Time**
0	drivers/acpi	17 MB	3.2/2.4
1	drivers/atm	5.9 MB	0.06/0.01
2	drivers/bluetooth	9.8 MB	0.02/0.08
3	drivers/char	13 MB	0.8/0.6
4	drivers/comedi	21 MB	0.4/7.9
5	drivers/firewire	2.2 MB	0.08/0.05
6	drivers/firmware	8 MB	0.01/0.01
7	drivers/gpu/drm/i915	102 MB	33.5/0.3
8	drivers/i2c	23 MB	1/0.03
9	drivers/isdn	8.8 MB	0.04/0.01
10	drivers/mailbox	2 MB	0.01/0.01
11	drivers/md	24 MB	5.2/4.6
12 - 14	drivers/misc	27 MB	0.01/0.01 0.01/0.01 0.2/0.01
15	drivers/mmc	21 MB	8.1/0.01
16 - 17	drivers/mtd	29 MB	12.3/0.01 4.3/2.1
18	drivers/net/ppp	2.1 MB	4.1/0.01
19	drivers/platform/surface	3 MB	0.01/0.01
20	drivers/platform/x86	19 MB	0.01/0.01
21	drivers/scsi	121 MB	1.2/1.2
22	drivers/staging	75 MB	0.2/0.01
23	drivers/tee	746 KB	0.01/0.01
24	drivers/thermal	8.9 MB	0.01/0.01
25 - 26	drivers/tty	24 MB	4.7/0.9 0.03/3.5
27	drivers/usb/gadget	19 MB	8/1.5
28	drivers/usb/misc	4.4 MB	0.3/3.1
29	drivers/vdpa	3.3 MB	0.01/0.01
30	drivers/vfio	3.3 MB	0.17/0.01
31	drivers/vhost	2.1 MB	0.08/0.01
32 - 33	drivers/video	41 MB	3.8/0.01 0.3/5.2

*: The same path may contain multiple modules (*e.g.,* related to different sub-devices or vendors), which share the bitcode file.
**: per-entry analysis time / UAF detection time (hrs)

TABLE I: The List of Tested Kernel Modules

Subject	#Warning	#TP	$\#FP_r^0$	Pre_r^0
Linux Kernel v5.17.11	977	78	101	43.6%
lrzip	18	2	3	40.0%

0: FP_r: reviewer-perceived false alarms; Pre_r: #TP/(#TP+#FP_r)

TABLE II: Vulnerability Discovery Precision of UAFX

```
00 kfree(dev->private); // FREE
01 dev->driver = NULL; // SET (missed by UAFX)
02 ......
03 if (dev->driver) // CHECK
04   // USE in multiple different indirect callees
05   dev->driver->detach(dev);
```

Fig. 4: Example False Alarms with the Same Root Cause

the UAF root cause (*e.g.,* use/free sites and alias relationship) for further analysis.

Hardware Configuration. Our evaluations are conducted on a server with Intel Xeon Gold 6248 CPU @ 2.5 GHz and 1 TB RAM.

A. Precision

We report UAFX's vulnerability discovery precision *perceived by the warning reviewers* (two paper authors) in Table II. Specifically, when reviewing the warnings generated by UAFX, we adopt the following procedure:

Step 1. Pick one warning and investigate its validity, then repeat this step if it is a true positive. Otherwise, go to *Step 2*.

Step 2. The current warning is a false alarm, we then utilize a simple string match to automatically filter out all the other similar false alarms from the warning pool.

With the above procedure, FP_r denotes the false alarms that actually need the reviewers' inspection (*i.e.,* reviewer-perceived false alarms). While the number of raw false alarms (*i.e.,* #Warnings - #TP in Table II, where #Warnings are unfiltered) is much larger than FP_r, many of them are highly similar and can be *automatically* excluded in *Step 2*. This is mainly because the same false alarm root cause, expressed in a simple string pattern, is associated with multiple warnings. Specifically, (1) UAFX is context-sensitive, so the same use/free sites may be reached in many different calling contexts, resulting in multiple warnings. However, the false alarm root cause can be context-insensitive - identifiable in multiple warnings. (2) Even false alarms with different use/free sites can share the same root cause. We show a concrete example in Fig. 4, as seen, UAFX incorrectly identifies the case as a sequential UAF because it fails to recognize the condition set at line 1, which is hidden in a low-level bulk `memset()` due to the compiler optimization. As a result, UAFX fires 33 false alarms because there are many different use sites in different indirect callees at line 5. However, once the root cause is understood from one warning, all false warnings in our pool involving line 0 and line 5 can be immediately filtered out with a simple string match (*e.g.,* based on line numbers and calling contexts). While partial-order based filtering is also feasible here, we choose string-based method due to its simplicity and effectiveness. Following this procedure, inspecting the warnings takes the two reviewers about one week.

The aforementioned false alarm reduction technique is in the same spirit as the clustering-based FP mitigation [19], helping UAFX achieve a reviewer-perceived precision of more than 40% (Pre_r in Table II), comparable to many existing static bug-finding works including those for the Linux kernel [20], [21], [22], [23], [24], [25], [26], [27], [14], [28], [29], [30]. While we consider it necessary to improve UAFX's precision further, we believe UAFX's enhanced capability of cross-entry UAF detection is still valuable (*e.g.,* according to a recent study [31], some developers are more sensitive to missed vulnerabilities than lower precision).

False Alarm Analysis. Despite our various efforts to reduce false alarms (§III-E and §IV), they still constitute the major portion of UAFX's generated warnings - a well-known problem for static analyzers. We briefly discuss the significant reasons behind the false alarms. It is worth noting that UAFX is built on top of SUTURE [14], so some limitations of the latter are inherited.

(1) Limited Path-Sensitivity. UAFX recognizes conflicting path condition set and checks of simple yet common forms (§III-C), however, it is not fully path-sensitive. Consequently, false alarms arise due to their complex sanity check or overestimated alias sets due to path-insensitivity, accounting for about 35% of UAFX's false positives.

(2) Inaccurate Indirect Call Resolution. Linux kernel is rich in indirect function calls, partially because it simulates many object-oriented programming features (*e.g.,* polymorphism) with C structs and function pointers, posing great challenges for static analyzers. Though UAFX utilizes the state-of-the-art approaches for indirect call resolution [23], [32], inaccuracies remain in the result, contributing to about 10% of UAFX's false alarms.

(3) Implementation Imperfection. Implementation flaws are inevitable in static analyzers, even mature ones [33], due to the complexity of real-world code. This is especially true for UAFX that needs to reason about various code aspects (§III-C). These imperfections follow a long-tail distribution and lead to about 55% of UAFX's false alarms. Some significant flaws include unrecognized condition set/check (*e.g.,* the example in Fig. 4), lock/unlock primitives not properly modeled, and FPs involving recursive structures or reference counting but not captured by our heuristic-based filters (§IV). As a research prototype, UAFX focuses on validating the idea of cross-entry UAF detection and we plan to improve the implementation to unleash its potential better.

Effectiveness of Systematic FP Filtering. To understand the performance of our partial-order-based systematic FP filtering (§III-E), we further calculate how many UAF candidates gen-

erated in §III-D can be filtered out by it. The median filtering-out ratio among our tested code subjects is 98.5%, while more than 80% subjects have a ratio higher than 95%. Due to the large volume of UAF candidates, we randomly investigate 100 filtered-out candidates and confirm that all are indeed false alarms (though not observed here, we acknowledge that our filtering can cause false negatives, as detailed in §V-B). These results suggest that our systematic FP filtering has effectively eliminated a major portion of false alarms.

B. Recall of Known UAF bugs

Type[0]	Fix Commit	Detectable? Y: Yes, N (Reason): No		
		UAFX [1]	DCUAF[2]	Canary[3]
SEQ	ed9605a66b62	N (U0)	N (D0)	N (C0)
	dd613a4e45f8	Y	N (D0)	N (C0)
	68035c80e129	N (U0)	N?[4]	N?[4]
	84b01721e804	Y	Y	Y
	5f0b5f4d50fa	N (U1)	N?[4]	N (C0)
	24013314be6e	N (U1)	N (D0)	N (C0)
CRS	b436acd1cf7f	Y	N (D1)	N (C0)
	5c15c60e7be6	Y	N (D0)	N (C0)
	6f9c4d8c468c	Y	N (D0)	N (C0)
	804ca14d04df	Y	Y	N (C0)
	7d21e0b1b41b	Y	N (D0)	N (C0)
	f80cfe2f2658	N (U0)	N (D0)	N (C0)
	0a0b79ea55de	Y	N (D0)	N (C0)
	ded85b0c0edd	Y	N (D0)	N (C0)
	36e8169ec973	Y	Y	N (C0)
	3c4f8333b582	Y	N (D1)	N (C0)
	c052cc1a069c	Y	Y	N (C0)
	e9e6aa51b273	Y	N (D0)	N (C0)
	d18dcfe9860e	N (U1)	Y	N (C0)
	f326ea63ecc6	N (U1)	N (D1)	N (C0)
	2191c00855b0	N (U1)	N (D0)	N (C0)
	b6702a942a06	Y	N (D0)	N (C0)
	0ac4827f78c7	Y	N (D0)	N (C0)
SUM	23	15 Y (65.2%)	5 Y, 2 N? (30.4%)	1 Y, 1 N? (8.7%)

0: SEQ: single-entry sequential UAF; CRS: cross-entry UAF
1: U0: Aggressive FP filtering - Recursive Structures
U1: Aggressive FP filtering - Reference Counting
2: D0: Complicated Alias Relationship; D1: Oversimplified FP Filtering
3: C0: Unsupported Control-Flow Model;
4: N?: Unsure but likely not detectable due to subtle linked list operations.

TABLE III: Inspection of Syzkaller Reported UAFs in Linux Device Drivers within 3 years of Apr 2024

It is challenging to evaluate the false negative rate of UAFX due to the lack of ground truth. Furthermore, there are few established benchmarks for cross-entry UAFs (*e.g.,* the widely-used Juliet vulnerability dataset [34] only contains basic sequential UAF cases). To better understand UAFX's capability, as aforementioned, we crawl Syzkaller-reported UAF cases in Linux kernel device drivers in the last three years, as listed in Table III.

Most UAFs in this collection are cross-entry (*i.e.,* 17 out of 23), meaning that use and free happen in different entry function invocations. Even for the remaining 6 seemingly sequential cases where use happens straightly after free, most (*e.g.,* 5 out of 6) still require a separate entry function invocation to establish the alias relationship between the use/free sites, or to (maliciously) modify the reference count. These observations highlight the importance of detecting cross-entry UAFs which have become dominant. In Table III, UAFX recognizes 15 out of 23 benchmark UAFs, a promising result given the substantial difficulties of statically discovering cross-entry UAFs while striking a balance between coverage, precision, and efficiency. Next, we discuss the major reasons for UAFX's false negatives.

False Negative Analysis. First, our intentional drop of warnings involving difficult code features (§IV) causes FNs:

Reference counting. Refcnt is a well-known difficulty in UAF analysis, existing works tackle this problem with various heuristics and compromises (*e.g.,* limit the scope to single entry functions) in static reasoning [35], [27], [36], [37], or target at dynamic traces [37] or fuzzing [38]. Accurate and comprehensive static refcnt reasoning remains challenging. To quantitatively understand the impact of our drop of refcnt-related warnings on FNs, we inspect the historical kernel UAFs to see how many involve refcnt: (1) in Table III, we identify 5 of 23 (21.7%) Syzkaller UAFs involving refcnt (*U1* in Table III) and are missed by UAFX; (2) we further go through Linux kernel UAF CVEs within three years of Oct 2024 - 315 in total, and check their fix commits (including the commit messages) to determine whether refcnt is involved, we find that about 23.5% are refcnt-related. A larger-scope syzbot UAF survey by a previous work [38] reports about 36.1% of UAFs related to refcnt.

Recursive structures. Recursive data structures pose significant challenges for static analyzers, while few studies explore their impact on UAF detection. In Table III, we find 3 of 23 (13%) Syzkaller UAFs missed by UAFX due to recursive structures (*U0*). Our inspection of the same three-year CVE dataset as in *reference counting* reveals that about 17.8% of them are related to recursive structures.

Combining the two code features (overlaps exist), about 38.7% of our three-year CVE dataset is affected. These numbers confirm that these difficult code features are considerable kernel UAF causes, while our current prototype focuses on cross-entry UAF reasoning and is limited in refcnt and recursive structure analysis, we believe the aforementioned dedicated works (*e.g.,* on refcnt reasoning) can be complementary and leave better support for these features as future works.

Besides, other FN-inducing limitations of UAFX include: (1) UAFX only analyzes each loop once and limits the depth of the calling stack for better performance and fewer FPs, which are common practices in static bug finding (*e.g.,* soundy analysis [39], [40]), and (2) inaccuracies mentioned in §V-A could also generally lead to FNs.

C. Efficiency

As mentioned, we run UAFX on 34 kernel driver modules and 1 user-space program. 16 of these subjects take more than 1 hour to analyze, we show their detailed time breakdown in Fig. 5 (full statistics can be found in Table I). Specifically, the time is spent on two stages: per-entry analysis (§III-C) and UAF detection with demand-driven cross-entry analysis (§III-D and §III-E). As seen, most modules are scanned within 10 hours, however, certain challenging modules can take up to more than 30 hours, mainly because of their large amount of cross-entry use and free sites to be reasoned

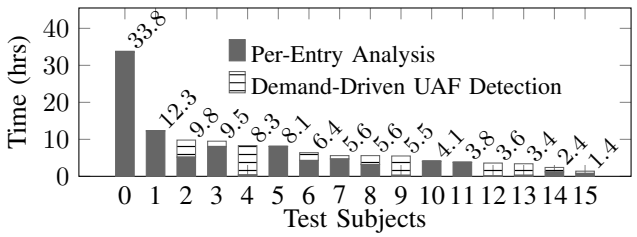

Fig. 5: UAFX Time Breakdown for Test Subjects (>1hr)

about. We note that (1) UAFX is more expensive than many existing tools since it needs to perform the advanced cross-entry analysis accurately, (2) UAFX utilizes a single thread to process each subject, so multiple subjects can be naturally paralleled, making an acceptable overall efficiency on typical multi-core platforms. A further optimization is to re-implement UAFX's per-entry analysis in the bottom-up style (*e.g.,* analyze and summarize every single callee only once), which could improve efficiency by 50% - 90% according to our estimation. Due to the significant engineering challenges and efforts, we leave it as a future work.

Regarding the time distribution between different stages, we observe no clear patterns as expected. Because the cost of per-entry analysis (§III-C) mainly relies on the code size of the subject, while that of the UAF detection stage (§III-D and §III-E) relies on the number of cross-entry use/free pairs - they are not necessarily correlated.

In terms of memory cost, UAFX has a moderate footprint. Among all the software subjects evaluated, the median of the average (during the whole execution) memory usage is 214 MB, while the peak usage reaches 2423 MB, observed when analyzing a large kernel GPU driver. We thus consider UAFX's memory efficiency reasonable.

D. Comparison with Existing Tools

Given the difficulties of cross-entry UAF detection, few existing static detectors are available for comparative evaluation. Most existing static UAF detectors are only capable of identifying simple sequential UAF cases (*e.g., use* happens straightly after *free* on the CFG), sometimes even limited to the intraprocedural setting, for example, CRED [4], Palfrey [5], Clang Static Analyzer [6], Infer [7], Cppcheck [8], and Flawfinder [9]. To our best knowledge, DCUAF [10] and Canary [12] are only tools supporting cross-entry UAF detection to a certain degree, unfortunately, neither is available for use and testing (we plan to open source UAFX). Given the situation, we start with an in-depth design-level analysis of DCUAF [10] and Canary [12], then investigate their capabilities of detecting our collected benchmark UAF cases in Table III, that is also used in UAFX's false negative evaluation (§V-B). We will also discuss other aspects including precision and efficiency.

Design Analysis of DCUAF [10]. DCUAF aims to discover concurrent UAFs in Linux kernel device drivers (*e.g., free* and *use* in different driver entry functions). It focuses on automatically identifying pairs of driver interface functions that could be potentially executed concurrently, thus prone to concurrent UAFs. To achieve this, DCUAF develops a *local-global* strategy that relies on lock-based heuristics (*e.g.,* concurrent functions likely use the same locks) and file number statistics (*e.g.,* concurrent functions are likely to be invoked by the same source file). Besides the function pair identification, DCUAF employs a coarse-grained type- and field-based pointer analysis to recognize aliased *free* and *use* sites, and filters false alarms mainly by single-aspect lockset analysis (*e.g.,* whether the U/F sites are protected by the same lock), as seen in Section 3.1, 3.2, and 5.4 in [10]. As mentioned (§II-A), coarse-grained alias analysis and simple FP filtering can cause both false positives and negatives. To address these issues, UAFX develops sophisticated cross-entry alias analysis (§III-D) and systematic FP filtering (§III-E). We also believe that DCUAF's concurrent function pair identification technique can be complementary to UAFX.

Design Analysis of Canary [12]. Canary is capable of finding inter-thread UAF issues (*e.g., use* happens in a `fork`'ed child thread) with static inter-thread pointer analysis. Its technique is scoped to the *fork*-based thread model (Section 2 in [12]), where the inter-thread alias relationship is explicitly established via the pointer argument of `fork()`. However, cross-entry UAFs, especially in the Linux kernel, go beyond `fork`-based model, for example, the general global variable-based cross-entry aliases and `pthread`-based thread model (*e.g.,* see our case studies in Fig. 6 and Fig. 7). Compared to Canary, UAFX handles different cross-entry UAF cases with advanced *escape-fetch*-based cross-entry alias analysis (§III-D) and extensible thread lifecycle modeling (§III-C). As a result, UAFX can identify more cross-entry UAFs (*e.g.,* cases in §V-E). Canary also employs partial-order constraints to filter out false alarms, however, it only considers simple code aspects including natural intra-thread order and thread creation and termination, on the other hand, UAFX innovatively encodes a much richer set of essential code aspects such as lock primitives and condition set/check (that could happen in different entry functions), as described in §III-E1.

Comparison on Bug-Finding Capability. Based on our deep understanding of both the existing tools and the collected UAF cases in Table III, we investigate whether DCUAF [10] and Canary [12] are capable of detecting these benchmark UAF cases. We assume that entry functions involved in these UAFs are given to all three tools, and the implementation of DCUAF and Canary is flawless (*e.g.,* all lock/unlock could be correctly identified and paired, as well as new thread creation). For DCUAF, we consider a UAF case undetectable if (1) the use and free sites have a complicated alias relationship that is beyond the capability of its simple field-based alias analysis (D0 in Table III), and (2) if the use and free sites are protected by the same lock, which will be wrongly filtered out by DCUAF's oversimplified FP elimination logic (D1 in Table III). For Canary, we deem a UAF unrecognizable if the bug-triggering control and data flows are outside of its `fork()`-based multi-threading model - most kernel UAF cases belong to this

unsupported category by Canary (C0 in Table III). We test UAFX on relevant functions and code snippets involved in these collected UAFs. As shown in Table III, UAFX shows the best bug-finding capability (*e.g.,* a 65.2% recall, while the second-best is 30.4%) thanks to its advanced *escape-fetch*-based alias analysis and more systematic FP filtering. We note that Canary's limited recall in Table III is mainly because it is not specifically designed for the kernel concurrency model (*e.g.,* multiple entry functions operating on shared variables).

Comparison on Precision and Efficiency. As reported in their original papers, both DCUAF [10] and Canary [12] have better precision (*e.g.,* Claimed TPR of DCUAF: 94.3%, of Canary: 73.3%) and efficiency (*e.g.,* DCUAF: less than 20 minutes, Canary: 4.67 hours) than UAFX. However, as previously mentioned in §V-A, UAFX targets complex cross-entry UAFs which inevitably incurs more false alarms and requires more computational resources. We believe our tool represents a novel design in the tradeoff space that favors covering more subtle bugs (a top priority of many developers and security practitioners [31]) with a reasonably worsened precision and increased cost.

Summary. In conclusion, UAFX provides a significantly enhanced bug-finding capability for challenging cross-entry UAFs, compared to existing UAF detectors, UAFX excels at identifying subtle, real-world UAFs, as evidenced in Table III. While existing UAF detectors could achieve better precision and efficiency due to simpler problem scopes, we believe UAFX is a valuable tool given its advanced capability and reasonable performance (on par with many static kernel bug-finding works discussed in §V-A).

E. Study of the Found UAF Issues

We have reported the identified true positive warnings by UAFX (Table. II) to relevant developers or maintainers, so far, 37 warnings have been confirmed by developers or bug-triggering PoCs, which are related to 10 independent UAF bugs.

Characteristics of the Found UAFs. We first collect the metrics related to different code aspects (§III-E) for an overview of our found UAFs. *(1) EFP Length.* Interestingly, all true positives UAFX identified are cross-entry, involving multiple racy threads and *EFP*s (*i.e.,* a single entry sequential UAF does not have any *EFP* as there is no cross-entry object escape and fetch). 24% of our TP warnings have 1 *EF Segment* (Def. 5) and the others have 2, *(2) Locks.* 96% of TP cases involve the reasoning of at least 1 pair of lock/unlock (among them: 1 pair: 44%, 2 pairs: 13%, 3 pairs: 43%), *(3) Condition Set/Check.* On average, each TP case needs to reason about 14.8 condition checks and 2.2 condition set. We believe these metrics demonstrate UAFX's capability of discovering cross-entry UAF issues that require careful reasoning of multiple code aspects.

Upon successful exploitation, the found UAFs could lead to various security consequences such as denial of service and information leakage. In the remainder of this section, we showcase some representative UAF cases found by UAFX.

```
00 void md_unregister_thread(struct md_thread **threadp)
01 {
02  struct md_thread *thread = *threadp;
03  if (!thread)
04        return;
05  pr_debug("... %d\n", task_pid_nr(thread->tsk)); //USE
06  spin_lock(&pers_lock);
07  *threadp = NULL;
08  spin_unlock(&pers_lock);
09  kthread_stop(thread->tsk); //USE
10  kfree(thread); //FREE
11  }

12 static void mddev_detach(struct mddev *mddev)
13 {
14  ...
15  md_unregister_thread(&mddev->thread);
16  ...
17 }

dm_ctl_ioctl() -> dev_suspend() -> do_resume() ->
dm_table_destroy() -> raid_dtr() -> md_stop() -> __md_stop() -
> mddev_detach()
```

Fig. 6: UAF Case Study 1 (in Linux Kernel)

Case 1. Fig. 6 shows a cross-entry UAF case in the Linux kernel. To trigger the bug, the function `md_unregister_thread()` is concurrently executed in two threads, while one accesses the freed memory (line 5 and 9) by the other (line 10), due to the seemingly correct but actually unsafe thread synchronization. We highlight the complexities of this case and show UAFX's strengths as follows.

(1) Cross-Entry Control Flow. At first glance, Fig. 6 seems to be a normal *free-after-use*, likely to be missed by detectors that only support simple sequential UAF cases. However, cross-entry UAF is actually possible when competing threads invoke `md_unregister_thread()` multiple times. UAFX is aware of such cross-entry control flows at its core design and does not blindly exclude the cases like Fig. 6 simply based on the use/free positioning on the CFG.

(2) Cross-Entry Data Flow. To identify this issue it is also critical to determine that the local pointers `thread` held by both threads are aliased, which is a non-trivial task as the analyzer needs to track the pointer propagation (via both top-level variables and address-taken memory objects) through the whole call chain as shown at the bottom of Fig. 6. Eventually, UAFX concludes that the object pointed to by `thread` (in both threads) comes from the same global source (*e.g.,* `mddev` at line 12), establishing the alias relationship between the use and free sites.

(3) Multi-Aspect Validation. Notably, the use and free sites in Fig. 6 are guarded by both condition set/check (line 3 and 7) and lock primitives (line 6 and 8) - seemingly strong protection that could confuse the static analyzers with simple FP filtering logic, and even human analysts. However, UAFX's systematic multi-aspect reasoning can still figure out the clear path toward

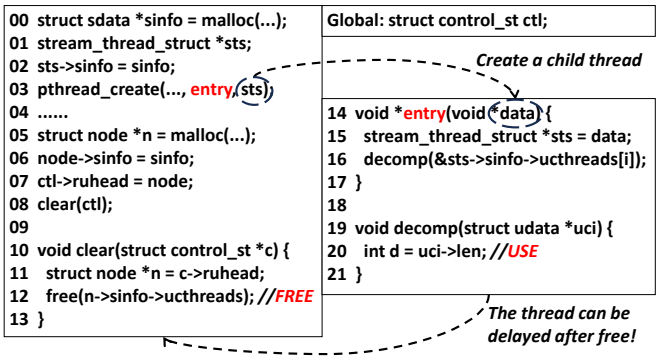

Fig. 7: UAF Case Study 2 (in lrzip)

the UAF by partial-order constraint solving. Specifically, both threads can obtain the non-null `thread` pointer (line 2) and pass the sanitization check (line 3) simultaneously (due to the insufficient range of the lock protection), then one of them proceeds to the free site (line 10). Although at this point, `*threadp` is nullified, since the other thread has already passed the check at line 3, it can still access the freed memory (line 5 and 9). We believe UAFX's capability of analyzing such subtle interaction between multiple code aspects is one major advantage over existing tools.

Case 2. Fig. 7 shows a complex inter-thread cross-entry UAF case identified in lrzip [16]. Line 3 spawns a child thread with `entry()` as its starting point and `sts` as an additional pointer argument. The problem is that the child thread lifecycle is not properly controlled, such that the execution of line 20 (use site, by the child thread) could be postponed after line 12 (free site, by the parent thread), causing a UAF. We discuss some notable aspects of this case below.

(1) Inter-Thread Analysis. The analyzer needs to be aware of the multi-threading model (*e.g.,* semantics of `pthread_create()`) to spot this kind of issue. As described in §III-C, UAFX includes the thread lifecycle recognition in its per-entry analysis, enabling it to correctly identify the thread creation, entry points, and arguments in Fig. 7, essential for inter-thread cross-entry UAF detection.

(2) Complex Alias Relationship. Establishing the alias relationship between `n->sinfo->ucthread` at the free site (line 12) and `uci` at the use site (line 20) is challenging, as the analyzer needs to accurately track the lengthy and subtle interprocedural pointer propagation, handling memory read-/write in a flow-, field-, and context-sensitive way. Less precise techniques employed by previous works (*e.g.,* DCUAF [10]) may cause both false positives (*e.g.,* type-based analysis treat all same-typed pointers as aliases) and false negatives (*e.g.,* field-based analysis may fail to recognize the same `sinfo` instance assigned to different host structures: `sts` at line 2 and `node` at line 6). UAFX can confidently recover such complex alias relationships thanks to its accurate pointer tracking (§III-C), even crossing different entry functions.

VI. Related Work

We discuss several categories of related works as follows.

Static UAF detection. As pointed out by the previous study [17], most existing static UAF detectors only support simple UAF cases (*e.g.,* intraprocedural sequential UAFs). For example, UAFChecker [41] utilizes symbolic execution and taint analysis to catch sequential UAFs following the pre-defined finite state machine. Similarly, many other tools employ these general static analysis techniques to discover (simple) UAF cases, such as Clang Static Analyzer [6], Infer [7], Cppcheck [8], Coverity [42], and Pinpoint [43]. CRED [4] develops context and path reduction techniques to improve the scalability of static UAF detection while maintaining a high precision. UAFDetector [44] uses summary-based alias analysis to efficiently find UAF bugs in binaries. Sys [45] supports UAF detection by combining the scalable (but imprecise) static analysis and precise (but expensive) symbolic execution, for a better tradeoff between efficiency and precision. Many tools like Falwfinder [9] and Palfrey [5] rely on manually summarized patterns for UAF detection, limiting their capabilities. These patterns can also be wrapped into "queries" used by general-purpose static code analysis engines (*e.g.,* CodeQL [46] and Joern [47]) for finding UAFs. Besides the above works, some research aims to develop specific techniques to detect bugs related to UAF, for example, use-after-cgc [48] and use-after-cleanup [49]. LinkRID [36] can vet the imbalance in reference counting for Linux kernel in a local scope (*e.g.,* intra-entry-function), which could lead to UAF bugs, we consider their approach helpful in addressing UAFX's false negatives related to reference counting (§IV), with necessary extensions for cross-entry analysis.

As mentioned, DCUAF [10] and Canary [12] are the only tools we are aware of that support cross-entry UAF detection to certain degrees, however, both have significant limitations as detailed in §V-D. SUTURE [14] targets high-order taint-style vulnerabilities involving multiple entry functions, however, UAF is not in its scope. UAFX develops a precise *escape-fetch*-based cross-entry alias analysis and systematic partial-order based FP filtering on top of SUTURE, for effective and efficient UAF detection.

Dynamic UAF detection. Many previous works utilize dynamic methods (*e.g.,* fuzzing) to hunt concurrency vulnerabilities including UAFs. DDRace [50] employs directed fuzzing to guide the dynamic testing to potential program sites that could contain UAF bugs. UAFL [51] models UAF bugs as a typestate violation, which is the used to guide the fuzzing process. PERIOD [52] proposes to find concurrent bugs with periodical scheduling of simultaneous threads to explore their interleavings more systematically, in a similar spirit, SnowBoard [53] also tries to schedule concurrent threads more intelligently based on their memory communication behaviors for more effective bug finding. MUZZ [54] develops thread-aware instrumentations to help the grey-box fuzzer better trigger bugs in the multithreaded setting. SnowCat [55] utilizes a graph neural network to guide kernel dynamic testing to reveal

concurrency bugs more efficiently. Some other works, like UFO [56], ConVul [57], ConVulPOE [58], ToccRACE [59], and ConPTA [60], focus on predicting potential concurrent bugs from existing program execution traces, by exploring alternative thread interleavings and event orders. Generally speaking, dynamic UAF detection methods suffer from low code coverage due to their random exploration, which can be improved by static approaches as UAFX.

UAF Defense. Researchers propose different mechanisms to defend against UAF bugs. DangNull [61] prevents UAFs at runtime by automatically nullifying the freed pointers. DangSan [62] is a scalable runtime UAF detection mechanism with an efficient shadow memory-based metadata management scheme, UAFSan [63] and ViK [64] achieve a similar goal by assigning and matching unique object identifiers for pointers and objects. BOGO [65] utilizes Intel MPX - a hardware feature - to identify memory safety issues at runtime, similarly, PTAuth [66] and PACMem [67] achieve this with ARM Pointer Authentication, xTag [68], on the other hand, implements a software-based pointer tagging mechanism on Intel x86-64 where no hardware features like ARM PAC are unavailable. CHERI is an architecture-level extension that can enforce memory safety properties [69], [70], [71]. Oscar [72] develops a memory page permission-based method to thwart UAF access, while DangZero [73] takes advantage of the ring0 level direct page table access for more efficient UAF detection and prevention. PSweeper [74] utilizes concurrent threads on spare cores to sweep the memory pointers and diagnose potential UAF issues, while MineSweeper [75] retains the freed memory buffers in an isolated area until no dangling pointers exist, effectively preventing UAFs. GUARDER [76] is a secure and tunable heap allocator that randomizes the memory buffer allocation, making the UAF exploitation more difficult. Allocator-based approaches are also widely adopted by many other works, such as MarkUs [77] and FFmalloc [78]. SAVER [79] aims to automatically repair the UAF issues reported in programs, with static analysis, on the other hand, Zhou *et al.* [80] develop a secure pointer extension at the C language level to help mitigate UAF cases. CRCount [81] introduces an accurate reference counting mechanism to help mitigate UAF problems, while FreeWill [37] helps developers understand and patch the known UAF issues with a reference count analysis, such analysis can also be complementary to UAFX. In general, despite the existence of various approaches for UAF prevention, we believe that early discovery and fixing of UAF vulnerabilities is still necessary, given the limited availability and performance overhead of the defenses.

VII. Conclusion

We propose UAFX, a static analyzer capable of finding cross-entry UAF bugs in Linux kernel and potentially general C programs. UAFX employs a novel escape-fetch-based pointer analysis to accurately recover the subtle cross-entry alias relationship between the use and free sites, and a systematic multi-factor UAF validation framework based on partial-order constraint solving. UAFX successfully discovers unknown cross-entry UAF issues in Linux kernel and user programs, with practical accuracy and performance.

Acknowledgment

We sincerely thank the anonymous reviewers and our shepherd for their insightful comments and actionable suggestions. We thank Weiteng Chen (Microsoft Research), Chulwoo Shim (Independent Researcher), and Mingyi Liu (Georgia Tech) for their help to this project. We greatly appreciate the Affiliated Institute of ETRI (Electronics and Telecommunications Research Institute, ROK) 's support for this project through a collaborative research grant.

References

[1] Mitre, "Stubborn Weaknesses in the CWE Top 25," https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html, 2024.
[2] Google, "AddressSanitizer," https://github.com/google/sanitizers/wiki/AddressSanitizer, 2024.
[3] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. A. Sani, "Demystifying the dependency challenge in kernel fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 659–671.
[4] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 327–337.
[5] Z. Chen, D. Liu, J. Xiao, and H. Wang, "All use-after-free vulnerabilities are not created equal: An empirical study on their characteristics and detectability," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 623–638.
[6] "Clang Static Analyzer," https://clang-analyzer.llvm.org, 2024.
[7] "Infer Static Analyzer," https://fbinfer.com, 2024.
[8] "Cppcheck," https://cppcheck.sourceforge.io, 2024.
[9] "Flawfinder," https://dwheeler.com/flawfinder/, 2024.
[10] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency {Use-After-Free} bugs in linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 255–268.
[11] Google, "Syzkaller," https://github.com/google/syzkaller, 2024.
[12] Y. Cai, P. Yao, and C. Zhang, "Canary: practical static detection of inter-thread value-flow bugs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1126–1140.
[13] "The Z3 Theorem Prover," https://github.com/Z3Prover/z3, 2024.
[14] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 811–824.
[15] "The LLVM Compiler Infrastructure," https://llvm.org, 2024.
[16] "lrzip," https://github.com/ckolivas/lrzip, 2024.
[17] B. Gui, W. Song, H. Xiong, and J. Huang, "Automated use-after-free detection and exploit mitigation: How far have we gone?" *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4569–4589, 2021.
[18] Google, "Syzbot Dashboard: Issues in Linux Kernel Subsystems," https://syzkaller.appspot.com/upstream/subsystems, 2024.
[19] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," *IEEE Transactions on Software Engineering*, 2023.
[20] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux." in *NDSS*, 2018.
[21] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1899–1913.
[22] J. Wang, S. Ma, Y. Zhang, J. Li, Z. Ma, L. Mai, T. Chen, and D. Gu, "{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)*, 2019, pp. 309–321.

[23] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "{PeX}: A permission check analysis framework for linux kernel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1205–1220.

[24] K. Lu, A. Pakki, and Q. Wu, "Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1769–1786.

[25] A. Pakki and K. Lu, "Exaggerated error handling hurts! an in-depth study and context-aware detection," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1203–1218.

[26] Y. Zhai, Y. Hao, H. Zhang, D. Wang, C. Song, Z. Qian, M. Lesani, S. V. Krishnamurthy, and P. Yu, "Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 221–232.

[27] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, "Detecting kernel refcount bugs with {Two-Dimensional} consistency checking," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2471–2488.

[28] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *The 2021 Annual Network and Distributed System Security Symposium (NDSS'21)*, 2021.

[29] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. Clement, and N. Sundaresan, "Learning to reduce false positives in analytic bug detectors," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1307–1316.

[30] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger *et al.*, "Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel," in *2022 Network and Distributed System Security Symposium*, 2022.

[31] A. S. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni, "" false negative-that one is going to kill you."-understanding industry perspectives of static analysis based security testing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 19–19.

[32] K. Lu and H. Hu, "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*, 2019, pp. 1867–1881.

[33] W. He, P. Di, M. Ming, C. Zhang, T. Su, S. Li, and Y. Sui, "Finding and understanding defects in static analyzers by constructing automated oracles," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1656–1678, 2024.

[34] NIST, "Juliet C/C++ 1.3 - NIST Software Assurance Reference Dataset," https://samate.nist.gov/SARD/test-suites/112, 2024.

[35] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, "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*, 2016, pp. 531–544.

[36] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi, "{LinKRID}: Vetting imbalance reference counting in linux kernel with symbolic execution," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 125–142.

[37] L. He, H. Hu, P. Su, Y. Cai, and Z. Liang, "{FreeWill}: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2497–2512.

[38] S. Bai, Z. Zhang, and H. Hu, "Countdown: Refcount-guided fuzzing for exposing temporal memory errors in linux kernel," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[39] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[40] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "{DR}.{CHECKER}: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1007–1024.

[41] J. Ye, C. Zhang, and X. Han, "Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1529–1531.

[42] "Coverity," https://scan.coverity.com, 2024.

[43] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 693–706.

[44] K. Zhu, Y. Lu, and H. Huang, "Scalable static detection of use-after-free vulnerabilities in binary code," *IEEE Access*, vol. 8, pp. 78 713–78 725, 2020.

[45] F. Brown, D. Stefan, and D. Engler, "Sys: A {Static/Symbolic} tool for finding good bugs in good (browser) code," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 199–216.

[46] "CodeQL," https://codeql.github.com, 2024.

[47] "Joern," https://joern.io, 2024.

[48] H. Han, A. Wesie, and B. Pak, "Precise and scalable detection of {Use-after-Compacting-Garbage-Collection} bugs," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2059–2074.

[49] L. Ma, D. Zhou, H. Wu, Y. Zhou, R. Chang, H. Xiong, L. Wu, and K. Ren, "When top-down meets bottom-up: Detecting and exploiting use-after-cleanup bugs in linux kernel," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2138–2154.

[50] M. Yuan, B. Zhao, P. Li, J. Liang, X. Han, X. Luo, and C. Zhang, "Ddrace: Finding concurrency uaf vulnerabilities in linux drivers with directed fuzzing," in *32th USENIX Security Symposium (USENIX Security 23)*, 2023.

[51] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 999–1010.

[52] C. Wen, M. He, B. Wu, Z. Xu, and S. Qin, "Controlled concurrency testing via periodical scheduling," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 474–486.

[53] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 66–83.

[54] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "{MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2325–2342.

[55] S. Gong, D. Peng, D. Altınbüken, P. Fonseca, and P. Maniatis, "Snowcat: Efficient kernel concurrency testing using a learned coverage predictor," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 35–51.

[56] J. Huang, "Ufo: predictive concurrency use-after-free detection," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 609–619.

[57] R. Meng, B. Zhu, H. Yun, H. Li, Y. Cai, and Z. Yang, "Convul: an effective tool for detecting concurrency vulnerabilities," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1154–1157.

[58] K. Yu, C. Wang, Y. Cai, X. Luo, and Z. Yang, "Detecting concurrency vulnerabilities based on partial orders of memory and thread events," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 280–291.

[59] S. Zhu, Y. Guo, L. Zhang, and Y. Cai, "Tolerate control-flow changes for sound data race prediction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1342–1354.

[60] Y. Guo, S. Zhu, Y. Cai, L. He, and J. Zhang, "Reorder pointer flow in sound concurrency bug prediction," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 192–204.

[61] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*, 2015.

[62] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 405–419.

[63] B. Gui, W. Song, and J. Huang, "Uafsan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 309–321.

[64] H. Cho, J. Park, A. Oest, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Vik: practical mitigation of temporal memory safety violations through object id inspection," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 271–284.

[65] T. Zhang, D. Lee, and C. Jung, "Bogo: Buy spatial memory safety, get temporal memory safety (almost) free," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 631–644.

[66] R. M. Farkhani, M. Ahmadi, and L. Lu, "{PTAuth}: Temporal memory safety via robust points-to authentication," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1037–1054.

[67] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1901–1915.

[68] L. Bernhard, M. Rodler, T. Holz, and L. Davit, "xtag: Mitigating use-after-free vulnerabilities via software-based pointer tagging on intel x86-64," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 502–519.

[69] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 457–468, 2014.

[70] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson *et al.*, "Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 545–557.

[71] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin *et al.*, "Cornucopia: Temporal safety for cheri heaps," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 608–625.

[72] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical {Page-Permissions-Based} scheme for thwarting dangling pointers," in *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 815–832.

[73] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "Dangzero: Efficient use-after-free detection via direct page table access," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1307–1322.

[74] D. Liu, M. Zhang, and H. Wang, "A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1635–1648.

[75] M. Erdős, S. Ainsworth, and T. M. Jones, "Minesweeper: a "clean sweep" for drop-in use-after-free prevention," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 212–225.

[76] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu, "Guarder: A tunable secure allocator," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 117–133.

[77] S. Ainsworth and T. M. Jones, "Markus: Drop-in use-after-free prevention for low-level languages," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 578–591.

[78] B. Wickman, H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim, "Preventing {Use-After-Free} attacks with fast forward allocation," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2453–2470.

[79] S. Hong, J. Lee, J. Lee, and H. Oh, "Saver: scalable, precise, and safe memory-error repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 271–283.

[80] J. Zhou, J. Criswell, and M. Hicks, "Fat pointers for temporal memory safety of c," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 316–347, 2023.

[81] J. Shin, D. Kwon, J. Seo, Y. Cho, and Y. Paek, "Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++." in *NDSS*, 2019.