

TriMatch: Detecting Linux Kernel Memory Bugs with 3D Object-Aware Symmetry Checking

Zilin Guan
Southeast University
Nanjing, China
zilin@seu.edu.cn

Jianhao Xu
Southeast University
Nanjing, China
jianhao.xu@seu.edu.cn

Dawei Feng
Southeast University
Nanjing, China
dawei.feng@seu.edu.cn

Kangjie Lu
University of Minnesota
Minneapolis, USA
kjl@umn.edu

Bing Mao
State Key Laboratory for Novel
Software Technology, Nanjing
University
Nanjing, China
maobing@nju.edu.cn

Guang Cheng
Southeast University
Nanjing, China
chengguang@seu.edu.cn

Abstract

Linux kernel memory management is characterized by deep wrapper chains, nested struct fields, and intricate error-handling paths. In such environments, a deallocation that appears syntactically correct at a call site can be semantically incomplete or mismatched, failing to truly invalidate the target logical object, thereby introducing subtle yet exploitable memory bugs. Existing pairing-based analyses reason about allocation-release symmetry mostly at the function level rather than at the object level, and therefore cannot verify whether the corresponding wrapper chains ultimately reach the same internal object. Summary-based approaches offer object-level precision, but cannot precisely track objects across nested fields and complex pointer manipulations, and ignore syntactic pairing constraints between alloc-side and release-side wrappers.

We present TriMatch, an LLVM-based static analyzer for detecting Linux kernel memory bugs through *3D object-aware symmetry checking*. TriMatch builds bidirectional memory-operation summaries over a normalized access-path representation, constructs a wrapper-aware pairing corpus for operations targeting the same logical object, and checks the resulting chains along three dimensions: Depth matching across wrapper layers, Horizontal matching across sibling branches, and Vertical matching within one execution path. This design detects incompatible release functions, missing releases, and repeated releases of the same logical object. Applied to Linux 6.13-rc1, TriMatch found 115 new bugs; as of this writing, 94 have been fixed or confirmed; 42 CVE IDs have been assigned to the reported bugs. These results show that TriMatch can uncover deep logic flaws in kernel memory management that evade existing sanitizers and static tools while remaining practical for large-scale analysis.

CCS Concepts

• **Security and privacy** → **Operating systems security**; *Software security engineering*; • **Software and its engineering** → *Software testing and debugging*; *Operating systems*.

Keywords

Linux kernel; function pairing; memory bug detection; static analysis

ACM Reference Format:

Zilin Guan, Jianhao Xu, Dawei Feng, Kangjie Lu, Bing Mao, and Guang Cheng. 2026. TriMatch: Detecting Linux Kernel Memory Bugs with 3D Object-Aware Symmetry Checking. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3830454.3846675>

1 Introduction

Linux kernels sit on privileged execution paths where a memory-management mistake can become a crash, a denial-of-service vector, or a stepping stone to privilege escalation [31, 32, 44]. While dynamic approaches such as fuzzing are highly effective at exposing concrete failures [5, 8, 9, 12–15, 34, 39, 41], they are inherently restricted to the execution paths they can exercise. Static analysis is thus indispensable, as it can exhaustively inspect memory-related behaviors that are difficult to trigger in practice, such as deep error-handling logic and inter-procedural allocation-release obligations [4, 10, 17–19, 25, 26, 35, 38, 40]. The hard cases are rarely plain `kmalloc()` / `kfree()` mismatches. Kernel allocation and release operations are usually performed through helper functions that wrap the underlying primitives. We refer to such helpers as *alloc wrappers* and *release wrappers* and to their nesting as wrapper chains. In real code, an object may be allocated inside one helper, exposed through another, stored into a structure field, propagated across deep, multi-layer wrapper chains, and finally released on a heavily branched cleanup path. Static verification becomes difficult precisely because the lifetime obligation is scattered across deep call chains, nested fields, and complex path contexts.

One line of prior work addresses this problem through function pairing. Systems such as SinkFinder [3], APISpecGen [22], and



This work is licensed under a Creative Commons Attribution 4.0 International License. CCS '26, The Hague, Netherlands.

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
<https://doi.org/10.1145/3830454.3846675>

```

1 static void tipc_aead_free(struct rcu_head *rp)
2 {
3     struct tipc_aead *aead = container_of(rp, struct tipc_aead, rcu);

4     free_percpu(aead->tfm_entry);
5     kfree_sensitive(aead->key);
6     kfree(aead);
7 }

```

Figure 1: Simplified `tipc_aead_free` cleanup wrapper entered through an embedded-field pointer.

HERO [36] are effective at discovering plausible allocation-release function pairs. However, these analyses remain function-centric: they can infer which allocator and deallocator functions may pair, but cannot verify whether a concrete alloc-side and release-side wrapper chain ultimately operates on the same internal object. In kernel code, a top-level object may carry multiple nested resources, so an apparently well-paired release wrapper may still miss or mismatch the cleanup required for a particular subobject. Function-level pairing therefore provides a useful candidate corpus, but not sufficient evidence of object-level correctness. Object-level reasoning is therefore necessary for deep kernel wrappers.

This analytical complexity has driven the adoption of summary-based analyses like Goshawk [30] and IMMI [23], which abstract memory behavior around logical objects to avoid exponential path explosion [1, 2, 20]. However, existing summary frameworks exhibit two main limitations when analyzing deep kernel wrappers. First, they lack pairing constraints to determine whether an alloc-side and release-side wrapper are plausible syntactic counterparts. Second, they struggle with precise object modeling in the presence of nested fields and complex pointer manipulations.

Figure 1 illustrates the challenge. The wrapper enters through an embedded-field pointer `rp` (line 1), recovers the enclosing `tipc_aead` object via `container_of` (line 3), then frees nested resources (lines 4–6). This pattern exposes a fundamental limitation of existing summaries. A forward-only summary can trace field accesses like `aead->tfm_entry` and `aead->key`, but cannot explain how the analysis reached `aead` from `rp`. Conversely, a recovery-only summary captures the `container_of` transition yet loses the subsequent field-sensitive cleanup operations. The same logical object thus fragments across disjoint forward and backward paths. This is not a corner case: Uncontained [16] reveals that over 23% of types in Linux v5.17 participate in such downcast relationships.

Our solution, TriMatch, bridges function-level pairing and object-level validation through bidirectional summaries and 3D symmetry checking. TriMatch first builds bidirectional memory-operation summaries that unify forward field propagation and backward object recovery into a normalized access-path representation, allowing the same logical object to be tracked consistently across nested fields and pointer transformations. It then constructs a wrapper-aware pairing corpus over canonical object paths and evaluates each candidate along three dimensions: **Depth** (wrapper chain compatibility), **Horizontal** (branch consistency), and **Vertical** (no duplicate releases along a path).

Together, these components define TriMatch: pairing provides a broad candidate corpus, while 3D symmetry checking validates

object-level matches and reports mismatches, missing releases, and overlaps.

Our evaluation on Linux 6.13-rc1 shows that TriMatch is both effective and scalable for practical full-kernel discovery of memory-management bugs. Within 1 hour and 41 minutes, it completes a full-kernel analysis and identifies diverse bug classes across multiple subsystems, including memory leaks, double frees, API misuses, and use-after-frees. This coverage further extends to long-standing legacy bugs, several of which have remained hidden for more than 20 years, and most of these bugs are missed by existing state-of-the-art tools.

In summary, the key contributions of this work are:

- **An object-level pairing validation framework for deep kernel wrappers.** We present TriMatch, an approach that bridges permissive function-level pairing with rigorous semantic verification by validating allocation-release symmetry at the object level.
- **Bidirectional object tracking via normalized access paths.** We introduce field-aware bidirectional summaries. By formalizing forward field propagation alongside backward object recovery into a unified access-path schema, TriMatch tracks resources consistently across deep summary boundaries and complex data structures, enabling effective 3D symmetry-checking for Depth, Horizontal, and Vertical dimensions.
- **A scalable implementation and numerous new bugs.** We implement TriMatch as an LLVM-based prototype and apply it to Linux 6.13-rc1. It found 115 new bugs; as of this writing, 94 have been fixed or confirmed; 42 CVE IDs have been assigned to the reported bugs.

2 Motivation

Function pairing alone or mere memory-operation summaries are insufficient for detecting memory bugs in deep kernel wrappers. We present two failure modes that motivate object-level function pairing.

Function-pairing mismatch. Figure 2 shows a cleanup call on an error path that is syntactically present but semantically wrong. The allocation uses `vzalloc()` (`vmalloc` family), while `bitmap_free()` eventually calls `kfree()`. An analysis relying merely on surface-level pairing rules or checking only whether the tracked object is eventually passed to a release function would accept this code, failing to detect the underlying allocator-family mismatch across wrapper layers.

Object-level asymmetry. Even plausible function pairs can be internally inconsistent. In Figure 3, `ipc_mux_init()` and `ipc_mux_deinit()` appear well-paired by name, but the `deinit` routine only releases two of the three allocated resources (the `pp_qlt` array elements are leaked). Function pairing does not capture which object is managed, where it is stored, or how wrappers behave internally.

These cases show that function pairing should serve only as a permissive candidate corpus, not as final evidence. True validation requires field-sensitive, object-level reasoning and cross-layer wrapper compatibility.

```

1 static int pds_vfio_dirty_alloc_bitmaps(struct pds_vfio_region *
    region,
2                                     unsigned long bytes) {
3     region->host_seq_bmp = vzalloc(bytes);
4     ...
5     if (!region->host_ack_bmp) {
6         bitmap_free(region->host_seq_bmp); // internally calls kfree()
7         return -ENOMEM;
8     }
9 }

```

Figure 2: A release-like cleanup call paired with the wrong allocator family.

```

1 struct iosm_mux *ipc_mux_init(struct ipc_mux_config *mux_cfg, ...)
2 {
3     // Alloc resource 1
4     struct iosm_mux *ipc_mux = kzalloc(sizeof(*ipc_mux), ...);
5     ...
6     for (i = 0; i < IPC_MEM_MUX_IP_SESSION_ENTRIES; i++) {
7         // Alloc resource 2
8         ipc_mux->ul_adb.pp_qlt[i] = kzalloc(qlt_size, ...);
9     }
10    for (i = 0; i < ul_tds; i++) {
11        // Alloc resource 3
12        skb = ipc_pcie_alloc_skb(...);
13        skb_queue_tail(&ipc_mux->ul_adb.free_list, skb);
14    }
15    return ipc_mux;
16 }
17 void ipc_mux_deinit(struct iosm_mux *ipc_mux) {
18     ...
19     while ((skb = skb_dequeue(&ipc_mux->ul_adb.free_list))) {
20         // Release resource 3
21         ipc_pcie_kfree_skb(ipc_mux->pcie, skb);
22     }
23     // Release resource 1
24     kfree(ipc_mux);
25 }

```

Figure 3: A function-paired init/deinit pair can still be object-level asymmetric.

3 Threat Model and Scope

This paper targets memory-management bugs arising from asymmetric cleanup of kernel objects. Our threat model assumes an adversary, either an unprivileged local user or a remote party capable of sending network packets, who can invoke system calls or device-specific inputs to steer the kernel into deep execution paths and error-handling logic where such asymmetries reside.

These bugs translate into concrete security consequences in two ways. First, a missing release allows leaked memory to accumulate across repeated invocations, progressively exhausting kernel resources and causing denial of service; abandoned objects may also retain sensitive data. Second, incompatible or repeated releases corrupt kernel heap state, manifesting as double frees or use-after-free conditions that can be chained into memory corruption and, potentially, privilege escalation.

4 Workflow

Figure 4 gives the high-level architecture of TriMatch. The workflow has three parts. The *summary generation* stage starts from kernel source code, compiles it to LLVM IR, extracts call-graph and control-flow information, and uses a compact API knowledge base as seeds for data-flow and wrapper analysis to build *bidirectional summaries*. These summaries record, for each function, the memory objects it allocates or releases and how those objects are propagated beyond the function boundary, so later matching can reason about one logical object across wrapper layers.

The *matching analysis* stage consumes those summaries in two phases. Pairing corpus construction first builds a permissive alloc/free pairing corpus by combining wrapper-derived summaries, intra-procedural alias connectivity, and token-level name matching. The analysis then applies this corpus along concrete allocation and release call chains. In the depth dimension, it performs multi-level pairing analysis to decide whether a candidate release chain is compatible with the allocation chain for the same logical object. Only depth-matched candidates are then carried forward to the horizontal and vertical dimensions.

Two design choices make this pipeline effective. First, TriMatch materializes wrapper summaries before any pair mining. Each summary caches the wrapper’s relevant memory operation together with how the tracked object crosses the function boundary. These cached records let later stages reuse concrete object-level evidence across wrapper composition instead of rediscovering the same wrapper behavior for every later comparison. Second, TriMatch keeps function pairing deliberately permissive and uses it only to build a broad candidate set from these wrapper summaries. Bug reports are produced only after object-aware symmetry validation, which prevents noisy pairing evidence from directly becoming reports and ensures that *mismatch*, *lack*, and *overlap* reports are all supported by validated object-level inconsistencies.

The output is a structured bug report for one of three target classes. *Function mismatch* reports identify objects that are released through an incompatible cleanup chain. *Function lack* reports identify error-handling paths that miss a required release. *Function overlap* reports identify repeated releases of the same object. Each report includes the allocation instruction, the relevant access path, the release or missing-release context, and the call chains that led to the decision.

5 Design

5.1 Memory-Operation Summarization

Object-centric memory-operation summaries balance cost and precision for Linux-kernel analysis. Prior systems such as Goshawk [30] and IMMI [23] demonstrate the value of this approach but fall short for our setting. Goshawk’s summaries grow deeper as the call chain expands and consume substantial resources, whereas IMMI mitigates this cost with on-demand, partitioned summary construction. Crucially, however, neither incorporates backward object recovery from an embedded field to its enclosing object into its summary model.

The wrapper in Figure 1 captures the requirement that drives the core design of our summarization module. The analysis must connect entry through an embedded-field pointer, recovery of the

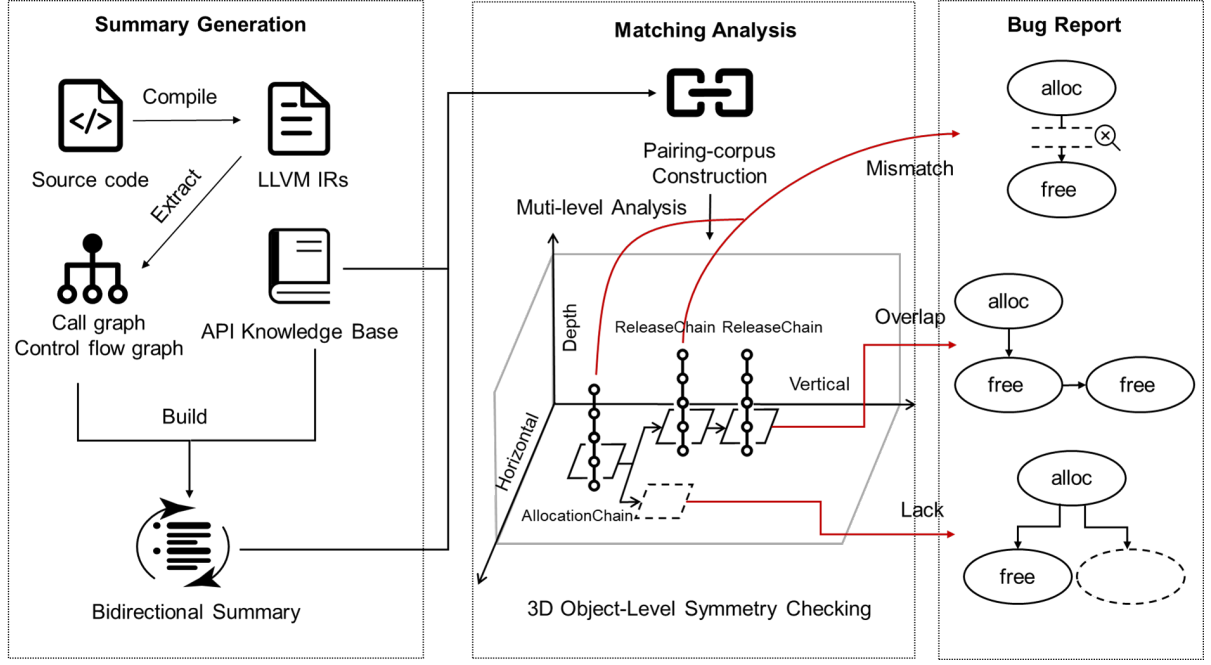


Figure 4: Overview of TriMatch.

```

1 #define container_of(ptr, type, member) \
2 ((type *)((char *)(ptr) - offsetof(type, member)))

```

Figure 5: Simplified container_of macro.

enclosing object, and the subsequent cleanup of nested resources as one object lifecycle. We therefore preserve the recovery step and the later field traversals in the same normalized representation so that later matching can reason about one logical object consistently.

Challenges. In the Linux kernel, backward object recovery is performed through `container_of` and its wrapper macros, which in LLVM IR are lowered into pointer arithmetic and type conversions; TriMatch must therefore recognize these patterns and distinguish them from ordinary structure-field accesses. Moreover, once backward object recovery is recorded, the allocation side and release side may reach the same object through different access forms, requiring a normalized representation that collapses them to the same logical object.

Identifying container_of patterns. Figure 5 presents a simplified version of `container_of` that keeps only the semantics relevant to our analysis, namely, converting a member pointer to a byte pointer, subtracting the member offset, and casting the result back to the enclosing type.

Guided by the macro semantics in Figure 5 and the IR lowering in Figure 6, the analysis recognizes `container_of` patterns in two steps. Syntactically, a candidate must match the lowering pattern, which consists of a cast from an inner-structure pointer to `i8*`, a negative-offset byte-level `getelementptr`, and a cast back to an outer-structure pointer. Layout-based validation then uses LLVM’s `DataLayout` to

```

1 %field = load %struct.callback_head*, %struct.callback_head** %rp.
   addr
2 %field.i8 = bitcast %struct.callback_head* %field to i8*
3 store i8* %field.i8, i8** %tmp1
4 %p = load i8*, i8** %tmp1
5 %base.i8 = getelementptr i8, i8* %p, i64 -48
6 %obj = bitcast i8* %base.i8 to %struct.tipc_aead*
7 store %struct.tipc_aead* %obj, %struct.tipc_aead** %tmp2
8 %res = load %struct.tipc_aead*, %struct.tipc_aead** %tmp2

```

Figure 6: Simplified LLVM IR lowering of container_of.

check that the outer structure contains a field whose type matches the inner structure and whose byte offset cancels the negative GEP displacement. Only then is the GEP treated as a genuine `container_of` recovery, and it is recorded with the same field index used for ordinary forward traversal, so forward propagation and backward recovery share a unified object model.

Interprocedural chain construction and access-path normalization. The analysis builds interprocedural chains through a memoized recursion over summary transitions. Starting from an exposed allocation or release position, the collector traverses outgoing summary edges; each edge contributes a local access-path fragment and a call-chain node. If the edge reaches a primitive endpoint, the fragment is emitted directly; otherwise the collector recurses on the downstream node and composes the returned suffixes with the local fragment, while prepending the current call-chain node so that path and chain composition advance together. Repeated subproblems are served from a cache keyed by the current summary state, while a visited set cuts off cycles.

Algorithm 1 Interprocedural chain construction and access-path normalization

```

1  CollectChains( $v, \pi, \gamma, n$ );
   Input: start summary node  $v$ , caller-side path prefix  $\pi$ , caller-side
   chain prefix  $\gamma$ , normalization flag  $n$ ;
   Output: finalized chain map  $M$ : normalized access path  $\rightarrow$  set of
   call chains;
   Notation:  $\cdot$  concatenates access-path fragments;  $\circ$  concatenates
   call-chain nodes;
2   $C \leftarrow \emptyset$ ;  $V \leftarrow \emptyset$ ;  $R \leftarrow \text{Expand}(v, C, V)$ ;
3  foreach  $(\sigma, S) \in R$  do
4     $p \leftarrow \pi \cdot \sigma$ ;
5    if  $n$  then  $p \leftarrow \text{Normalize}(p)$ ;
6    foreach  $c \in S$  do
7       $M[p] \leftarrow M[p] \cup \{\gamma \circ c\}$ ;
8    end
9  end
10 return  $M$ ;
11 Function Expand( $v, C, V$ ):
12   if  $v \in \text{dom}(C)$  then return  $C[v]$ ;
13   if  $v \in V$  then return  $\emptyset$ ;
14    $V \leftarrow V \cup \{v\}$ ;  $R \leftarrow \emptyset$ ;
15   foreach outgoing summary edge  $e$  of  $v$  do
16      $p \leftarrow \text{AccessFragment}(e)$ ;
17     if  $e$  reaches a primitive endpoint then
18        $R[p] \leftarrow R[p] \cup \{\text{Node}(e)\}$ ;
19     else
20        $R' \leftarrow \text{Expand}(\text{NextNode}(e), C, V)$ ;
21       foreach  $(\sigma, S) \in R'$  do
22          $p' \leftarrow p \cdot \sigma$ ;
23         foreach  $c \in S$  do
24            $R[p'] \leftarrow R[p'] \cup \{\text{Node}(e) \circ c\}$ ;
25         end
26       end
27     end
28   end
29    $V \leftarrow V \setminus \{v\}$ ;  $C[v] \leftarrow R$ ; return  $R$ ;
30 Function Normalize( $p$ ):
31   repeat  $p_{\text{old}} \leftarrow p$ ;
32   foreach  $f \in \text{UniqueFieldSymbols}(p)$  do
33      $p \leftarrow \text{DeleteAll}(p, f|f)$ ;  $p \leftarrow \text{DeleteAll}(p, |ff)$ ;
34   end
35   until  $p = p_{\text{old}}$ ; return  $p$ ;

```

Normalization follows the same object model. Forward traversal through field i is encoded as i , and container_of-style recovery through the same field is encoded as $|i$. After concatenation, the normalizer repeatedly cancels adjacent inverse pairs until a fixed point is reached, since $i|i$, which descends through a field and immediately recovers it, and $|i$, which performs the reverse, are semantically redundant. Algorithm 1 summarizes the memoized chain-construction procedure. Under this normalized representation, equivalent traversals assembled from different combinations of forward traversal and backward recovery collapse to the same logical object before matching; the recursion terminates because the visited set cuts off cycles and the cache expands each subproblem at most once.

5.2 Pairing-corpus Construction

Function names carry useful pairing evidence, but TriMatch uses them only to build a permissive candidate corpus, not to prove an alloc/free match. Following prior work, we borrow two lexical signals: paired interfaces often expose role-specific operation words [27, 28, 36], and related interfaces share a common family stem [11]. The corpus builder therefore favors wrappers whose names preserve the same stem while differing in alloc/free-specific tokens, leaving object-level validation to the later symmetry-checking stage.

Wrapper-seeded local pairing. Rather than compare names globally, TriMatch starts from the bidirectional wrapper summaries of the previous stage. The primary *witness* that two operations act on the same logical object is intra-procedural aliasing between an allocation-side exposed object and a release-side argument. For each function body F , it keeps only pairs of summary-mapped alloc/release callees whose exposed object and release-relevant argument connect to the same caller-local object. Let $P_A(a)$ and $P_R(r)$ denote the alloc-side exposed positions and release-side consumed argument positions recorded for wrappers a and r . For $k_a \in P_A(a)$, $\text{exp}_{k_a}(c_a)$ is the caller-visible value exposed at call site c_a , which is the returned SSA value when $k_a = \text{ret}$ and the actual argument rooted at the recorded out-parameter otherwise. For $k_r \in P_R(r)$, $\text{arg}_{k_r}(c_r)$ is the actual argument at position k_r of c_r . With $x \sim_F y$ denoting the intra-procedural alias relation in F , the witness is

$$\begin{aligned}
 \text{AliasWitness}_F(a, r) = 1 &\iff \exists c_a, c_r, k_a, k_r : \\
 &\text{callee}(c_a) = a \\
 &\wedge \text{callee}(c_r) = r \\
 &\wedge k_a \in P_A(a) \wedge k_r \in P_R(r) \\
 &\wedge \text{exp}_{k_a}(c_a) \sim_F \text{arg}_{k_r}(c_r).
 \end{aligned} \tag{1}$$

Thus wrapper names are compared only after an exposed allocation-side object aliases a release-relevant argument in the same function body.

Name compatibility scoring. Given a witness, TriMatch evaluates lexical compatibility. Both names are split on underscores; pairs with fewer than two tokens, or where one full name is a substring of the other, are discarded as trivial variants within the same interface family. Operation tokens are recognized by role-specific lexicons K_A and K_R (Table 1), curated from Linux kernel naming conventions and prior paired-function mining work [27, 28, 36]; non-operation tokens must still align by substring containment to preserve the shared family stem. Let $T(a)$ and $T(r)$ be the token sequences, m and n their lengths, $[1, m]$ abbreviate $\{1, \dots, m\}$, and $|x|$ the character length of token x . We use $\mathbf{1}[\cdot]$ for the indicator function and $x \sqsubseteq y$ for substring containment:

$$T(a) = \langle a_1, \dots, a_m \rangle, \quad T(r) = \langle r_1, \dots, r_n \rangle \tag{2}$$

$$\begin{aligned}
 \text{op}_A(x) &= \mathbf{1}[\text{alloc} \sqsubseteq x \vee x \in K_A], \\
 \text{op}_R(y) &= \mathbf{1}[\text{free} \sqsubseteq y \vee y \in K_R].
 \end{aligned} \tag{3}$$

For a token pair (a_i, r_j) , the explained length $w(a_i, r_j)$ is

$$\begin{aligned}
 w(a_i, r_j) &= \mathbf{1}[\text{op}_A(a_i) \wedge \text{op}_R(r_j)] \max(|a_i|, |r_j|) \\
 &\quad + \mathbf{1}[\neg \text{op}_A(a_i) \wedge \neg \text{op}_R(r_j) \\
 &\quad \wedge (a_i \sqsubseteq r_j \vee r_j \sqsubseteq a_i)] \min(|a_i|, |r_j|).
 \end{aligned} \tag{4}$$

Table 1: Allocation-side and release-side operation keywords.

Class	Alloc ops	Release ops
Resource	alloc, new, request, create, reserve	free, release, destroy, delete, del, erase
RefCount	get, inc	put, dec, unref, consume
Registration	register, charge	unregister, deregister, uncharge
State	enable, set	disable, clear
Topology	join, add, map, ioremap	leave, remove, unmap, iounmap
Others	apply, open, setup	revert, close, exit, finish, clean, cleanup

An operation-token match receives the longer token length, whereas a stem match receives only the shorter shared length. For an injective token matching $M \subseteq [1, m] \times [1, n]$, let $U_A(M)$ and $U_R(M)$ denote the unmatched token indices on the two sides. The normalization term counts the total token length, charging each matched pair by the longer spelling and each unmatched token by its own length:

$$\text{Norm}(M) = \sum_{(i,j) \in M} \max(|a_i|, |r_j|) + \sum_{i \in U_A(M)} |a_i| + \sum_{j \in U_R(M)} |r_j| \quad (5)$$

$$\text{Score}(M) = \frac{\sum_{(i,j) \in M} w(a_i, r_j)}{\text{Norm}(M)} \quad (6)$$

The resulting score is the explained-length ratio. A name pair is accepted only if every token of the shorter name participates in the matching and the score exceeds the implementation threshold:

$$\begin{aligned} \text{NamePair}(a, r) = 1 \iff & m, n \geq 2 \wedge a \not\subseteq r \\ & \wedge r \not\subseteq a \wedge \exists M : \\ & |M| = \min(m, n) \\ & \wedge \text{Score}(M) \geq \tau_{\text{name}}. \end{aligned} \quad (7)$$

To assess how name filtering affects corpus quality and bug detection, we sweep τ_{name} over $[0, 1]$ with a step size of 0.1, with M constructed by a left-to-right greedy scan. Thresholds 0.6, 0.7, and 0.8 retain 2,982, 2,761, and 2,539 inferred pairs and produce 63, 64, and 76 Mismatch reports, respectively. Manual validation confirms that the report-count differences are all false positives and that the validated bug set remains unchanged. We therefore set $\tau_{\text{name}} = 0.70$, balancing corpus filtering and false-positive control without sacrificing recall.

Corpus completion and normalization. Each accepted wrapper pair becomes a candidate (a, r) in the seed corpus. TriMatch then applies three lightweight, recall-oriented expansions: propagation across naming variants with an optional double-underscore prefix, enumeration of release-name variants obtained by replacing the matched release token with other release keywords, and union with a small primitive API knowledge base such as `kmalloc/kfree` and

`vmalloc/vfree`. The seed and final corpora are

$$C_0 = \{(a, r) \mid \exists F : \text{AliasWitness}_F(a, r) = 1 \wedge \text{NamePair}(a, r) = 1\}, \quad (8)$$

$$C = C_0 \cup \text{PrefixExpand}(C_0) \cup \text{KeywordExpand}(C_0) \cup C_{\text{API}}, \quad (9)$$

5.3 3D Object-Aware Symmetry Checking

A pairing-corpus entry is only a permissive hypothesis. Because the corpus is built from wrapper summaries, local alias witnesses, and lexical evidence, custom alloc/free counterparts with no usable naming or wrapper cues may never enter it. A normalized access path can identify a logical object, but it does not establish that the resource is released by a compatible cleanup function, on every required branch, and only once. The final stage therefore asks whether, under the concrete path context of the observed call sites, the release behavior remains symmetric to the allocation for that same object. We model this symmetry in three dimensions, namely depth compatibility between allocation and release wrapper chains, horizontal coverage across comparable post-allocation branches, and vertical uniqueness along each execution path. For one allocation call site c_a and one normalized object access path o , the analyzer maintains an allocation-chain set $\mathcal{A}_o(c_a)$, the reachable release calls with their release-chain witnesses, and the post-allocation branch partition induced by the surrounding error checks. A candidate is accepted only if it passes all three checks:

$$\text{Match}(c_a, o) = D(c_a, o) \wedge H(c_a, o) \wedge V(c_a, o). \quad (10)$$

Failure of D yields a *mismatch* report. Failure of H yields a *lack* report, and failure of V yields an *overlap* report.

Depth matching across wrapper chains. The first dimension validates alloc/free compatibility across wrapper layers. Starting from c_a , TriMatch expands all allocation chains keyed by normalized access path, directly for primitive allocators or recursively through allocation summaries for wrappers, then scans the reachable code region after c_a under the same intra-procedural alias context. Every release call or wrapper whose argument connects to the allocated object produces a release-chain set; a release candidate is retained only when its access path agrees with an allocation path, meaning both chains reach the same logical object after normalization.

The core test is a suffix-style comparison from the chain base outward. For witnesses $\gamma_a \in \mathcal{A}_o(c_a)$ and $\gamma_r \in \mathcal{R}_o(c_r)$ at release call site c_r , TriMatch first checks compatibility between the release family at the base of γ_r and the allocator at the base of γ_a . It then continues outward through both chains, discharging each pairing-relevant alloc-side wrapper with a recognized release-side counterpart while skipping intermediate wrappers with no pairing role. This depth predicate can be written as

$$\begin{aligned} D(c_a, o) = 1 \iff & \exists c_r, \gamma_a, \gamma_r : \gamma_a \in \mathcal{A}_o(c_a) \\ & \wedge \gamma_r \in \mathcal{R}_o(c_r) \\ & \wedge \text{Compat}(\gamma_a, \gamma_r) = 1 \end{aligned} \quad (11)$$

where `Compat` is the implementation-level realization of this comparison. The rule is intentionally permissive in two ways. First, multiple alloc-side layers may be consumed by one release-side

```

1 static struct field_var *create_field_var(...)
2 {
3     struct hist_field *val = parse_atom(...);
4     struct field_var *field_var;
5
6     if (IS_ERR(val))
7         return ERR_PTR(PTR_ERR(val));
8
9     field_var = kzalloc(sizeof(*field_var), GFP_KERNEL);
10    if (!field_var) {
11        kfree(val); // should be destroy_hist_field(val)
12        return ERR_PTR(-ENOMEM);
13    }
14
15    field_var->val = val;
16    return field_var;
17 }

```

Figure 7: Simplified depth-mismatch example from create_field_var().

chain when an outer alloc wrapper only forwards to another allocator, so the scan need not force a distinct cleanup wrapper per layer. Second, when the scan has already consumed the lower pairing-relevant layers, TriMatch still accepts a single remaining outermost alloc wrapper, since a wrapper may fail before finishing initialization and cleanup may legitimately proceed through a next-layer wrapper or through lower-level steps that discharge only the initialized portion. Outside these exceptions, a depth mismatch is reported whenever a pairing-relevant alloc-side layer is left unconsumed.

Figure 7 shows the representative bug behind this depth check. The object in `val` is allocated through the wrapper chain `parse_atom` → `create_alias` → `create_hist_field` → `kzalloc`. When the later allocation of `field_var` fails, the error path releases `val` with a direct `kfree(val)`, which reaches the correct base free family but skips the destructor layer required by `create_hist_field`, namely `destroy_hist_field` or `__destroy_hist_field`. As Figure 8 shows, the cleanup chain must discharge this pairing-relevant layer before reaching the base `kfree`; intermediate wrappers like `create_alias` may be skipped, but leaving `create_hist_field` unconsumed is a depth mismatch. Conversely, this outward matching establishes non-obvious pairings such as `parse_atom()` and `destroy_hist_field()`, because TriMatch validates structural consumption of inner layers rather than relying on surface names and can thus safely recognize these outer functions as genuine counterparts once their underlying obligations are discharged.

Depth matching also suppresses conflations that would otherwise pollute later reasoning. If the tracked value becomes aliased with a fresh allocation later in the same region, the scanner stops and records a reinitialization boundary instead of mixing two lifetimes. If both matched and mismatched release candidates exist, TriMatch suppresses the mismatch when a depth-matched release already appears on a sibling branch guarded by the same condition, or in an earlier or later release along the same branch segment—cases that usually reflect branch-specific choices or alias imprecision.

Horizontal matching across post-allocation branches. The second dimension checks that the cleanup obligation for the same logical

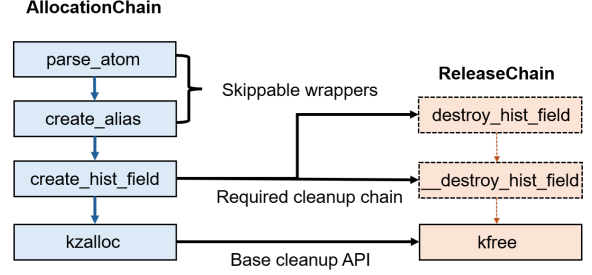


Figure 8: Multi-layer depth matching for parse_atom().

object is handled consistently across the post-allocation branches after c_a . After depth matching identifies which release candidates truly discharge o , TriMatch asks whether every path that still carries the cleanup duty for o either releases it or transfers that obligation elsewhere.

Formally, horizontal consistency requires branches in $\mathcal{B}_o^{\text{req}}(c_a)$ to either release o or reach a proved exemption point:

$$H(c_a, o) = 1 \iff \forall b \in \mathcal{B}_o^{\text{req}}(c_a) : \text{Freed}_o(b) \vee \text{Exempt}_o(b) \quad (12)$$

where $\mathcal{B}_o^{\text{req}}(c_a)$ is the set of post-allocation branches for o whose local behavior must still discharge the cleanup duty created at c_a . A branch b belongs to this set when it is an error-handling or early-exit path in a region where some other path releases o before leaving, and the releasing path serves only as a witness that the duty is still live in the region. This covers both multiple later failure paths that inherit the cleanup obligation and a normal continuation that eventually releases o while an error-handling path does not.

The exemption predicate is effect-based. A branch is exempt when the duty for o is provably transferred, deferred, discharged elsewhere, or rendered unnecessary by non-returning termination. Signals include publishing o through a caller-visible location, returning or delegating the object, installing callback-based cleanup, or non-returning termination such as panic. A monotone disappearance of local frees after a later post-allocation check is also treated as an ownership-transition boundary by default.

The horizontal dimension also extends interprocedurally in restricted form. When a function propagates the cleanup duty through its return value or a pointer argument, TriMatch checks whether its callers discharge that duty on comparable branches of their own. A lack report therefore indicates the joint absence of a proved ownership handoff and of any reliable caller-side discharge of the outstanding obligation.

Vertical matching along a single path. The third dimension checks for repeated cleanup along one execution path. For each access path o , TriMatch orders the release calls that may co-execute and asks whether a later release can be reached before intervening overwrite evidence has discharged the earlier one. It tracks three post-release evidence states based on subsequent variable assignments, denoted S_0 , S_1 , and S_2 for no overwrite, field overwrite, and root-pointer overwrite respectively, as illustrated in Figure 9. A later field-based release is reportable only from S_0 ; a direct-pointer release remains

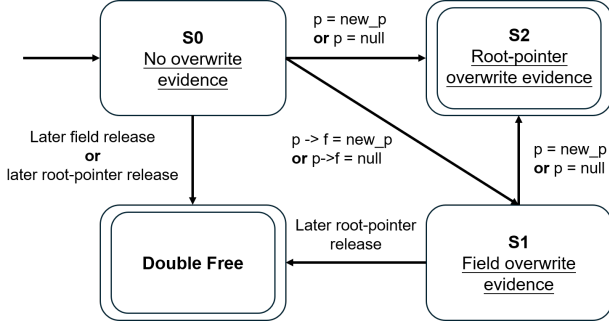


Figure 9: Double-free reportability under overwrite evidence.

reportable from S0 and S1 but is discharged once the state reaches S2.

Formally, the vertical condition is

$$V(c_a, o) = 1 \iff \forall c_{r1} < c_{r2} : \neg \text{CoExec}(c_{r1}, c_{r2}) \vee \text{Overwritten}_o(c_{r1}, c_{r2}) \quad (13)$$

where CoExec denotes that the two releases may both execute in order along one feasible path, and Overwritten_o denotes that intervening effects discharge the earlier release before the later one; V holds only when every co-executable release pair is separated by such evidence. Together, these dimensions convert permissive pairing hypotheses into proof-oriented decisions. By maintaining the normalized access path, wrapper chains, and path context throughout, TriMatch generates precise reports explaining exactly which object, under which path context, violates the symmetry requirements.

6 Implementation

We implement TriMatch as an LLVM-based prototype comprising 4.9 KLoC of C++ code, organized as a pipeline of LLVM passes for call-graph construction, field-sensitive alias analysis, error-handling path identification, construction of wrapper summaries, inference of wrapper pairs, and symmetry bug checking. We reuse IMMI's [23] multi-layer call-graph and field-sensitive alias infrastructure and build on Crix's [29] security-check analysis to identify post-allocation error-handling paths. Below we detail the decisions that matter most for precision and scalability.

Error-handling path identification. TriMatch extends Crix's backward data-flow propagation from error codes to security checks with forward reasoning, because some error paths do not preserve the original error value along the whole chain and instead jump to a later location that returns a fixed error code. TriMatch therefore infers whether a condition may lead to an error return and then identifies the error-handling path through control-flow jumps.

Wrapper-summary construction. TriMatch builds alloc-side and release-side summaries by recursively checking whether the tracked object propagates to outer layers. For release wrappers, construction stops when the function contains an alloc wrapper or a free that belongs only to an error-handling path. Because release summaries depend on alloc summaries, alloc wrappers cannot use the same stopping rule; we instead impose stricter constraints when

Table 2: Categorization of 115 new bugs by 3D dimension.

Dimension	Violation Pattern	Manifested Symptom	Count
Horizontal	Lack of release	Memory leak	94
Depth	Wrapper mismatch	Alloc/free misuse	8
	Wrapper mismatch	Memory leak (cross-layer)	2
Vertical	Overlap release	Double free	9
	Overlap release	Use-after-free	1
	Overlap release	Null-pointer dereference	1

the allocated object is exposed through parameters. Specifically, the alloc operation must not appear inside an if condition, since kernel code checks for failure before the normal path and an in-condition allocation suggests conditional or partial initialization rather than a stable interface, and no more than one parameter may receive the object, since multiple recipients usually reflect imprecise aliasing rather than a clear wrapper boundary.

Bug reporting. Each allocation report identifies the logical resource, by normalized object path, that is mismatched, missing, or repeatedly released on an error-handling path. Mismatch and lack reports also list the expected release functions from the pairing corpus to aid patch development, and all reports include the allocation and release instructions, source locations, and wrapper call chains for direct inspection.

7 Evaluation

Experiments run on a KVM-based x86-64 virtual machine with 32 CPUs, 256 GB of memory, and one NUMA node. We evaluate TriMatch on Linux 6.13-rc1 (commit b8f52214c61a), compiling the target kernel to LLVM IR with -O0 and running the analysis on the resulting bitcode. The evaluation addresses the following questions:

- **RQ1:** How effective is TriMatch at discovering security-relevant memory bugs in the Linux kernel?
- **RQ2:** What are the precision and recall of TriMatch?
- **RQ3:** How much does each core design choice contribute to TriMatch's overall effectiveness?
- **RQ4:** How do TriMatch and state-of-the-art static tools complement each other on their respective validated bug sets?
- **RQ5:** What is the performance overhead of TriMatch when performing full-kernel analysis, and how scalable is it?

7.1 RQ1: Bug Detection Effectiveness

We manually analyzed all reports and found 145 to be real bugs, of which 115 were previously unknown. All 115 have been reported to the kernel community with developed and tested patches. To date, 80 fixes have been merged into mainline, 75 of which are also backported to LTS branches; 14 bugs have been confirmed by maintainers, and 21 remain under review. Additionally, 42 CVE IDs have been assigned to the reported bugs.

3D symmetry categorization. Table 2 categorizes the 115 new bugs by dimension. Horizontal violations, a lack of cleanup on error paths, account for 94 memory leaks and form the dominant

```

1 static ssize_t do_rbd_add(...) {
2     rc = device_add_disk(&rbd_dev->dev, rbd_dev->disk, NULL);
3     if (rc)
4         goto err_out_cleanup_disk;
5     ...
6 err_out_cleanup_disk:
7     rbd_free_disk(rbd_dev); // 1st release: frees rbd_dev->disk
8     err_out_image_lock:
9     rbd_dev_device_release(rbd_dev); // 2nd release (fallthrough)
10 }
11
12 static void rbd_dev_device_release(struct device *dev) {
13     rbd_free_disk(rbd_dev); // repeated release -> NULL-ptr deref
14 }

```

Figure 10: Premature release in `do_rbd_add()` causing a Vertical symmetry violation and a null-pointer dereference.

category that leads to progressive memory exhaustion. Depth violations contribute 10 findings where allocation and release wrapper chains belong to incompatible families, such as `vzalloc` freed via `bitmap_free` in Figure 2, causing heap corruption. Vertical violations contribute 11 findings from repeated or premature releases, comprising 9 double frees, 1 use-after-free, and 1 null-pointer dereference. For example, in Figure 10, when `device_add_disk()` fails in `do_rbd_add()`, the error path calls `rbd_free_disk()` and then falls through to `rbd_dev_device_release()`, which frees the disk a second time, invalidating block-queue state and triggering the null-pointer dereference.

Code distribution. Figure 11a shows the distribution across subsystem families. Networking is the largest bucket with 39 bugs, followed by `scsi` with 16 and `fs` with 15; staging and media contribute 8 each, `gpu` 7, `usb` and `infiniband` 4 each, core kernel code 1, and the residual other category 13. Alloc/free asymmetry thus recurs across networking, storage, filesystem, device, and core-kernel code rather than being confined to any single programming style.

Bug lifetime. Figure 11b shows how long the bugs survived before TriMatch found them: only 8 bugs are younger than two years, 44 remained for 2–5 years, 34 for 5–10 years, and 29 persisted for at least a decade, including 15 for 10–15 years, 9 for 15–20 years, and 5 for more than 20 years. These are long-lived symmetry violations that can hide in uncommon error-handling paths and deep wrapper layers across many release cycles, not short-lived development mistakes.

7.2 RQ2: Practical Precision and Recall

Overall precision. Two authors independently reviewed every report against the Linux source. A report was a genuine bug if the flagged asymmetry was an unhandled cleanup obligation or improper deallocation. A missing local release, however, may reflect a broader resource lifecycle rather than a bug, most commonly when the object or a derived pointer is published through a caller-visible location such as a linked list, subsystem registry, or asynchronous callback. We treat such publication as evidence of ownership hand-off only when it is not rolled back before the path exits. This criterion confirms 75 ownership transfers. In another 13 reports, the reviewers could not conclusively determine whether ownership had

been transferred; under our precision-oriented policy, we conservatively attributed them to ownership transfer and counted them as false positives. These 88 reports are included among the 319 false positives rather than the 145 real bugs, with all 464 reports retained in the denominator. The resulting precision is 31.3%, with a false discovery rate of 68.8%. This is practical for an expert-assisted analyzer, since TriMatch provides allocation sites, normalized access paths, and complete call chains to support rapid triage.

Root causes of false positives. We categorize the 319 false positives into four primary technical causes:

- **Lifecycle recognition (126 cases, 39.5%):** TriMatch may expect local cleanup even when a resource is governed by a broader lifecycle. This category includes the 88 reports that we conservatively classified as false positives for ownership-transfer reasons. Of the remaining 38 reports, 28 follow allocation failures with no live object to release, 5 rely on callee-side cleanup after failure, and 5 involve reference-counting distinctions, intentional retention, or other lifecycle-specific semantics.
- **Alias imprecision (76 cases, 23.8%):** Because our alias analysis is flow- and array-index-insensitive, TriMatch may merge values separated by later updates or conflate distinct array elements as the same logical object.
- **Error-path recognition (61 cases, 19.1%):** TriMatch may misidentify cleanup behavior across error-handling paths or combine paths that cannot execute together. This causes 37 reports on misidentified error-handling paths and 24 on mutually exclusive or unreachable paths.
- **Incorrect wrapper pairing (56 cases, 17.6%):** These arise when TriMatch establishes an incorrect allocation-to-release correspondence across wrappers. Complex control flow or long call chains that keep the true counterpart out of the corpus can shift the inferred symmetry boundary, so TriMatch may pass an unrelated match to the 3D checks or flag a valid match as a mismatch.

Recall analysis. Since no ground-truth bug benchmark exists for the full kernel, we measure recall on a CVE replay benchmark built from CVE-confirmed memory bugs published after 6.13-rc1 whose affected version ranges still cover this snapshot, ensuring every bug was present in the analyzed code. We first excluded CVEs whose root cause depends on runtime dynamic behavior beyond static analysis, such as concurrency races or data-dependent branch conditions; keeping only the remaining bugs that match TriMatch’s three-dimensional detection scope left 55 bugs, of which TriMatch detected 27, for a recall lower bound of 49.1%. We root-caused the 28 missed bugs into five categories:

- **Absent local allocation site.** This accounts for 35.7% of the missed bugs. In 8 of the 10 cases, the function under analysis contains no allocation of the object at all; the resource is allocated by a caller or obtained from elsewhere, such as dequeued from a list or received through the network stack, so no alloc/free candidate is formed within the analyzed scope. In the remaining 2 cases, the allocator exists locally but is hidden behind an indirect call or is an unrecognized map-style allocation.

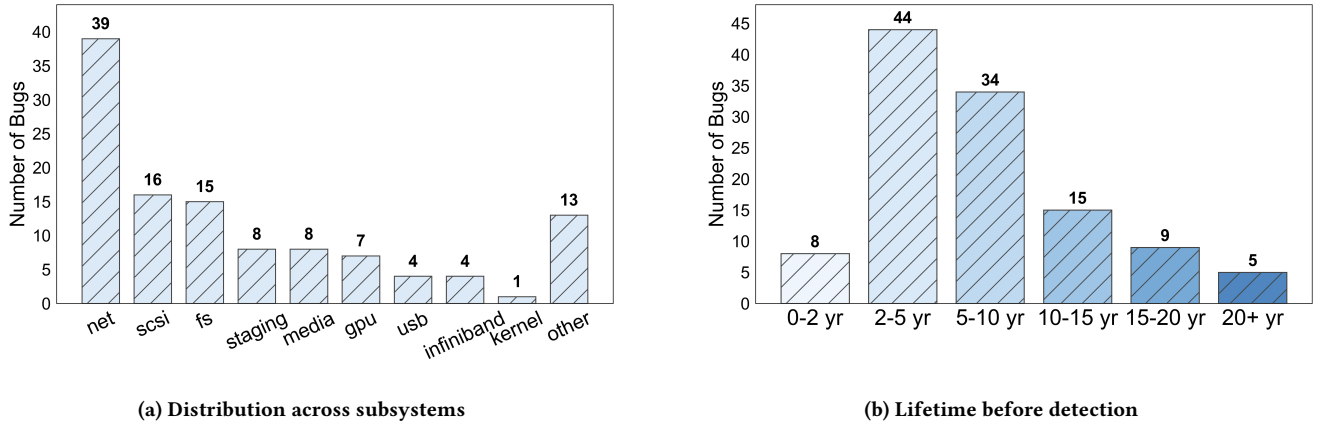


Figure 11: Characterization of the 115 newly discovered bugs.

- **Error-path recognition.** This accounts for 21.4% of the missed bugs, where the cleanup bug sits in a branch outside the recognized error-handling patterns, such as direct returns with non-negative values, failure paths defined inside macros, or switch branches.
- **Conservative ownership-transfer classification.** This accounts for 17.9% of the missed bugs, where the resource was conservatively judged as handed off to a caller-visible location under our precision-oriented criterion, although the subsequent code never discharged the cleanup obligation.
- **Alias imprecision.** This accounts for 17.9% of the missed bugs, caused by cross-function aliasing and the inability to distinguish different array indices.
- **Release-pairing precision.** This accounts for 7.1% of the missed bugs, in which TriMatch could not distinguish semantically different variants of the same release function, such as `kfree` and `kfree_sensitive`. The appropriate variant depends on the context of the allocation, and the pairing treats them coarsely as equivalent.

Four of these categories stem from imprecision in allocation, error-path, and pairing recognition, while the ownership-transfer category reflects the deliberate precision/recall trade-off of our conservative criterion.

7.3 RQ3: Effectiveness of Core Design Choices

We evaluate each core design choice by (i) validating the recall of the `container_of` recognizer and (ii) measuring the impact of removing individual components from the full system.

Recall of `container_of` recognition. The bidirectional tracking in TriMatch relies on correctly identifying `container_of` patterns in LLVM IR. To quantify the recognizer, we wrote a script that identifies every `container_of` use point, including those reached through wrapper macros. We treat the resulting set as the kernel-wide candidate set over Linux 6.13-rc1. Across 100 randomly sampled C files, we manually identified 452 valid use cases, of which TriMatch recognized 407, for a recall of 90.0% (407/452). We manually inspected all 407 recognized cases and confirmed each as valid, with

no false positives in this sample. Kernel-wide, the recognizer identifies 64,194 of 68,784 use points, corresponding to a recognition rate of 93.3%. We categorized the 45 missed use points in the manually validated sample into four causes:

- **Void types.** This accounts for 42.2% of the missed use points, where the pointer passed to `container_of` has void type, defeating the syntactic type constraints of the pattern.
- **Backtracking through array offsets.** This accounts for 35.6% of the missed use points, where the backtracking first traverses an array index whose IR offset is non-constant, preventing the fixed-offset validation from resolving the enclosing object.
- **Multi-level nested structures.** This accounts for 17.8% of the missed use points, where the object is recovered by backtracking through multiple nested struct levels.
- **Uncompiled code.** This accounts for 4.4% of the missed use points, where the use point lies in code that is not compiled in the analyzed configuration and is therefore absent from the bytecode.

These results show that syntactic GEP constraints and DataLayout-based offset validation accurately distinguish backward object recovery from ordinary field access in the inspected sample. The remaining misses arise when type or offset information is unavailable, recovery spans multiple nested structures, or the relevant code is absent from the analyzed bytecode.

Ablation study. Table 3 reports the effect of removing individual design choices from the full TriMatch configuration. Because absolute recall is unavailable on Linux, we report relative recall against the 145 real bugs found by the full configuration. The Horizontal and Vertical dimensions remain active in all configurations unless stated otherwise.

- **Allocation-wrapper filtering.** Removing it inflates reports from 464 to 669 and drops relative recall to 95.9%, because spurious allocation wrappers enter the candidate corpus and a misidentified later allocation for the same object prematurely terminates the analysis of the earlier one, hiding the missing cleanup.

Table 3: Ablation study: effect of removing individual design choices.

Config.	Reports	Real	Prec. (%)	Rel. recall (%)
Full TriMatch	464	145	31.3	100.0
w/o wrapper filtering	669	139	20.8	95.9
w/o corpus expansion	545	134	24.6	92.4
w/o back. sum.	428	132	30.8	91.0
w/o multi-pair	3151	109	3.5	75.2
w/o back. sum. & multi-pair	2747	108	3.9	74.5

```

1 static tx_desc *ls1x_dma_prep_dma_cyclic(...) {
2     struct ls1x_dma_desc *desc = ls1x_dma_alloc_desc();
3     if (!desc)
4         return NULL;
5
6     sgl = kcalloc_array(...);
7     if (!sgl)
8         return NULL; // missing ls1x_dma_free_desc(&desc->vd)
9
10    if (ls1x_dma_prep_lll(...)) {
11        ls1x_dma_free_desc(&desc->vd);
12        return NULL;
13    }
14    ...
15 }
16
17 static struct ls1x_dma_desc *ls1x_dma_alloc_desc(void) {
18     return kzalloc(sizeof(struct ls1x_dma_desc), ...);
19 }
20
21 static void ls1x_dma_free_desc(struct virt_dma_desc *vd) {
22     struct ls1x_dma_desc *desc = to_ls1x_dma_desc(vd);
23     kfree(desc);
24 }

```

Figure 12: Simplified memory leak in `ls1x_dma_prep_dma_cyclic()` that requires both backward summaries and multi-layer pairing to detect.

- **Pairing corpus expansion.** Removing it increases reports from 464 to 545 and lowers real bugs from 145 to 134, cutting precision from 31.3% to 24.6% and relative recall to 92.4%. The simultaneous drop in both metrics shows that the additional valid pairings benefit precision and recall alike.
- **Bidirectional summaries.** Removing backward summaries lowers real bugs from 145 to 132, with reports changing only slightly from 464 to 428, for a relative recall of 91.0%. In Figure 12, `desc` is allocated at line 2 but the only release witness is `ls1x_dma_free_desc(&desc->vd)` at line 11, which recovers `desc` from `vd` and calls `kfree(desc)`; without this backward recovery step, TriMatch cannot connect the cleanup to the allocation and misses that the failure branch at lines 7–8 should also free `desc`.
- **Multi-layer pairing.** Removing it increases reports from 464 to 3151 and lowers real bugs from 145 to 109, reducing precision from 31.3% to 3.5% and relative recall to 75.2%. In the same example, the allocated value is `desc` while the free-side argument is `&desc->vd`, which do not directly

Table 4: Baseline coverage of validated TriMatch findings.

Baseline	Reference real bugs	Baseline reports matching reference	Reference bugs not matched
IMMI	145	12	133
APISpecGen	145	1	144
Knighter	145	0	145

alias; multi-layer pairing combines summary and call-chain information to recover the object-level link to the eventual `kfree(desc)`. Without it, TriMatch again misses that the `kmalloc_array()` failure branch should free `desc`, and the resulting mispairing produces an approximately 9.5-fold surge in false positives, showing that multi-layer pairing is essential for precision, not only for recall.

- **Joint removal of backward summaries and multi-layer pairing.** Removing both together yields 2747 reports and 108 real bugs with a relative recall of 74.5%, nearly identical to removing multi-layer pairing alone. On the recall side, the joint removal loses only one more real bug than removing multi-layer pairing alone, confirming that once multi-layer pairing is gone, most of the object-level benefit of backward summaries is already lost, and their remaining value lies mainly in building more complete release wrappers. On the report side, the counts show a clear asymmetry: removing backward summaries alone from Full TriMatch trims just 36 reports, whereas once multi-layer pairing is already removed it trims back 404 reports. This asymmetry further highlights the role of multi-layer pairing in suppressing false positives.
- **Residual recall.** The residual recall is supported by two improvements retained in the ablated configurations: enhanced error-handling path identification and stronger Horizontal and Vertical symmetry checks. Since these components underpin multiple aspects of the analysis, their individual contributions are difficult to isolate through ablation of a single stage.

7.4 RQ4: Comparison with Existing Tools

To assess complementary coverage, we evaluate three state-of-the-art tools: IMMI [23] (summary-based), APISpecGen [22] (pairing-based), and Knighter [38] (LLM-generated checkers). Table 4 and Table 5 separate the comparison into two directions. First, each baseline runs on TriMatch’s Linux 6.13-rc1 version, using the 145 manually validated bugs from TriMatch’s reports as the reference set. Second, TriMatch runs on each baseline’s evaluation version, using that baseline’s manually validated real bugs as the reference set. In both directions, we manually count a report as a match only when it corresponds to a reference bug in both location and root cause.

How well the three tools detect our bugs. To understand why the baselines detect so few of TriMatch’s validated bugs (Table 4), we examine each tool in turn.

- **IMMI.** IMMI detects inconsistencies between callers’ and callees’ memory-management intentions from on-demand

Table 5: TriMatch coverage of baseline-validated findings.

Baseline	Reference real bugs	TriMatch reports matching reference	Reference bugs not matched by TriMatch
IMMI	80	11	69
APISpecGen	100	1	99
Knighter	92	0	92

summaries, yet detects only 12 of our bugs. Three reasons explain this. Its summary representation lacks backward structure recovery, so it loses track of objects undergoing such transitions. Its data-flow-based check of whether an allocation is canceled by a free cannot capture pairing mismatches where the surface data flow holds but the cleanup semantics do not. And its intraprocedural reasoning relies exclusively on symmetry among error-handling branches, ignoring the normal exit path that serves as a baseline for comparison.

- **APISpecGen.** APISpecGen propagates seed APIs through interprocedural call chains to identify paired functions, but found only 1 of our bugs. This limited coverage stems from its function-level pairing, which does not validate operations against specific nested resources, allowing surface-level matches to hide deeper semantic mismatches.
- **Knighter.** Knighter learns patterns from historical patches and synthesizes static checkers, but found none of our bugs. Such patterns may miss unknown or underrepresented vulnerabilities, while LLMs struggle with cross-call resource lifetimes and the control-flow reasoning needed for checker synthesis.

How well TriMatch detects the three tools' bugs. TriMatch matches 11 of IMMI's 80 reference bugs and 1 of APISpecGen's 100, but none of Knighter's 92 (Table 5). Beyond the false negatives quantified in subsection 7.2, this gap reflects three factors. First, some reference bugs lack a comparable witness path, so no post-allocation branch provides a release witness and the Horizontal check cannot establish a baseline expectation. Second, some bugs are defects in the implementation of an API itself rather than misuse of an alloc/free API, which falls outside TriMatch's model. Third, others span multi-level call chains that TriMatch does not traverse when linking an allocation to its release.

Complementary bug models. The limited overlap reflects differences in the violations modeled by these tools. TriMatch targets 3D object-aware alloc/free asymmetry, whereas the baselines focus on memory-management intentions, inferred API pairings, or patch-derived bug patterns. Conversely, some baseline findings lack the witness paths or alloc/free structure required by TriMatch. The tools therefore provide complementary coverage rather than redundant implementations of the same bug model.

7.5 RQ5: Full-Kernel Analysis Scalability

Table 6 breaks down the analysis time. TriMatch completes the full-kernel analysis in 1 hour and 41 minutes, with symmetry bug

Table 6: Time cost of TriMatch.

Stage	Time (s)	Share (%)
Load	99	1.6
Call graph building	278	4.6
Error-handling path identification	123	2.0
Summary construction and function-pair inference	342	5.6
Symmetry bug detection	5214	86.1
Total	6056	100.0

detection dominating at 86.1% of the total, since the depth, horizontal, and vertical checks for each candidate require call-graph traversal and control-flow analysis. Call-graph building, error-handling path identification, and summary construction together account for about 12.3%. The total cost is reasonable given the comprehensiveness and accuracy of the analysis.

8 Discussion

Generalizing to other resource pairs. TriMatch targets alloc/free asymmetry, but the same 3D object-aware symmetry checking could extend to other bug types with similar symmetry properties, such as lock/unlock mismatches; adapting the core techniques to these domains is future work.

Limitation: context-specific pairing. TriMatch treats any established pairing as valid, but some allocation functions have multiple legitimate release counterparts depending on context. For instance, `kzalloc` can be paired with either `kfree` or `kfree_sensitive`; while `kfree` is correct in most cases, memory holding cryptographic keys or other sensitive data requires `kfree_sensitive` to erase the contents before deallocation. TriMatch currently over-approximates this relationship and does not analyze data semantics to enforce the stricter counterpart, an important direction for future work.

9 Related Work

Function pairing and specification inference. Previous work has shown that pairing- or specification-based reasoning is a practical way to avoid full object-lifetime tracking on large code bases. Early paired-function mining relied mainly on lexical or co-usage evidence [27, 28]. Later systems improved this direction through seeded pair harvesting [3], error-handling-aware pairing [36], patch-derived post-handling rules [21], and bidirectional specification propagation across API hierarchies [22]. Differential inconsistency checking has also been used to detect security bugs by comparing structurally similar paths [24]. These techniques are effective at inferring likely cleanup counterparts or post-handling obligations, but they stop at candidate generation or rule inference. They do not answer the question we target here: whether the release-side wrapper chain is actually symmetric to the alloc-side chain for the same logical object.

Kernel memory bug detection. Another line of work designs bug-specific abstractions for kernel memory and lifetime errors. K-MELD uses ownership reasoning to detect leaks in specialized modules [7]. Goshawk introduces object-centric memory-operation

summaries to improve scalability for custom memory management [30]. IMMI detects bugs through inconsistencies in inferred memory-management intentions rather than direct whole-program tracking [23]. IncreLux incrementally detects UBI bugs after kernel updates [42]. UAFX extends static reasoning to cross-entry UAFs with dedicated alias analysis and validation constraints [44]. Systematic static analysis has also been used to target error-handling bugs through cross-control-flow taint reasoning [43]. Compared with these systems, TriMatch is not built around one specific bug oracle such as leak, intention inconsistency, incremental UBI detection, error-handling misuse, or cross-entry UAF. Instead, it focuses on validating object-level alloc/free symmetry itself, combining wrapper-aware pairing with bidirectional summaries and checking the resulting allocation and release behavior along the depth, horizontal, and vertical dimensions.

LLM-assisted bug detection and program analysis. Recent work has explored LLMs for vulnerability discovery in large systems. One line of research uses LLMs to assist fuzzing (KernelGPT [39]) or to mine bug patterns from historical fixes (BUGSTONE [37]). Another line integrates LLMs into static analysis: LLift [18] targets UBI bugs, BugLens [19] reduces false positives in taint analysis, and LAMeD [33] generates function-level memory leak annotations. Most relevant is KNightier [38], which synthesizes checkers from historical patches, yet our evaluation shows it finds none of the validated findings reported by TriMatch. Vul-RAG [6] augments LLM detection with retrieved knowledge, and VulnLLMEval [45] provides a benchmark of 307 real kernel vulnerabilities. These approaches share a common limitation: they lack precise, object-level tracking of resources across container_of transformations, wrapper chains, and nested fields, the core requirement for validating alloc/free symmetry in the Linux kernel, which TriMatch addresses.

10 Conclusion

In this paper, we presented TriMatch, an LLVM-based static analyzer that detects Linux kernel memory bugs through 3D object-aware symmetry checking over bidirectional memory-operation summaries and wrapper-aware pairing. Our evaluation on Linux 6.13-rc1 shows that TriMatch scales to whole-kernel analysis, completing in 1 hour and 41 minutes while finding 145 real bugs, including 115 previously unknown ones. All 115 previously unknown bugs have been reported; 80 fixes have been merged into mainline, 14 bugs have been confirmed by maintainers, and 21 remain under review. 42 CVE IDs have been assigned to the reported bugs. The ablation results further show that backward summaries and multi-layer pairing are both important to effectiveness, while the version-specific baseline comparison provides evidence of complementary coverage between TriMatch and existing approaches. Overall, these results show that 3D object-aware symmetry checking is an effective and practical direction for large-scale kernel memory-bug detection.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported, in part, by grants from the Chinese National Natural Science Foundation (62402107, U22B2025, 62172201, 62572231), the Start-up Research Fund of Southeast University and the Big Data

Computing Center of Southeast University, the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118). Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

- [1] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. 2020. Effective Detection of Sleep-inatomic-context Bugs in the Linux Kernel. *ACM Transactions on Computer Systems* 36, 4, Article 10 (2020), 30 pages. doi:10.1145/3381990
- [2] Jia-Ju Bai, Julia Lawall, Wende Tan, and Shi-Min Hu. 2019. DCNS: Automated Detection Of Conservative Non-Sleep Defects in the Linux Kernel. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 287–299. doi:10.1145/3297858.3304065
- [3] Pan Bian, Bin Liang, Jianjun Huang, Wenchang Shi, Xidong Wang, and Jian Zhang. 2020. SinkFinder: Harvesting Hundreds of Unknown Interesting Function Pairs with Just One Seed. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1101–1113. doi:10.1145/3368089.3409678
- [4] Yuandao Cai, Peisen Yao, Chengfeng Ye, and Charles Zhang. 2023. Place Your Locks Well: Understanding and Detecting Lock Misuse Bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, USA, 3727–3744. <https://www.usenix.org/conference/usenixsecurity23/presentation/cai-yuandao>
- [5] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, Piscataway, NJ, USA, 4661–4677. doi:10.1109/SP54263.2024.00269
- [6] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2026. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. *ACM Transactions on Software Engineering and Methodology* (Feb. 2026), 25 pages. doi:10.1145/3797277 Published online 20 February 2026.
- [7] Navid Emamdoust, Qiushi Wu, Kangjie Lu, and Stephen McAmant. 2021. Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning. In *Network and Distributed System Security Symposium (NDSS) 2021*. The Internet Society, Reston, VA, USA, 18 pages. doi:10.14722/ndss.2021.24416
- [8] Sishuai Gong, Dinglan Peng, Deniz Altunbükten, Pedro Fonseca, and Petros Maniatis. 2023. Snowcat: Efficient Kernel Concurrency Testing using a Learned Coverage Predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 35–51. doi:10.1145/3600006.3613148
- [9] Sishuai Gong, Rui Wang, Deniz Altunbükten, Pedro Fonseca, and Petros Maniatis. 2025. Snowflow: Effective Kernel Fuzzing with a Learned White-box Test Mutator. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1124–1138. doi:10.1145/3676641.3716019
- [10] Priya Govindasamy, Joseph Bursey, Hsin-Wei Hung, and Ardan Amiri Sani. 2025. Spinner: Detecting Locking Violations in the eBPF Runtime. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Piscataway, NJ, USA, 2171–2183. doi:10.1109/ASE63991.2025.00180
- [11] Jinyu Gu, Jiacheng Shi, Haroran Su, Wentai Li, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Understanding and Mitigating Twin Function Misuses in Operating System Kernel. *IEEE Trans. Comput.* 72, 8 (2023), 2181–2193. doi:10.1109/TC.2023.3240365
- [12] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, Piscataway, NJ, USA, 3262–3278. doi:10.1109/SP46215.2023.10179298
- [13] Dae R. Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. SegFuzz: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, Piscataway, NJ, USA, 2104–2121. doi:10.1109/SP46215.2023.10179398
- [14] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. In *Network and Distributed System Security Symposium (NDSS) 2022*. The Internet Society, Reston, VA, USA, 18 pages. doi:10.14722/ndss.2022.24296
- [15] Kyungtae Kim, Taeyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin R. B. Butler, Antonio Bianchi, and Dave Jing Tian. 2022. FuzzUSB: Hybrid Stateful Fuzzing of

- USB Gadget Stacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, Piscataway, NJ, USA, 2212–2229. doi:10.1109/SP46214.2022.9833593
- [16] Jakob Koschel, Pietro Borrello, Daniele Cono D'Elia, Herbert Bos, and Cristiano Giuffrida. 2023. Uncontained: Uncovering Container Confusion in the Linux Kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, USA, 5055–5072. <https://www.usenix.org/conference/usenixsecurity23/presentation/koschel>
- [17] Baptiste Lepers, Josselin Giet, Willy Zwaenepoel, and Julia Lawall. 2023. OFence: Pairing Barriers to Find Concurrency Bugs in the Linux Kernel. In *Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 33–45. doi:10.1145/3552326.3567504
- [18] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1, Article 111 (2024), 26 pages. doi:10.1145/3649828
- [19] Haonan Li, Hang Zhang, Kexin Pei, and Zhiyun Qian. 2025. Towards More Accurate Static Analysis for Taint-Style Bug Detection in Linux Kernel. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Piscataway, NJ, USA, 380–392. doi:10.1109/ASE63991.2025.00039
- [20] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2024. SPATA: Effective OS Bug Detection with Summary-Based, Alias-Aware, and Path-Sensitive Typestate Analysis. *ACM Transactions on Computer Systems* 42, 3–4 (2024), 1–40. doi:10.1145/3695250
- [21] Miaoqian Lin, Kai Chen, and Yang Xiao. 2023. Detecting API Post-Handling Bugs Using Code and Description in Patches. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, USA, 3709–3726. <https://www.usenix.org/conference/usenixsecurity23/presentation/lin>
- [22] Miaoqian Lin, Kai Chen, Yi Yang, and Jinghua Liu. 2025. Uncovering the Iceberg from the Tip: Generating API Specifications for Bug Detection via Specification Propagation Analysis. In *Network and Distributed System Security Symposium (NDSS) 2025*. The Internet Society, Reston, VA, USA, 17 pages. doi:10.14722/ndss.2025.242001
- [23] Dinghao Liu, Zhipeng Lu, Shouling Ji, Kangjie Lu, Jianhai Chen, Zhengguang Liu, Dexin Liu, Renyi Cai, and Qinming He. 2024. Detecting Kernel Memory Bugs through Inconsistent Memory Management Intention Inferences. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, USA, 4069–4086. <https://www.usenix.org/conference/usenixsecurity24/presentation/liu-dinghao-detecting>
- [24] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhengguang Liu, Jianhai Chen, and Qinming He. 2021. Detecting Missed Security Operations Through Differential Checking of Object-Based Similar Paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 1627–1644. doi:10.1145/3460120.3485373
- [25] Haoran Liu, Zhouyang Jia, Shanshan Li, Yan Lei, Yue Yu, Yu Jiang, Xiaoguang Mao, and Xiangke Liao. 2024. Cut to the Chase: An Error-Oriented Approach to Detect Error-Handling Bugs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1796–1818. doi:10.1145/3660787
- [26] Haoran Liu, Shanshan Li, Zhouyang Jia, Yuanliang Zhang, Linxiao Bai, Si Zheng, Xiaoguang Mao, and Xiangke Liao. 2025. Error Delayed Is Not Error Handled: Understanding and Fixing Propagated Error-Handling Bugs. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2571–2593. doi:10.1145/3729384
- [27] Hu-Qiu Liu, Jia-Ju Bai, Yu-Ping Wang, Zhe Bian, and Shi-Min Hu. 2015. PairMiner: Mining for Paired Functions in Kernel Extensions. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Piscataway, NJ, USA, 93–101. doi:10.1109/ISPASS.2015.7095788
- [28] Hu-Qiu Liu, Yu-Ping Wang, Jia-Ju Bai, and Shi-Min Hu. 2016. PF-Miner: A Practical Paired Functions Mining Method for Android Kernel in Error Paths. *Journal of Systems and Software* 121 (2016), 234–246. doi:10.1016/j.jss.2016.02.007
- [29] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1769–1786. <https://www.usenix.org/conference/usenixsecurity19/presentation/lu>
- [30] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. 2022. Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, Piscataway, NJ, USA, 2096–2113. doi:10.1109/SP46214.2022.9833613
- [31] Lin Ma, Duoming Zhou, Hanjie Wu, Yajin Zhou, Rui Chang, Hao Xiong, Lei Wu, and Kui Ren. 2023. 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, Piscataway, NJ, USA, 2138–2154. doi:10.1109/SP46215.2023.10179356
- [32] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. 2024. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, USA, 4051–4068. <https://www.usenix.org/conference/usenixsecurity24/presentation/maar-slubstick>
- [33] Ekaterina Shemetova, Ivan Smirnov, Anton Alekseev, Ilya Shenbin, Alexey Rukhovich, Sergey Nikolenko, Vadim Lomshakov, and Irina Piontkovskaya. 2025. LAMeD: LLM-generated Annotations for Memory Leak Detection. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1024–1034. doi:10.1145/3756681.3756999
- [34] Yue Sun, Yan Kang, Chenggang Wu, Kangjie Lu, Jiming Wang, Xingwei Li, Yuhao Hu, Jikai Ren, Yuanming Lai, Mengyao Xie, and Zhe Wang. 2025. SyzParam: Incorporating Runtime Parameters into Kernel Driver Fuzzing. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 1484–1498. doi:10.1145/3719027.3744838
- [35] Keita Suzuki, Kenta Ishiguro, and Kenji Kono. 2024. Balancing Analysis Time and Bug Detection: Daily Development-friendly Bug Detection in Linux. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, USA, 493–508. <https://www.usenix.org/conference/atc24/presentation/suzuki>
- [36] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2041–2058. <https://www.usenix.org/conference/usenixsecurity21/presentation/wu-qiushi>
- [37] Qiushi Wu, Yue Xiao, Dhillung Kirat, Kevin Eykholt, Jiyong Jang, and Douglas Lee Schales. 2025. One Bug, Hundreds Behind: LLMs for Large-Scale Bug Discovery. arXiv:2510.14036 [cs.SE] <https://arxiv.org/abs/2510.14036>
- [38] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. 2025. KNIghter: Transforming Static Analysis with LLM-Synthesized Checkers. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 655–669. doi:10.1145/3731569.3764827
- [39] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLoS '25)*. Association for Computing Machinery, New York, NY, USA, 560–573. doi:10.1145/3676641.3716022
- [40] Chengfeng Ye, Yuandao Cai, and Charles Zhang. 2024. When Threads Meet Interrupts: Effective Static Detection of Interrupt-Based Deadlocks in Linux. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, USA, 6167–6184. <https://www.usenix.org/conference/usenixsecurity24/presentation/ye>
- [41] Ming Yuan, Bodong Zhao, Penghui Li, Jiahuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, USA, 2849–2866. <https://www.usenix.org/conference/usenixsecurity23/presentation/yuan-ming>
- [42] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul Yu. 2022. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *Network and Distributed System Security Symposium (NDSS) 2022*. The Internet Society, Reston, VA, USA, 18 pages. doi:10.14722/ndss.2022.24380
- [43] Dongyang Zhan, Xiangzhan Yu, Hongli Zhang, and Lin Ye. 2023. ErrHunter: Detecting Error-Handling Bugs in the Linux Kernel Through Systematic Static Analysis. *IEEE Transactions on Software Engineering* 49, 2 (2023), 684–698. doi:10.1109/TSE.2022.3160155
- [44] Hang Zhang, Jangha Kim, Chuhong Yuan, Zhiyun Qian, and Taesoo Kim. 2025. Statically Discover Cross-Entry Use-After-Free Vulnerabilities in the Linux Kernel. In *Network and Distributed System Security Symposium (NDSS) 2025*. The Internet Society, Reston, VA, USA, 17 pages. doi:10.14722/ndss.2025.240559
- [45] Arastoo Zibaeirad and Marco Vieira. 2024. VulnLLMEval: A Framework for Evaluating Large Language Models in Software Vulnerability Detection and Patching. arXiv:2409.10756 [cs.SE] doi:10.48550/arXiv.2409.10756

A Open Science

To support transparency and reproducibility, we provide the artifacts needed to evaluate the core contributions of TriMatch at <https://doi.org/10.5281/zenodo.22744046>.

The review artifact includes:

- The TriMatch analyzer source code.
- List of disclosed bugs.
- The complete C API seed corpus, role-specific keyword lexicons, their curation methodology, and the pairings inferred by TriMatch.

B Ethical Considerations

Our work aims to improve Linux kernel security by detecting memory-management bugs in publicly available source code. The main beneficiaries are kernel developers, downstream distributors, and end users, who may benefit from earlier detection and remediation of long-lived bugs.

This work is dual use: the same analysis that helps defenders may also expose vulnerabilities before they are fixed. We therefore rigidly follow responsible disclosure. For all 115 previously unknown bugs, we actively assist the community by developing, testing, and submitting a corresponding patch through the established Linux kernel channels. To date, 80 fixes have been merged into mainline, 14 bugs have been confirmed by maintainers, and 21 remain under review. 42 CVE IDs have been assigned to the reported bugs. Furthermore, our work focuses strictly on static bug detection and patch development; it does not involve generating, utilizing, or releasing any exploits. We do not publicly disclose bug details in ways inconsistent with responsible disclosure principles. Sensitive details of unfixed bugs remain confidential before maintainer review. The CVE assignments followed standard vulnerability-reporting procedures after the corresponding patches were accepted and merged.

This work does not involve human participants, personal data, or private user traces. Our analysis uses public Linux kernel source code, public patch history, and artifacts generated by our own experiments.

C Generative AI Usage

We used generative AI tools to assist in refining some content and making limited language edits. All AI-assisted content was critically reviewed, revised, and verified by the authors. The authors take full responsibility for the accuracy, originality, and integrity of the final manuscript. All factual claims, citations, and experimental results were checked by the authors.

Table 7: Assigned CVEs and associated buggy functions. Functions sharing one CVE ID are separated by commas.

Buggy function	Type	CVE ID
ieee80211_register_hw	Memory-leak	CVE-2026-72004
otx2_init_hw_resources	Memory-leak	CVE-2026-72023
itd_submit (fotg210-hcd.c), itd_submit (ehci-sched.c), sitd_submit (ehci-sched.c)	Memory-leak	CVE-2026-64348
hws_bwc_matcher_move	Memory-leak	CVE-2026-72032
xfs_dqinode_metadir_create	Memory-leak	CVE-2026-64356
helper_firmware_cb	Memory-leak	CVE-2026-68410
create_queues_with_size_backoff	Memory-leak	CVE-2026-72056
proc_sys_call_handler	Misuse	CVE-2026-63809
qed_cxt_tables_alloc	Double-free	CVE-2026-64118
otx2_sq_aura_pool_init, otx2_rq_aura_pool_init	Memory-leak / Double-free	CVE-2026-64222
rvu_rep_rsrc_init	Double-free	CVE-2026-64224
meson_crypto_probe	UAF	CVE-2026-64599
do_rbd_add	NPD	CVE-2026-46079
xhci_disable_slot	Memory-leak	CVE-2026-43432
pqi_report_phys_luns	Memory-leak	CVE-2026-45872
ath11k_mac_setup_bcn_tmpl_mbssid	Memory-leak	CVE-2026-53113
starfive_aes_aead_do_one_req	Memory-leak	CVE-2026-45950
amdgpu_acpi_enumerate_xcc	Memory-leak	CVE-2026-45947
amdgpu_gmc_get_nps_memranges	Misuse	CVE-2026-45853
amdgpu_ras_init	Memory-leak	CVE-2026-45976
setup_nic_devices	Memory-leak	CVE-2026-23258
dw_i3c_master_i2c_xfers	Memory-leak	CVE-2026-45863
mlx5_ib_handler_MLX5_IB_METHOD_GET_DATA_DIRECT_SYFS_PATH	Memory-leak	CVE-2026-45922
raid1_run	Memory-leak	CVE-2026-45888
mtdd_parser_tplink_safeloader_parse	Memory-leak	CVE-2026-45921
ice_set_ringparam	Memory-leak	CVE-2026-23389
octep_device_setup	Memory-leak	CVE-2026-23160
mt76_connac_mcu_sta_ba, mt76_connac_mcu_add_key, mt7915_mcu_add_sta, mt7925_mcu_add_key	Memory-leak	CVE-2026-53102
wbrf_record	Memory-leak	CVE-2026-23065
unittest_data_add	Memory-leak	CVE-2026-23137
ni_usb_init	Memory-leak	CVE-2026-45939
svs_enable_debug_write	Memory-leak	CVE-2026-45881
mpfs_sys_controller_probe	Memory-leak	CVE-2026-23464
ext4_ext_shift_extents	Memory-leak	CVE-2026-45948
nfs4_ff_alloc_deviceid_node	Memory-leak	CVE-2026-23038
smb3_reconfigure	Memory-leak	CVE-2025-71151
get_file_all_info	Memory-leak	CVE-2025-71153
__tegra_channel_try_format	Memory-leak	CVE-2026-43162
qla2xxx_process_purls_ioch	Misuse	CVE-2025-68741
mt7615_mcu_wtbl_sta_add	Memory-leak	CVE-2025-68765
wave5_vpu_open_dec, wave5_vpu_open_enc	Memory-leak	CVE-2026-45928
bm_register_write	Misuse	CVE-2025-68239