UNCONTAINED: Uncovering Container Confusion in the Linux Kernel

Jakob Koschel, *Vrije Universiteit Amsterdam;* Pietro Borrello and Daniele Cono D'Elia, *Sapienza University of Rome;* Herbert Bos and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

https://www.usenix.org/conference/usenixsecurity23/presentation/koschel

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

UNCONTAINED: Uncovering Container Confusion in the Linux Kernel

Jakob Koschel†
Vrije Universiteit Amsterdam
j.koschel@vu.nl

Pietro Borrello†
Sapienza University of Rome
borrello@diag.uniroma1.it

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

† Equal contribution joint first authors

Abstract

Type confusion bugs are a common source of security problems whenever software makes use of type hierarchies, as an inadvertent downcast to an incompatible type is hard to detect at compile time and easily leads to memory corruption at runtime. Where existing research mostly studies type confusion in the context of object-oriented languages such as C++, we analyze how similar bugs affect complex C projects such as the Linux kernel. In particular, structure embedding emulates type inheritance between typed structures. Downcasting in such cases consists of determining the containing structure from the embedded one, and, like its C++ counterpart, may well lead to *bad casting* to an incompatible type.

In this paper, we present UNCONTAINED, a systematic, two-pronged solution to discover type confusion vulnerabilities resulting from incorrect downcasting on structure embeddings—which we call *container confusion*. First, we design a novel sanitizer to dynamically detect such issues and evaluate it on the Linux kernel, where we find as many as 11 container confusion bugs. Using the patterns in the bugs detected by the sanitizer, we then develop a static analyzer to find similar bugs in code that dynamic analysis fails to reach and detect another 78 bugs. We reported and proposed patches for all the bugs (with 102 patches already merged and 6 CVEs assigned), cooperating with the Linux kernel maintainers towards safer design choices for container manipulation.

1 Introduction

Complex software often makes use of class and type hierarchies to achieve modularity in the design and favor code reuse for operations meant to work on similar objects. Interestingly, this phenomenon is not exclusive to software written in object-oriented languages. One compelling case involves the C language, as implementors of kernels and large userland applications commonly resort to custom means, namely *structure embedding*, to model inheritance between typed structures. In the lack of explicit language provisions, the validity of casting operations becomes an implicit assumption from code semantics (i.e., on implementation correctness).

Structure embedding operates by declaring an instance of a more general typed structure (the parent) as a field of a more specific one (the child). A well-known example is the `list_head` structure in the Linux kernel. In this paper, we will sometimes refer to such structures as *objects*. Code that needs to access the more general representation of an object, thus realizing an *upcast*, will simply use the member field for the parent in the object. This operation is intuitively safe. Code that needs to access a more specialized representation of an object, thus realizing a *downcast*, will (unsafely) manipulate the parent pointer to recover the address of the child.

In more detail, an object downcast subtracts the offset of the parent field in the child object from the address available for the parent, yielding the address of its *container* structure (i.e., the child). The term container follows from the popular `container_of` macro pioneered by the Linux kernel. Issuing a downcast is not only always unsafe, but even not conforming to any C language standard [43]. Thus, the correctness and safety burden is on the shoulder of the developers, who have to guarantee through program semantics that the requested child type is correct. Failing to meet this requirement would cause a type confusion, which may have possibly disastrous consequences, such as a memory corruption vulnerability [39].

For object-oriented languages, runtime type information (RTTI) enables straightforward validation of downcasting operations. For example, current solutions that look for type confusion in C++ code rely on forms of RTTI tracking [13, 15, 21, 31]. Solutions with provisions for C code can detect (some) cases of type confusion by intercepting heap allocations of objects and binding them with their top-level allocation type [13, 31] in userland code. Automatic type identification is difficult in C programs due explicit/implicit unions, pointer casting, allocation wrappers, and other factors as shown in previous work [16, 58]. For kernels, current type-based solutions resort to manually annotating allocation sites with the necessary type information [15].

In this paper, we take a systematic approach to discover type confusion vulnerabilities resulting from incorrect downcasting on structure embeddings, which we call *container confusion*. We design a new sanitizer that does away with runtime type tracking of objects and uses instead information on object allocation boundaries, which we obtain using an off-the-shelf solution. In more detail, we rely on redzones from memory sanitization literature [50] to augment allocation sites for out-of-bound access detection. Our sanitizer checks type compatibility for a downcasting operation by checking the relative position of the embedded parent structure, the outer child structure, and the redzones. This scheme transforms a type check in multiple straightforward structure bound checks, with low runtime overhead and no manual code changes.

We apply our sanitizer to the Linux kernel, one of the most complex and security-sensitive program instances. An initial study of its code base, which we conducted to gauge the potential bug surface, reveals more than 50,000 occurrences of `container_of` involving nearly 4,000 structure types. The type graph is also highly connected, with extreme cases such as `list_head` used as parent for over 1,800 child types.

We fuzzed a sanitized build of the kernel for one week and uncovered 11 cases of container confusion, including long-standing container confusion bugs present in its code base since 18 years. As the kernel is continuously fuzzed under multiple sanitizers and configurations, these findings lead us to argue that our approach can find bugs that current state-of-the-art testing practices fail to capture.

By analyzing the nature of such bugs, we identify five container confusion patterns of general interest. We use such patterns to develop a static code analyzer that can process the whole kernel in only a few seconds, allowing us to reach also code compartments that fuzzers may not cover. The static analyzer identifies 366 potential cases of confusion: by manual analysis, we identify 78 other bugs along with 179 anti-patterns where code correctness hinges only on implicit assumptions on program semantics.

We reported our findings to the Linux kernel maintainers, who acknowledged them, and proposed patches for all the bugs we found. At the time of writing, 102 patches have been merged in the kernel, and 6 CVE identifiers have been assigned for bugs whose security implications were immediately apparent. Our reports sparked valuable discussions which, among others, resulted in upgrading the C standard (to mitigate recurrent issues that we found) and in an attempt to change the list iterator integral to the kernel.

In sum, this paper proposes the following contributions:

- We systematize a class of type confusion bugs, showing how C programs are affected by incorrect downcasting on structure embeddings. We dub it *container confusion*.
- We design a sanitizer for them that does away with type tracking and show its applicability to the Linux kernel.
- We derive 5 general patterns of container confusion from bugs we found in the kernel and design a static analyzer around them to make our approach scale in coverage.
- We evaluate our approach on a recent Linux kernel version, identifying 11 bugs with dynamic analysis (e.g., fuzzing) and another 78 bugs through our static analyzer.

Our sanitizer and static analyzer together form a framework, termed UNCONTAINED, which is open source and available at: https://vusec.net/projects/uncontained.

2 Background

In this section, we will provide the relevant background to understand the remainder of the paper.

2.1 Type Confusion Bugs in C++... *and in C*

Casting an object to an incompatible type violating casting rules (i.e., bad-casting) causes *type confusion*. For instance, a static downcast in C++ checks only if the source and destination types are in the same type hierarchy, but not if the runtime destination type is the expected one. As a result, large C++ projects, such as the major browsers, parts of Windows, and the Oracle JVM [21], are rife with type confusion bugs.

Downcasting in C. The problem is not limited to object-oriented languages such as C++ but also extends to large programs written in C. Since C is not an object-oriented programming language, it does not support classes like C++. However, developers use *structure embedding* to benefit from an approximation of classes and inheritance. In particular, properties shared by multiple types are defined as a struct *embedded* in all the relevant types. In such a way, all the child types inherit the struct members declared in the parent type that is embedded. We show a simplified example of such use in Listing 1. Since the child type includes the parent type in this design, it is called a *container*.

Analogous to C++, we require primitives to go from the child type to its parent ("upcasting") and from the parent to its child type ("downcasting"). Upcasting is implemented by obtaining a pointer to the embedded parent structure from the child structure and is guaranteed safe. Downcasting is not defined in the C standard since it would require using a pointer to the parent structure to obtain a pointer outside of the memory defined by the type of the parent structure itself [43]. Still, many projects, including the Linux kernel, do exactly that. Given a pointer to the parent in a type hierarchy based on structure embedding, they implement their own version of downcasting, often in the form of a macro, that uses pointer arithmetic to calculate a pointer to the child type.

Such a macro is often named `container_of`. The reference implementation in the Linux kernel is shown in Listing 2. The `container_of` macro is not exclusive to the Linux kernel but present in many large C projects such as Qemu, Nodejs, Xorg, the Windows kernel, git, FreeBSD, and XNU.

List Iterators. As an example, consider the popular `list_head` structure that programmers embed in their data structures in the Linux kernel to create a double-linked circular list, with `next` and `prev` pointers pointing to the next and previous `list_head` element of the list. Iterating over a list, we know we have reached the end when we encounter the same pointer a second time. An empty list has its `next` and `prev` pointers pointing to itself. Issuing a `container_of` on a `list_head` allows access to the derived type, i.e., the element of the entry.

While there are different ways to use `list_head`, adding a linked list to a structure in the Linux kernel is a matter of embedding a `list_head` whose `next` field points to the first entry of the list, while that of the last entry points back to the `list_head` in the "owning" data structure. In this way, all list entries have the same type, except the owning structure that anchors the *head* of the circular list. Similarly, it is safe to issue a `container_of` from any list entry, except for the `list_head` in the owning structure, where it would lead to container/type confusion. The owning structure need not even be a struct, as it could also be a single `list_head` variable.

To iterate over a list, the kernel uses macros such as `list_for_each_entry`. It repeatedly follows the `next` pointer to find the next `list_head` and then uses `container_of` to set the iterator to the base of the entry that embeds it. For instance, we can iterate over all inodes of a superblock as follows:

```
struct superblock *sb; // owning data structure -> struct
↪  superblock embeds 'struct list_head s_inodes'
struct inode *inode;   // iterator -> struct inode embeds
↪ 'struct list_head i_sb_list'
  ...
list_for_each_entry(inode, &sb->s_inodes, i_sb_list) {
  spin_lock(&inode->i_lock);
  ... // do more with inode
}
```

This is safe if the possibly invalid list iterator, upon loop exiting, is not used afterwards. While the most common, `list_head` is not the only iterator in the Linux kernel but most work in a similar way. Well-known further examples include single-linked lists (`hlist_node`) and red-black trees (`rb_node`).

This paper will highlight several cases where iterator invariants are violated, resulting in buggy code.

2.2 Sanitizers

Sanitizers are runtime tools to detect undefined behavior in programs, typically through compiler-based instrumentation that checks undefined behavior. The best-known example is AddressSanitizer (ASan) [50], which detects memory errors such as buffer overflows and use-after-frees. ASan instruments every memory access with a check that consults a shadow memory to see if the memory access is valid. In particular, to detect buffer overflows, ASan pads memory allocations with *redzones* and poisons the memory in the shadow

```
// parent struct
struct usb_request {
    void *buf;
    unsigned length;
    dma_addr_t dma;
    ...
}
// child struct
struct gr_request {
    struct usb_request req;    // member field
    ...
    struct gr_dma_desc *first_desc;
    ...
};
// child struct
struct goku_request {
    struct usb_request req;    // member field
    ...
    unsigned mapped:1;
};
```

Listing 1: Structure embedding example, where `gr_request` and `goku_request` "inherit" from `usb_request`.

```
#define container_of(ptr, type, member) ({     \
  void *__mptr = (void *)(ptr);                \
  ((type *)(__mptr - offsetof(type, member))); \
})
```

Listing 2: `container_of` implementation in the Linux kernel.

memory (setting it to a nonzero value) so that any future access results in an ASan error. In this paper, we will repurpose ASan redzones to detect object boundaries.

3 Container Confusion in the Linux Kernel

In this section, we discuss security risks that can arise from container confusion, examine a real-world bug as a running example, and show to what extent the Linux kernel resorts to structure embedding.

3.1 Security Implications

Like C++'s `static_cast`, the `container_of` macro does not perform runtime checks to verify whether the structure is actually contained within the expected outer structure. When this is not the case, container confusion leads the program to access memory under wrong assumptions on its layout. Two base scenarios are possible: a) the structure is embedded in a different container, leading to member access over memory contents typed for another layout; or b) the structure is not embedded in a container, leading to a pointer that is out-of-bounds by the relative offset assumed within the container.

The security implications of bad casting have been well-researched for C++ (e.g., in the CaVeR paper [39]) and similarly apply here, being `container_of` equivalent to C++'s static downcasting. Such effects can range from subtle state corruptions to controlled out-of-bounds accesses that attackers can evolve for exploit construction. The security risk is mainly dependent on structure layouts, for example when memory containing function pointers can be overwritten. To

```
1  static int gr_dequeue(struct usb_ep *_ep,
2                        struct usb_request *_req) {
3    struct gr_request *gr_req; // renamed: was `req`
4    ...
5    struct gr_ep *ep = ...; // derived from `_ep`
6    list_for_each_entry(gr_req, &ep->queue, queue) {
7      if (&gr_req->req == _req)
8        break;
9    }
10   if (&gr_req->req != _req) {
11     ret = -EINVAL;
12     goto out;
13   }
14   ...
15 }
```

Listing 3: Using the list iterator `gr_req` past its validity causes container confusion.

probabilistically mitigate these and other issues, the Linux kernel can randomize the layout of some structures at compile time [27]. While this can make exploitation less reliable, in some cases it may also turn an unexploitable bug into a security vulnerability. In fact, as the offset for the embedded structure changes, also does the memory pointed by the type-confused pointer, directly affecting the bug exploitability (for example, when further memory corruption becomes possible under some randomized layouts). At the time of writing, only a few structure types (65 in the entire kernel) can undergo randomization: enabling it globally (as done in research operating systems [17]) can be difficult as code may assume a specific layout for some structures, while others have layouts that are tuned for better performance [12].

We will show concrete examples of security risks uncovered by the dynamic and static analyses of UNCONTAINED in Sections 6 and 7.3, where we outline, among others, a vulnerability that breaks Kernel Address Space Randomization (KASLR) and a controlled out-of-bound write. We will also discuss examples of bugs that may affect execution semantics.

3.2 Running Example

We discuss next our running example (Listing 3) involving the kernel USB stack to better illustrate container confusion.

The function `gr_dequeue()` iterates over a list of requests to find and remove the one matching the supplied `_req` argument. Under correct operation, `container_of(&ep->queue.next, struct gr_request, queue)` in the macro at line 6 takes the address of field `queue` in a `gr_request` list entry and subtracts a quantity χ=`offsetof(struct gr_request, queue)` to make it point to the entry itself.

However, if the list is empty or does not contain it, the execution leaves the list iterator variable `gr_req` with a container-confused pointer. As mentioned in Section 2.1, the list iterator would incorrectly reference the owning structure (i.e., the list head), which has `gr_ep` type. The confused `container_of` subtracts χ from the pointer to the field `queue` in this other structure: the result will point somewhere within structure `*ep`.

The exploitability of the bug depends on the position of field `req`, used at line 10, within `gr_request` structures. Listing 1 shows the partial structure layout. Had the position been "deeper", the resulting pointer could have reached and surpassed the outer `gr_ep` structure, referencing the adjacent heap storage. Were `_req` to match such an out-of-bound pointer, the code attempts to remove a list entry that is not present, possibly causing further memory corruption.

Rich discussions followed our disclosure of the bug to the Linux kernel mailing list. As a result, the maintainers opted to migrate to the C11 standard, which would allow them to define the iterator variable with a scope limited to specific loops, preventing its usage afterwards. In the next section, we will examine the potential surface for container confusion cases in the Linux kernel.

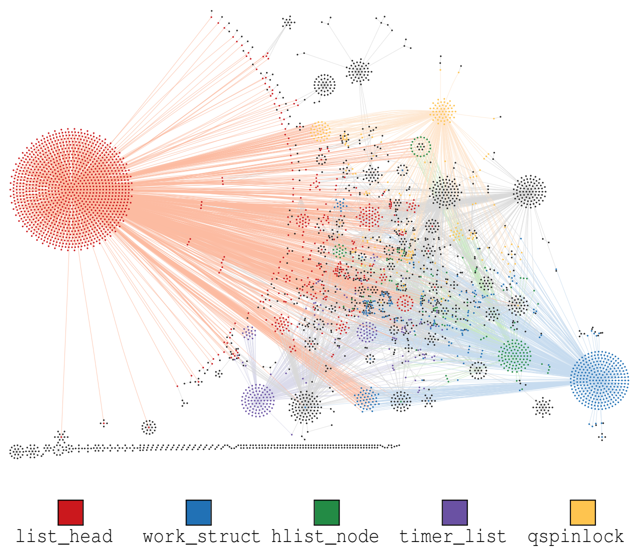

Figure 1: Type graph for `container_of` (and alike) instances.

3.3 Type Graph Complexity

To examine the use of structure embedding in the Linux kernel, we analyze the prevalence of `container_of` and its derivatives, as `container_of` takes part in several macros and inline functions. Depending on the selected kernel configuration, we note that the build system of the kernel can choose between different function implementations and even type definitions. Hence, we study the Linux kernel v.5.17 with the configuration in use to Google's syzbot [18] for continuous fuzzing.

We write an LLVM compiler pass to spot all the uses of `container_of` in the source code as lowered during compilation and track the parent and child types at each such use. This allows us to build a *type graph* that captures the possible containment relationships between different structure types. We count over 56,000 downcast instances (as `container_of` or any of its derivatives) under our kernel configuration.

As the paper will detail, the type graph is a foundational element of our approach to container confusion detection. Figure 1 shows the one being discussed here, highlighting the relationships between the embedded types. Each node represents a type involved in a downcast. We have a (directed) edge between two types if we find a downcast instance that derives a child of the destination node type from a parent of the source node type. We also compute edge weights based on the number of such instances.

While we count as many as 18323 types in all the code for the build, we find 4275 of them to be involved in downcast operations: 506 can occur as parent and 4033 as child object. To our surprise, this implies that almost one-fourth (23.3%) of all types are involved in structure embedding.

For example, the `usb_request` structure shown in Listing 1 can be embedded in 17 different child structures in use to different USB drivers. Generally speaking, a variety of destination types may favor cases of invalid runtime downcasts.

By looking at topological properties of the type graph, we find that 3486 of the 4033 possible destination types are not contained in any other type, meaning no other type "inherits" from them. 419 of the 506 possible source types have an outdegree greater than one, meaning that they can have multiple child types; 221 have more than 10 possible child types.

In the figure, we also highlighted the top-5 structure types by highest number of child types: `list_head` (1857), `work_struct` (611), `hlist_node` (244), `timer_list` (235), and `qspinlock` (223). Each colored cluster shows the possible destination types for such a source type during downcasting.

Looking at edge weights, the structure types most often used as parent when downcasting are `list_head` (22033), `inode` (7669), `device` (4130), `hlist_node` (3221), and `rb_node` (2272). Several of them are involved in iterators.

We also note that `list_head` emerges as the type with most child types that inherit from it and as the most used parent type across the whole kernel code base.

As the main takeaway of this study, we argue that the prevalence of `container_of` and derivatives, combined with the notable complexity of the type graph they induce, makes a compelling case for seeking container confusion bugs.

4 UNCONTAINED Overview

In this paper, we design and implement UNCONTAINED to detect container confusion bugs in the Linux kernel.

In Section 5, we present a novel container confusion sanitizer that uses object boundaries to detect invalid downcasts during dynamic analysis. After describing the design and implementation, we evaluate effectiveness and performance of the sanitizer by combining it with the well-known syzkaller [19] kernel fuzzer and other benchmarks. Finally, we use the sanitizer to analyze the occurrence of container confusion in the Linux kernel.

Achieving code coverage with dynamic analysis on the Linux kernel can be challenging due to the amount of complex code. In Section 6, we therefore analyze the bugs we detect through fuzzing and identify common bug patterns that result in invalid `container_of` usage. Based on these patterns, we develop a static analyzer to search for additional bugs without suffering from the lack of code coverage inherent to dynamic analysis in Section 7. In particular, we design and implement a configurable LLVM forward and backward dataflow analysis to identify potentially buggy code patterns. We then analyze any additional bugs found by the static analysis, including a worrying out-of-bounds write, and demonstrate an acceptable rate of false positives. Although static analysis has lower accuracy than dynamic analysis, it acts as an effective complement for code that dynamic analysis fails to reach.

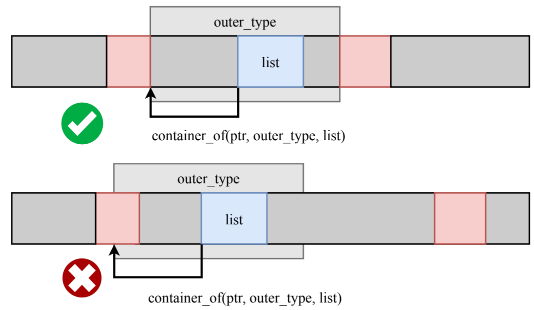

Figure 2: Redzone layout for a valid downcast (top) and for an invalid one (bottom). Here, *list* is the member field name.

5 Container Confusion Sanitizer

This section introduces the sanitizer component of UNCONTAINED meant to detect cases of container confusion at runtime. We explain its design and implementation in Section 5.1 and Section 5.2, respectively, and evaluate it in Section 5.3.

5.1 Design

Our sanitizer aims to expose `container_of` uses where an incorrect destination (i.e., child) type causes a container confusion. As we anticipated in Section 1, detecting such errors with existing approaches to type confusion detection would require maintaining a form of RTTI for each allocated object.

Our design aims instead for a general solution that does not incur code modifications and/or pointer tracking costs while achieving broad compatibility. The key idea is to turn a downcasting validity check into multiple bound checks relative to the current embedded object (the parent) and the requested container object (the child) of a `container_of` operation. Parent and child here are synonyms for *inner* and *outer* structure.

We analyze structure definitions and use the relative distances of an embedded structure from the start and the end of its container structure as the discriminating factor for violations. When the container object is of the requested type, its

```c
static int gr_dequeue(struct usb_ep *_ep,
                      struct usb_request *_req) {
  struct gr_request *gr_req; // renamed: was `req`
  ...
  struct gr_ep *ep = ...; // derived from `_ep`
  list_for_each_entry(gr_req, &ep->queue, queue) {
    if (&gr_req->req == _req)
      break;
  }
  if (!check_redzone(gr_req, sizeof(struct gr_request))) {
    uncontained_report(gr_req);
  }
  if (&gr_req->req != _req) {
    ret = -EINVAL;
    goto out;
  }
  ...
}
```

Listing 4: Running example with our bound checks added.

allocation boundaries will align perfectly with those that one can infer starting from the parent pointer. A violation occurs instead when the object enclosing the parent turns out to be larger or smaller than expected on either side.

To insert sanitization checks, inferring the expected boundaries of a child object is straightforward, as both its size and the displacement of the parent field from its start are known at compile time. However, even at runtime, the actual boundaries of an object are normally not available in C programs.

Object Boundaries. For reliable boundary identification, we rely on standard runtime means in use to sanitizers that target spatial memory safety violations. Namely, we pad object allocations with redzones (Section 2.2) and use them to recover object boundaries. The addresses immediately preceding and following an object will appear as invalid in the shadow memory, while those at the boundaries will be valid.

For a `container_of` operation, we can thus check for the validity of memory at the expected start and end addresses of the requested container, and the invalidity of the memory right before and after them, respectively. This will readily expose mismatches between expected and actual boundaries.

Figure 2 shows an example of valid and bad downcasting, highlighting the differences in their object redzone layouts.

We chose a redzone-based approach over other bounds-tracking designs due to its efficiency, practicality, and compatibility with complex code bases: mainly, inspections have O(1) cost and we can build on an existing, well-tested infrastructure from memory sanitizers for kernels. Alternative design points such as low-fat pointers [36] remain a possibility.

Container Nesting. The bounds-checking policy we just presented may mishandle containers that are embedded in another container. For those cases, we cannot expect the presence of redzones for the inner container, being it a structure field. However, we can still do the validation through the outer container. In the Linux kernel, only 547 of its 4033 container types may incur such a scenario, whereas for 3486

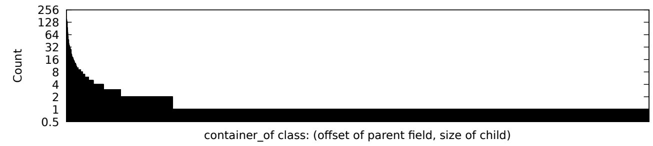

Figure 3: Distribution of `container_of` invocations according to offset of parent field and container size. Logarithmic scale.

no nesting is possible. Therefore, when the desired child type of a `container_of` instance is one of those 547, we apply the following scheme if the normal bound checks fail.

We note that a `container_of` operation carries the expected type for the innermost container only. Moving to an outer container, we can check if its boundaries (i.e., the redzones around it) align with the layout expected for any of the container types that have a field of the expected inner container type. This information is available in the type graph (Section 3.3) at compile time and we compute it recursively for multi-nesting cases. If the redzones of the outermost container do not match any feasible layout, we report a container confusion error.

This strategy effectively allows us to avoid false positives from container nesting. The attentive reader may notice that, by accepting more redzone layouts as valid, we open the door to more false negatives: however, as we will show later in this section, the probability of such layout collisions is very low.

Time-of-use Checking. In the Linux kernel code, we found several cases where a `container_of` instance sees at runtime also objects of an incompatible type but the following code is never affected by the confusion. For example, with list iterators, the obtained child pointer was used only to access the parent again through the child field corresponding to it. These cases in the programming practice are not strictly bugs. Therefore, in our design, we opted to validate a `container_of` instance at the time of use for its output pointer rather than immediately when downcasting. Listing 4 shows our running example augmented with bound checks around redzones.

To identify uses of the output pointer, we run a standard intra-procedural def-use [23] analysis. As the program may modify it before dereferencing it (e.g., to access a child field), we analyze pointer arithmetic operations and, when the modification can be determined statically, we forward the check to the next use of the pointer. When the program dereferences it or we can no longer follow it statically, we emit bound checks and have them account for the modified offset, if any.

Discussion. The sanitization scheme we propose can detect container confusion by relying solely on structure layout knowledge (known at compile time) and object boundaries (obtainable with off-the-shelf lightweight techniques). When both sources are accurate, no false positives are possible.

Compared to an ideal design that tracks pointer types, the price we may pay for our efficiency and compatibility relates to false negatives when an invalid downcast involves an object whose layout coincides with the one of a valid child type[1].

To look into this dimension, we identify a domain and a codomain for it. As domain, we study how many unique `container_of` instances are present in the Linux kernel as we consider the pair (parent field, child type) for a downcast operation. We include the field as one child may embed multiple parents. As codomain, we identify pairs of the form (offset of parent field, size of child) for such operations, since these are the two quantities that we use—independently from one another—for bound checking. We count $6,526$ unique instances mapping to unique $3,262$ pairs. A collision occurs when two distinct instances map to the same pair.

The distribution in Figure 3 shows that 40.8% of the unique `container_of` instances map to one pair exclusively, 16.9% to 2-4 pairs, 21.1% to 4-32 pairs, and only 5 of them to 100 or more pairs. Hence, we expect collisions to be infrequent. We then analyze them under the realistic hypothesis that incorrect downcasts happen only over objects of related types. When counting all the siblings and descendants in the type hierarchy for the expected downcast type of a unique `container_of` instance, we measure the probability of a collision to be 0.0283, which decreases to 0.0088 when considering siblings only.

Note also that one may avoid false negatives almost entirely by adding padding bytes to structures mapped to the same codomain point(s). We leave this investigation to future work.

5.2 Implementation

The sanitizer of UNCONTAINED consists of two components. The first one is a coccinelle [44] script to intercept occurrences of `container_of` at the source level, which the C preprocessor would otherwise expand before we may instrument them.

The second one is a pass for the intermediate representation (IR) of the LLVM compiler (v.12.0.1) implemented in 1640 lines of C++ code. The pass is responsible for building the type graph of the code base, expanding the intercepted `container_of` instances, and adding sanitization machinery.

We also develop a framework[2] of potentially independent interest to apply custom LLVM passes during kernel compilation and run VMs for testing (e.g., with syzkaller) and debugging, automatically spawning one with a breakpoint attached to the found crash site for manual inspection in gdb.

To have full visibility on type information, we run our pass as a link-time optimization. We then leverage the existing redzone insertion and shadow memory mechanisms of Kernel Address Sanitizer (KASAN) [33] to support object boundary identification for stack, global, and heap-allocated variables. While our sanitizer can coexist with KASAN's machinery to sanitize memory accesses for safety violations, we disable its generation as these checks are unnecessary for our purposes.

As mentioned in the previous section, correct object boundary identification is essential for precision. This aspect is not influenced by the redzone size (for which we use KASAN's defaults), as the shadow memory has always 1-byte granularity. However, even state-of-the-art techniques for redzones fail to handle the edge cases we discuss next. As they may lead to false positives, we disable confusion checks for them.

We find two object allocation schemes that require special handling. One involves a known limitation of redzones with arrays: in these cases, redzones cannot be inserted around their individual elements, unless one modifies the type definition. With a coccinelle script, we identify in the code base all the types that take part in array allocations and disable the validation of `container_of` instances using them as a child type. For future work, we are considering the addition of machinery to test all possible array cells when their number is known statically, whereas for dynamic sizes the recent proposal of bounded flexible C arrays [8] may be of help.

The second scheme involves the allocation of multiple, differently typed structures (e.g., `kalloc(sizeof(A) + sizeof(B), ...)`) followed by pointer extraction for each structure. This coding choice brings performance benefits, as it optimizes the use of the allocator, but complicates memory sanitization schemes. To avoid false positives for objects involved in such allocations, we devise a coccinelle script to disable the involved types from validation. However, for a few recurring cases and if code semantics allowed doing so safely, we manually split allocations and enable container confusion detection for types like `io_buffer` used in `io_uring` code or `net_device` private data in networking code.

Overall, for the two schemes, we disable validation for $13,926$ out of $56,468$ downcasts. We also highlight that the shadow memory and redzones of KASAN operate only after the early boot phase of the kernel. Heap objects allocated by the boot memory allocator `memblock` have no redzones: we identify and skip them using address range checks at runtime.

While we test and evaluate our sanitizer around the Linux kernel, the adaptations needed for other subjects would be limited. Redzone management for userland software is available in LLVM with AddressSanitizer [50], while kernels like FreeBSD and XNU have their own KASAN implementation.

5.3 Evaluation

We run our sanitizer on the Linux kernel v.5.17 (commit `c269497d248e`). For the fuzzing experiments, we use syzkaller (commit `9e8eaa75a18a`) and build two images compiled, respectively, with the default kernel configuration and the one in use to Google's syzbot [18], as it enables additional features. The choice is an attempt to slightly balance the exploration of code between pervasiveness and breadth.

[1] We deem a container confused if not immediately preceded (resp., followed) by a redzone byte and if its first (resp., last) byte is valid memory. With a false negative, the former check lands on invalid memory and the latter on valid memory. Note also that this property is not affected by the redzone size.

[2] Available at https://github.com/Jakob-Koschel/kernel-tools.

To stress specific/additional components, we also run typical userland workloads such as installing programs with the `aptitude` package manager, executing `binutils` utilities, code for SGX enclaves, and the Linux Test Project [37].

As experimental setup, we ran syzkaller for one week on two Ubuntu 22.04.1 (Linux kernel v.5.15) host machines with 16 cores @2.3GHz (AMD EPYC 7643), using a total of 16 QEMU-KVM virtual machines with 4GB RAM and even distribution of the default and the syzbot-configured builds.

5.3.1 Discovered Cases of Container Confusion

Our fuzzing campaign revealed 37 cases of container confusion. After manual analysis of the crash sites, we identified 11 unique bugs and 10 *anti-patterns* (see below). The remaining 16 are false positives deriving from missing redzones in mixed-type allocations that our coccinelle scripts miss (Section 5.2). Adding them to our filtering logic is a one-time effort that would prevent such false positives from occurring in future campaigns.

We consider *anti-patterns* type confusion cases where the use of a confused pointer is a "controlled" case of undefined behavior as the code does not incur a corruption only thanks to implicit assumptions on program semantics (which may silently change over time) and/or compiler behavior. Such anti-patterns might silently turn into bugs in future releases.

The 11 bugs affect the following kernel subsystems: `drivers/net`, `net/{ipv4&6, sctp}`, `fs/f2fs`, and `sgx`. We disclosed and proposed patches to the maintainers for all the bugs: at the time of writing, all patches have been or are being merged. We present five of these bugs in Section 6. The 11 bugs had not emerged, e.g., in the continuous fuzzing efforts from Google's syzbot, which uses state-of-the-art sanitizers like KASAN and tests several configurations.

The 10 anti-patterns relate to places where a container confusion occurred but developers manage it explicitly later. As examples, we briefly describe two of the anti-patterns that our sanitizer found. The first involves the function `crypto_alg_lookup()` of the Kernel Crypto API. The function can return a pointer to a synchronous-hash structure (`shash_alg`) confused as if it were an asynchronous (`ahash_alg`) one. However, all the users of the function eventually check the requested instance type through additional fields to differentiate them and correctly cast the confused pointer before use. The second involves the `inet_lookup_established()` networking function, which can return a pointer to a `struct inet_timewait_sock` confused as a `struct sock`. Similar to above, all the users of the function check the socket state to differentiate them.

5.3.2 Runtime Overhead

We conduct two sets of experiments to measure the overhead introduced by the sanitizer component of UNCONTAINED: the bare sanitization costs with LMbench [41] and their impact on the end-to-end throughput when fuzzing with syzkaller.

We run the LMbench programs on a single QEMU-KVM instance with 8 GB of RAM executing on an i7-10700K CPU host machine with minimal background activity and identical software to the previous experiments. We repeat each experiment 10 times, taking the median value for every program. Our sanitizer introduces a geomean overhead of 74%. As a reference, KASAN introduces a 126% overhead (with 33% coming from redzone management, which we use too). We list figures for the individual programs in Appendix B.

For fuzzing throughput, we measure how many test cases one syzkaller VM executes within the first hour of fuzzing. We take the median value of 10 experiment repetitions, starting from an empty fuzzing corpus. The syzkaller baseline with no sanitizers enabled executed 80348 test cases, whereas with UNCONTAINED 69734 with a net reduction of the fuzzing throughput of around 13%. As a reference, KASAN introduced a 55% net reduction of the throughput. We find our approach to induce an overhead[3] acceptable for fuzzing.

6 Retrospective Analysis and Bug Patterns

The cases of container confusion that our sanitizer detected when fuzzing revealed several lingering bugs and anti-patterns in the Linux kernel. Their analysis brought out two key reflections we present next, as they motivate and form the basis of the research from the remainder of the paper.

Unexplored Code. In spite of the widespread use of containers, the issues found were located in a fairly limited, yet relevant, subset of the Linux kernel code base. Prolonging the fuzzing campaign by a few days did not uncover new bugs.

We find this to stem directly from the inherent coverage problem of dynamic tools. Much code may be locked under specific kernel states [22, 61], require emulation for crossing the hardware/software barrier with device drivers [47], or need complex input generation logic (e.g., with protocols). Special-purpose fuzzers [11, 45, 47, 49, 52–54, 57], which one may run naturally on our instrumented kernels, currently exist only for a fraction of such components.

This led to us eventually to investigate container confusion detection through static approaches that could cover the whole code base, even if with a diminished precision/recall.

Dynamics of Bugs. We noted a few distinctive traits in the nature of the bugs spotted with the experiments of Section 5.3. These may make some bugs harder to reason about, especially for static analysis. However, as we show in Section 7, domain knowledge (e.g., on list operations) can come to the rescue.

[3] One opportunity to reduce it would be to follow [52] by disabling stack walking upon memory (de)allocation events, as it helps only for crash debugging/deduplication but is expensively frequent. Each crash may be analyzed offline by re-running the test case in an unmodified KASAN.

For example, one trait relates to whether, for a `container_of` instance that sees objects incoming from a given program path, confusion occurs on all or only a few of them (e.g., only on a list's owning element). Another relates to whether, on the path(s) from the container allocation to its confused use, pointer upcasts and downcasts involve indirection (e.g., the address is stored in a field of another object).

In the following, we present five bug patterns that encompass all the issues of Section 5.3 and represent general forms of container confusion. These patterns are distinct, albeit not exhaustive in terms of possible types of confusion (other than those we encountered). Most importantly, the descriptions we give are actionable for program analysis (Section 7).

Pattern ❶: Statically Incompatible Containers. This pattern describes the most generic and shallow container confusion that we identified. It involves using a type (or member field) that is always incorrect when downcasting object pointers incoming from a certain program path.

Listing 5 reports an exemplary bug found when fuzzing in the `sock_init_data()` function while manipulating a `socket` struct. The function assumes that its `struct socket* sock` parameter is embedded in a `socket_alloc` container. This assumption is correct for most sockets in the kernel, except for TUN and TAP ones. Hence, when a program path from function `tun_chr_open()` reaches the buggy function, its argument is embedded in a `tun_file` container instead.

When the function assigns the socket with the owner's UID, the confused bytes are always set to zero in the kernel configuration that we tested. Any TUN or TAP socket thus appears as owned by the root user, nullifying user-based firewall/routing rules possibly in place. The severity of the bug may be even amplified by the effects of structure randomization (Section 3.1). At the time of disclosure, the bug had been present in the Linux kernel for more than 6 years.

Pattern ❷: Empty-list Confusion. As we anticipated in Section 2.1, a confusion can originate when issuing a `container_of` operation on the owning structure of a circular list. When such a list is empty, the owning structure sees the `next` and `prev` fields of its embedded `list_head` point to itself. Accessing list members in a `list_entry`[4], `list_first_entry`, or `list_last_entry` operation causes container confusion.

Listing 6 reports an exemplary bug found in the kernel networking stack when fuzzing. Since the `inet_diag_msg_sctpasoc_fill()` function assumes that the `asoc->base.bind_addr.address_list` list is populated without checking for it, `laddr` points to a container-confused object when the `list_entry()` operates on an empty list. The code at line 11 copies some of its fields into memory provided to userspace. As these confused fields contain kernel heap pointers, this results in a KASLR leak that deterministically breaks the address randomization of the kernel, which often represents one of the first steps in kernel exploitation [20,26,28,34].

```
static int tun_chr_open(struct inode *inode, struct file *file) {
  struct tun_file *tfile;
  ...
  sock_init_data(&tfile->socket, &tfile->sk);
  ...
}

struct inode *SOCK_INODE(struct socket *socket) {
  return &container_of(socket,
    struct socket_alloc, socket)->vfs_inode;
}

void sock_init_data(struct socket *sock, struct sock *sk) {
  if (sock) {
    ...
    sk->sk_uid = SOCK_INODE(sock)->i_uid;
  } else {
    ...
  }
  ...
}
```

Listing 5: The first argument to `sock_init_data()` is contained within `tfile` when called from `tun_chr_open()`. `SOCK_INODE()` incorrectly assumes `sock` to be contained within a `socket_alloc` struct.

```
1  static void inet_diag_msg_sctpasoc_fill(
2          struct inet_diag_msg *r,
3          struct sock *sk,
4          struct sctp_association *asoc) {
5    union sctp_addr laddr;
6    ...
7    laddr = list_entry(asoc->base.bind_addr.address_list.next,
8      struct sctp_sockaddr_entry, list)->a;
9    ...
10   if (sk->sk_family == AF_INET6) {
11     *(struct in6_addr *)r->id.idiag_src = laddr.v6.sin6_addr;
12     ...
13   }
14   ...
15 }
```

Listing 6: `list_entry()` assumes the presence of at least one entry within `asoc->base.bind_addr.address_list`, causing a container confusion in `inet_diag_msg_sctpasoc_fill` due to the missing check for whether the list is empty.

At the time of disclosure, the bug had been present in the Linux kernel for almost 7 years.

Pattern ❸: Mismatch on Data Structure Operators. Insertion, deletion, selection, and other operations on objects taking part in container-based data structures (e.g., lists, trees) should see the use of consistent types and member fields.

Listing 7 shows an exemplary bug found when fuzzing involving the `sock` structure. A `struct sock` can be inserted into multiple lists and therefore embeds multiple list structures. Among others, it contains two single-linked lists using the fields `sk_bind_node` and `sk_node`. With a list, its elements must always be accessed via the field used to insert them into it. The socket code manages the `&tb->owners` list, which holds sockets using their `sk_bind_node` member. But `__inet_hash_connect()` accesses the same objects using the

[4]We recall that `list_entry` is simply an alias for `container_of`.

```
1  void inet_bind_hash(struct sock *sk,
2          struct inet_bind_bucket *tb,
3          const unsigned short snum) {
4    ...
5    hlist_add_head(&sk->sk_bind_node, &tb->owners);
6    ...
7  }
8
9  int __inet_hash_connect(..., struct sock *sk, ...) {
10   ...
11   struct inet_bind_bucket *tb;
12   ...
13   if (port) {
14     ...
15     tb = inet_csk(sk)->icsk_bind_hash;
16     ...
17     if (hlist_entry((&tb->owners)->first,
18         struct sock, sk_node) == sk &&
19         !sk->sk_bind_node.next) {
20       inet_ehash_nolisten(sk, NULL, NULL);
21       spin_unlock_bh(&head->lock);
22       return 0;
23     }
24     ...
25   }
26   ...
27 }
```

Listing 7: `inet_bind_hash()` inserts list elements using the `sk_bind_node` member, whereas `__inet_hash_connect()` accesses them incorrectly using the `sk_node` member.

`sk_node` member. In this case, the two members are located at different offsets, thus the downcast on the access adjusts the pointer incorrectly, causing container confusion.

As a result, the condition at line 17, which controls a fast path for the function, never evaluates to true. At the time of disclosure, the bug had been present in the Linux kernel for 18+ years (i.e., the extent of its git history).

Pattern ❹: Past-the-end Iterator. Developers often rely on a break-like logic when searching for an element in a data structure using iterators. Program semantics may sometimes deceive them into believing that a search will always succeed, so they may use an iterator without checking for its validity, which would not hold if the loop completes.

This container confusion characterized our running example (cf. Section 3.2). Listing 8 shows another exemplary bug that we found in SGX code when running an enclave in our instrumented kernel build using `qemu-sgx`. As the function processes an empty `&encl_mm->encl->mm_list` list, the `tmp` iterator is never assigned a valid entry, holding a confused pointer after the loop. At the time of disclosure, the bug had been present in the Linux kernel for more than 2 years.

Pattern ❺: Containers with Contracts. An object embedded in a data structure may come with additional metadata (e.g., custom RTTIs [39]) that program semantics uses as an implicit *contract* to control what operations can be done on it.

This is the case with the *sysfs* subsystem of the kernel, which lets userspace programs inspect and control several kernel features. Listing 9 shows a container confusion that we found in an inspection function when fuzzing. Here, the

```
void sgx_mmu_notifier_release(struct mmu_notifier *mn,
                              struct mm_struct *mm) {
  struct sgx_encl_mm *encl_mm = ...;
  struct sgx_encl_mm *tmp = NULL;
  ...
  list_for_each_entry(tmp, &encl_mm->encl->mm_list, list) {
    if (tmp == encl_mm) {
      list_del_rcu(&encl_mm->list);
      break;
    }
  }
  ...
  if (tmp == encl_mm) {
    synchronize_srcu(&encl_mm->encl->srcu);
    mmu_notifier_put(mn);
  }
}
```

Listing 8: Incorrect use of the list iterator variable `tmp` after the loop in `sgx_mmu_notifier_release()`.

```
1  ...
2    ret = kobject_init_and_add(&f2fs_feat,
3      f2fs_feat_ktype,
4      NULL, "features");
5  ...
6  ssize_t f2fs_attr_show(struct kobject *kobj,
7                          struct attribute *attr, char *buf) {
8    struct f2fs_sb_info *sbi = container_of(kobj,
9              struct f2fs_sb_info,
10             s_kobj);
11   struct f2fs_attr *a = ...;
12   return a->show ? a->show(a, sbi, buf) : 0;
13 }
```

Listing 9: Invalid `container_of` on `kobj` (originating from `&f2fs_feat`) in `f2fs_attr_show()`.

`kobject` that `kobject_init_and_add()` registers is not embedded in another structure, but the buggy `f2fs_attr_show()` function treats it as if embedded in a `f2fs_sb_info` structure.

This plays out as a "controlled" confusion, as the contract (i.e., the companion object of type `ktype` at line 3) carries a pointer, retrieved at line 11, to a function that does not access the confused `sbi` supplied at line 12. We classify this as an *anti-pattern*, as an imperfect knowledge of program semantics or changes to it would open up the possibility for bugs.

Bug Counts. With our sanitizer (Section 5.3.1), we discovered 6 mismatches on data structure operators, 2 cases of empty-list confusion, and 1 case for each of the other patterns.

7 Static Analyzer

This section introduces the static analyzer component of UNCONTAINED, which aims to identify the container confusion patterns presented in the previous section. We illustrate the design of our static analyses in Section 7.1, their implementation in Section 7.2, and the experimental results in Section 7.3.

7.1 Design

As anticipated in Section 6, our static analyzer aims for the code regions that are not within easy reach of current dynamic testing solutions. We note, though, that the reflections and bug patterns we presented involve phenomena, like indirection via memory, that may be expensive to reason about statically. Also, most of the bugs found involved inter-procedural flows.

For our analysis to scale to a code base as huge as the Linux kernel while maintaining satisfying accuracy, we make the following design choices. We cast bug pattern search to a static *information flow analysis* problem, relying on def-use information to track value propagation. The five bug patterns become rules for an on-demand backward or forward analysis where `container_of` instances act as sources or sinks depending on the pattern. We extend def-use chains through procedure boundaries (as a simplified form of [23]) and model memory as a single, coarse-grained symbolic location for scalability. We use semantic knowledge of common data structure manipulations (e.g., list iterators) to model several flows that involve indirection, enabling static reasoning.

We provide descriptions below for how we encode the five bug patterns as rules for the information flow analysis. Appendix C contains more rigorous definitions of what we use as (and do at) sources, sinks, and path-discarding filters.

Pattern ❶. To spot statically incompatible containers, we run a backward analysis from the pointer supplied to a `container_of` instance to every operation, if any, that obtains a pointer to an embedded structure starting from a pointer typed as a container. If the type (or member field) is incompatible with what `container_of` is asked for, we report a confusion.

Static reasoning is limited to instances for which we can infer the container type, i.e., cases where the code computes the parent structure pointer flowing into `container_of` by referencing the member field of the child structure—e.g., with a `&(child.member)` pattern. Our static reasoning gives up instead if the code reads the parent pointer value directly from memory: in these cases, even complex pointer analyses may be inconclusive due to aliasing, indirection, and other factors.

Pattern ❷. To spot potential accesses on empty lists, checking only for the use of dedicated helpers (e.g., `list_empty`, `list_is_head`, `list_entry_is_head`) would be prone to false positives. In fact, a code may keep track of the list size in a separate variable and check it before any downcasting; we find this to happen frequently in the Linux kernel.

We thus conduct a forward analysis from any occurrence of `list_{entry, next, prev, first, last}` to any use of the output pointer. If we encounter no conditional check guarding a use in the control flow, we report a potential confusion.

When reviewing buggy code, we also noted that some code erroneously compares the assigned pointer to `NULL` (whereas, when the list is empty, the result would reference the owning structure). Therefore, we added an analysis that detects such checks and deems them as *incorrect* (unless the code did not explicitly initialize the pointer as such before list iteration).

Pattern ❸. Object flows between operations involving container-based data structures (e.g., insertion and retrieval in a list) are in general hard to reason about statically, as they involve memory contents manipulation. However, we can rely on domain knowledge on the identity of the operations to detect cases of container confusion from inconsistent member selection.

We do a forward analysis from any operation on a data structure type to any subsequent operation on the same structure (e.g., from `list_add` to `list_entry`). If the pointers supplied to both can be determined to be the same but the container type or field is different, we report a potential confusion.

Pattern ❹. To detect when an iterator may have outlived its validity and cause container confusion if dereferenced, we analyze the instances of iterator-related macros that take part in loops. For each of them, we conduct an intra-procedural forward analysis to see if the code uses it outside the loop. We deem such a use as potentially confused if it is not guarded by a conditional check (e.g., using a boolean variable set by the loop), as developers typically insert one to assess whether the loop stopped advancing the iterator (i.e., before invalidity).

Pattern ❺. Confusion cases on containers with contracts are hard to spot in terms of code manipulations alone. We find it reasonable to assume that, for a given code base, the identity of such container types is known. For the Linux kernel, we devise an analysis for `kobject` containers that one may in principle adapt to other types from other code bases. The analysis comes with a forward and a backward component.

For each occurrence of the `kobject_init_and_add()` function, which is designed to register an object with its contract, we run a backward analysis to identify the containment relationships of the registered object and collect its `ktype` contract.

For each contract, we gather what functions of *sysfs* may be called on the object by inspecting its related fields. Then, we run a forward analysis from the `kobject` argument in each such function, looking for `container_of` invocations incompatible with any valid containment identified by the backward component.

7.2 Implementation

We implement the general forward and backward information flow analyses and the rules for patterns ❶, ❷, ❸, and ❺ as a pass for LLVM IR in 1286 lines of C++ code. Similarly to the dynamic analyzer (Section 5.2), we intercept every `container_of` occurrence at the source level and expose its source and destination type and object at the IR level. We run the pass at link time so we can effectively extend def-use chains across procedure boundaries. However, in this scenario LLVM would normally merge type definitions having an identical memory layout: to keep our analyses accurate,

Description	FP	AP	Bug
❶ Statically Incompatible Containers	72	27	3
❷ Empty-list Confusion	19	4	20
❸ Mismatch on Data Structure Operators	16	8	1
❹ Past-the-end Iterator	0	137	56
❺ Containers with Contracts	0	3	0

Table 1: Reports from the static analyzer categorized as False Positives (FP), Anti-Patterns (AP), and Bugs for each pattern.

we disabled this behavior by changing ~25 lines of code in the compiler.

The forward analysis starts from an IR value representing a source and follows its uses. When a use eventually reaches a function call argument, the analysis continues by seeing the uses of the arguments in the callee, recursively. The analysis also accounts for uses that concur to the return value of a callee, returning to the caller for continuing the analysis.

The backward analysis proceeds from a source IR value to its reaching definition(s). When it meets a function argument, it continues by exploring the code of each possible caller.

Both analyses stop exploring a path upon reaching a sink, a memory dereferencing operation (as we modeled memory as a single location), or an instruction already visited when analyzing a particular source. The rules for the patterns to check specify sources, sinks, direction of the exploration, and filters (if applicable) to stop a path exploration early.

Since our analysis visits each instruction at most once for each source location, and source locations are generally limited in number, we can approximately estimate the cost of our analysis as linear in the number of LLVM IR instructions.

As an implementation refinement, for pattern ❷ we suppress false positives involving container confusion in functions passed as callbacks for `list_sort()` or `seq_operations` structures. The reason is that the latter come with additional logic for emptiness checks before invoking the callbacks.

To ease the analysis of the reported confusion cases, we implement a Visual Studio Code plugin that recovers and presents to the developer the relevant code locations involved.

For pattern ❹, when reporting the bug presented in Section 3.2, the kernel maintainers pointed us to a coccinelle script proposed in 2012 by Julia Lawall on their mailing list to flag uses of iterators after loops. We assume that it had limited impact because of the high false positive rates. However, since our analysis for ❹ is simple and local, coccinelle is a great fit for it. We therefore extended the script in ways (mainly, with detection of checking logic already in place) that significantly reduced its false positive rate.

7.3 Evaluation

We run our static analyzer on the same kernel code base studied in Section 5.3. Table 1 summarizes the findings from a manual analysis of the reported cases of potential container confusion: we identified 80 bugs, 179 anti-patterns, and 107 false positives. We disclosed and proposed patches (144 in total with 97 already merged at the time of writing) for all the bugs as well as for the anti-patterns that can be removed without intrusive program semantics changes.

For the analysis time, we recall that pattern ❷ employs two rules whereas the others just one (❺ included, as its two analyses run in combination). We measure it took an average of 33.6 seconds for a rule to process all the container downcasts in the code that meet the definition of source for it.

We classify a report as a *bug* when the container confusion is unintended, which can lead to errors and possibly security-sensitive behavior. We consider as *anti-pattern* (AP) those cases where confusion can happen but program semantics prevents any use of the pointer. We consider as *false positive* (FP) those cases where pointers cannot have a confused value but the over-approximation of static analysis fails to see it.

Pattern ❶. Reports about *Statically Incompatible Containers* cases include 3 bugs, 27 anti-patterns, and 72 false positives. This pattern is prone to false positives (67.3% of the total among all five patterns) due to imprecision of the static analysis: we found most of them to occur when some backward control flows are unfeasible as they are guarded by checks on fields carrying explicit type tags[5]. A similar semantics is also behind most of the anti-patterns we found. As for the bugs, static checking identifies the TUN bug from fuzzing that we discussed when presenting the pattern in Section 6, but also a similar variant for TAP socket interfaces.

Pattern ❷. Reports about *Empty-list Confusion* cases are the second most numerous: we found 20 bugs (5 from missing checks and 15 from checks against `NULL`) and 2 anti-patterns.

For example, we found a container confusion in code that incorrectly checks HID device drivers reports, affecting all the 9 kernel drivers that rely on it. The bug had been present in the kernel for almost 9 years. In other HID driver code, we found 2 use-after-free and 1 NULL pointer dereference bugs. We also found a bug in the RT scheduler for an incorrect check on the task queue that had been present for 15 years.

The 19 false positives involve lists that cannot be empty due to program semantics, missing effects of indirect calls (like the sort comparators that we model already), and implementation limitations for non-nearby conditional checks.

Pattern ❸. We found a notable bug by looking for pattern *Mismatch on Data Structure Operators* cases. The bug affects the function `rds_rm_zerocopy_callback()`, which writes a

[5] It could be a one-time effort to add such domain knowledge to the checker and stop the analysis of the current path upon recognizing such explicit checks over fields. However, we found 72 false positives here to still be a reasonable number for the manual analysis we conducted.

cookie provided by userspace to memory. The function issues a `list_entry()` directly on the `list_head` instead of using `list_first_entry()`. The code passes the container-confused pointer to a function that finalizes the write.

The function uses confused values to write data to an offset where both are under userspace control, offering a controlled out-of-bounds (OOB) write primitive. Due to the container confusion, also an overlapping `lock` structure gets corrupted in the process, de-synchronizing it and potentially causing a use after free. The bug had been present in the kernel for 5 years. As the OOB write does not overlap with redzones, ongoing continuous fuzzing efforts could not detect it.

Anti-patterns mainly originate from iterating a list with an incorrect type, sharing a few initial member fields with the intended type. False positives come from implementation limitations with complex cases of GEP instructions in LLVM IR and unfeasible control flows from switch-case constructs.

Pattern ❹. Reports about *Past-the-end Iterator* are the most numerous in our results: this is quite expected, being list iteration popular in the kernel. We identify 56 bugs and 137 anti-patterns where the code may use a list iterator without checking whether it surpassed the end of the data structure.

The most immediate effect of our reporting and patching activity was upgrading the C standard for the Linux kernel to C11 [9]: this makes it possible to declare iterators valid only within loops, forcing developers to use (valid) retrieved values in a safer way. Shortly after, Linus Torvalds and other maintainers followed up with a proposal under adoption for a safer design of list iterators [10] that prevents anti-patterns of this kind completely.

Pattern ❺ We conclude by briefly mentioning that our reports from searching for *Containers with Contracts* cases uncovered two anti-patterns involving `kobject` container confusion in addition to the one discovered by dynamic analysis.

8 Discussion

We find that the dynamic and static components of UNCONTAINED operate synergetically to expose typically different instances of bugs over large code bases such as the Linux kernel.

Thanks to precise runtime information, the sanitizer component offers high accuracy by incurring only a few false positives in our tests.

The wealth of information also allows it to detect bugs that are out of reach of the static analyzer due to the latter's inherent under-approximation (e.g., for cases of memory indirections that we cannot recover via domain knowledge). This can be seen in the limited overlap in the bugs found: only 2 of the 11 bugs found dynamically occur in the reports of the static analyzer.

On the other hand, the static analyzer succeeds in its intended goals, revealing a large number of bugs (80) originating often in kernel areas that the dynamic experiments did not stress sufficiently or at all—and are also fundamentally difficult to cover due to configuration and hardware entropy. These include virtual drivers, ptrace facilities, the RT scheduler, and the kernel components of NFS and KVM, among others. Being a static analysis, the main shortcoming of the approach when it comes to analyzing reports is the lack of actionable test cases to reach the involved code. While this is an inherently hard problem for any static analysis, the patterns that we propose are quite intuitive, greatly helping manual analysis.

The majority of false positives come from pattern ❶, primarily because the static analysis is currently unable to recognize explicit type checks on structure fields that act as runtime type information and prevent container confusion bugs (Section 7.3). Therefore, violations of pattern ❶ can be regarded with lower confidence compared to the other patterns.

False negatives in the static analysis may be caused by incomplete control-flow information (e.g., indirect calls) and by inaccuracies in our modeling of program state. For example, precise modeling of memory may be an area worth examining to improve the reach of the static analyzer. We opted not to use pointer analyses as accurate ones are expensive on large programs [56] and features desirable in this context (e.g., flow- and context-sensitivity) would increase their costs considerably. Moreover, they would be unaware of the many indirect control transfers to functions caused by userland activities. We leave this investigation to future work.

Similarly, it would be interesting to explore directed fuzzing [3] and/or fuzzers specialized for certain kernel areas (Section 6) to reach functions/regions where static analyses report potential container confusion cases. Doing so may enable both their in-depth exploration and input generation for some reports.

The security impact of type confusion bugs depends on the memory layout of the objects involved. In an exploitation scenario, an attacker would leverage a controlled type confusion to overlap and corrupt interesting fields. On the other hand, the type confusion bugs found by our approach have no control over which types overlap. This may influence the immediate exploitability of the bugs we found and require more effort to turn a type confusion into memory corruption. However, 8 of our bugs were considered security-relevant for their exploitability and got assigned 6 CVEs (3 bugs got merged into the same CVE, as listed in Appendix A). As a concrete illustration of security impact, we have also demonstrated a controllable out-of-bounds write on the heap for one of the CVEs reported.

9 Related Work

This section covers literature on type confusion, sanitization, and static analysis that the research in this paper relates to.

Type Confusion Detection. Most existing type confusion detectors are limited to C++. UBSan [40], for instance, replaces static casts with dynamic casts in C++ to expose bugs. CaVeR [39], TypeSan [21], HexType [29], and Bitype [46] are specialized to find type confusion for C++ classes by managing runtime type metadata and performing checks on cast operations. CASTSan [42] efficiently detects type confusion leveraging C++ virtual tables, but is limited to polymorphic classes only. While all other existing approaches rely on dynamic analysis, TCD [62] uses a field-, context- and flow sensitive pointer analysis to detect type-confused C++ code.

libcrunch [31] and EffectiveSan [13] support C programs. However, both approaches rely on intercepting object allocations and binding them with their top-level allocation type. In practice, this would be hard, if not impossible, to collect in projects with the complexity of a kernel. For this reason, the typed allocator mitigation in XNU resorted to manual annotations in allocations [15]. Our approach overcomes the need of both allocation-time type inference and manual annotations.

Speculative Type Confusion. Previous work has explored speculative type confusion while dealing with objects of multiple types. Confusion in the speculative domain fundamentally differ from non-speculative one for observability and/or explainability. Kasper [30] scans the Linux kernel for arbitrary speculative gadgets. It shows how the current list iterator implementation is subject to speculative container confusion when dealing with the list heads if the terminating condition is mispredicted. Kirzner et al. [32] focus on speculative type confusion in the Linux kernel. The paper highlights possible type confusion originating from eBPF code, compiler-introduced vulnerabilities, and polymorphic types. BHI [2] leverages a speculative type confusion in eBPF code in their exploit. FPVI [48] and Spook.js [1] exploit speculative type confusion in JavaScript engines.

Other Sanitizers. Similarly to ASan [50], several sanitizers rely on redzones: Purify [25], Memcheck [51], Dr. Memory [6] and LPC [24] leverage them to detect memory corruptions in the form of spatial and temporal safety violations.

MSan [55] targets reads from uninitialized memory using a shadow map mechanism. Other sanitizers, such as Undangle [7], FreeSentry [60], DangNull [38], and DangSan [59] detect dangling pointers that cause use-after-free errors.

For boundary identification, other techniques encode tracking metadata within pointers, as with low-fat pointers [14, 36] and delta pointers [35]. For example, our approach could replace redzones with low-fat pointers on supported systems.

Static Analyzers. We conclude by mentioning a few popular static analysis tools for the Linux kernel. Coccinelle [44] is pervasively used as a program matching and transformation tool. In addition to its use for refactoring and code hardening, it also has provisions to find intra-procedural bugs. Sparse [5] uses Linux kernel-specific annotations to perform few specialized checks. Smatch [4] followed in its footsteps to build a generic static analysis framework for several kernel bug types; it can only conduct intra-procedural dataflow analyses.

10 Conclusion

We presented a sanitization scheme for container confusion designed as a compiler-based runtime checker. For demonstration, we implemented the sanitizer for the Linux kernel, finding 11 bugs, which were undetected by previous work. Those bugs have often existed in the kernel for several years. Based on our results, we identified common bug patterns and used those categories to build a tailored static analyzer to discover bugs in code often unreachable by dynamic analysis. With our static analyzer, we unveiled 78 additional, previously undiscovered bugs. We conclude that bad downcasting is not only problematic in object-oriented programming languages but also occurs in large C projects, with serious security impact.

We have disclosed and proposed possible fixes for all found bugs and relevant anti-patterns to the Linux kernel mailing list, with a total of 149 patches and 102 already merged. Some of the disclosed issues have prompted significant changes to core kernel design patterns, with fixes even requiring the kernel to transition to the modern C11 standard.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported by Intel Corporation through the “Allocamelus” project, the Dutch Ministry of Economic Affairs and Climate through the AVR program (“Memo” project), the Dutch Science Organization (NWO) through projects “TROPICS”, “Theseus”, and “Intersect”.

References

[1] Ayush Agarwal, Sioli O’Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking Chrome strict site isolation via speculative execution. In *S&P*, 2022.

[2] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security Symposium*, 2022.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *CCS*, 2017.

[4] Niel Brown. Smatch: pluggable static analysis for c. https://lwn.net/Articles/691882/.

[5] Niel Brown. Sparse: a look under the hood. https://lwn.net/Articles/689907/.

[6] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *CGO*, 2011.

[7] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[8] Kees Cook. Bounded flexible arrays in c. https://people.kernel.org/kees/bounded-flexible-arrays-in-c, 2023.

[9] Jonathan Corbet. Moving the kernel to modern C. https://lwn.net/Articles/885941/, 2022.

[10] Jonathan Corbet. Toward a better list iterator for the kernel. https://lwn.net/Articles/887097/, 2022.

[11] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *CCS*, 2017.

[12] Arnaldo Carvalho De Melo. Profiling data structures. https://lpc.events/event/16/contributions/1200/attachments/1054/2013/Profiling%20Data%20Structures.pdf, 2022.

[13] Gregory J. Duck and Roland H.C. Yap. EffectiveSan: Type and memory error detection using dynamically typed C/C++. In *PLDI*, 2018.

[14] Gregory J. Duck, Roland H.C. Yap, and Lorenzo Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *NDSS Symposium*, 2017.

[15] Apple Security Engineering and Architecture. Towards the next generation of xnu memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/, 2022.

[16] Cristiano Giuffrida, Călin Iorgulescu, and Andrew S. Tanenbaum. Mutable checkpoint-restart: Automating live update for generic server programs. In *Middleware*, 2014.

[17] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security*, 2012.

[18] Google. syzbot dashboard. https://syzkaller.appspot.com.

[19] Google. syzkaller. https://github.com/google/syzkaller.

[20] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*, 2016.

[21] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. TypeSan: Practical type confusion detection. In *CCS*, 2016.

[22] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. Demystifying the dependency challenge in kernel fuzzing. In *ICSE*, 2022.

[23] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *TOPLAS*, 16(2):175–204, 1994.

[24] Niranjan Hasabnis, Ashish Misra, and R Sekar. Lightweight bounds checking. In *CGO*, 2012.

[25] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *Proceedings of USENIX Winter'92 Conference*, 1992.

[26] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[27] Nur Hussein. Randomizing structure layout. https://lwn.net/Articles/722293/, 2017.

[28] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *CCS*, 2016.

[29] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient detection of type confusion errors for C++. In *CCS*, 2017.

[30] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for generalized transient execution gadgets in the Linux kernel. In *NDSS Symposium*, 2022.

[31] Stephen Kell. Dynamically diagnosing type errors in unsafe code. In *OOPSLA*, 2016.

[32] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security Symposium*, 2021.

[33] Andrey Konovalov and Dmitry Vyukov. KernelAddressSanitizer (KASan): a fast memory error detector for the Linux kernel. *LinuxCon North America*, 2015.

[34] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: breaking KASLR on the isolated kernel address space using tagged TLBs. In *Euro S&P*, 2020.

[35] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *EuroSys*, 2018.

[36] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *CCS*, 2013.

[37] Paul Larson. Testing Linux with the Linux test project. In *Ottawa Linux Symposium*, 2002.

[38] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS Symposium*, 2015.

[39] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium*, 2015.

[40] LLVM. UndefinedBehaviorSanitizer - Clang documentation. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[41] Larry W McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[42] Paul Muntean, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert. CastSan: Efficient detection of polymorphic C++ object type confusions with LLVM. In *ESORICS*, 2018.

[43] open std. Defect report 051. https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_051.html, 1993.

[44] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. *ACM SIGOPS Operating Systems Review*, 2008.

[45] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *USENIX Security Symposium*, 2018.

[46] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. Mapping to bits: Efficiently detecting type confusion errors. In *ACSAC*, 2018.

[47] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *USENIX Security Symposium*, 2020.

[48] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security Symposium*, 2021.

[49] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security Symposium*, 2017.

[50] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[51] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[52] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *USENIX Security Symposium*, 2022.

[53] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-OS boundary. In *NDSS Symposium*, 2019.

[54] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *USENIX Security Symposium*, 2020.

[55] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO*, 2015.

[56] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *CC*, 2016.

[57] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *USENIX Security Symposium*, 2018.

[58] Erik Van Der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-after-type: Practical and complete type-safe memory reuse. In *ACSAC*, 2018.

[59] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *EuroSys*, 2017.

[60] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS Symposium*, 2015.

[61] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System call-based state-aware Linux driver fuzzing. In *USENIX Security Symposium*, 2022.

[62] Changwei Zou, Yulei Sui, Hua Yan, and Jingling Xue. TCD: Statically detecting type confusion errors in C++ programs. In *ISSRE*, 2019.

A Assigned CVEs

Table 2 presents the list of CVE identifiers assigned to the type confusion bugs we reported.

B LMbench Evaluation

Table 3 presents detailed results for the LMbench tests mentioned in Section 5.3.2.

CVE	**Description**
CVE-2023-1073	Type confusion in `hid_validate_values()`, Type confusion in `bigben_probe()`, NULL pointer dereference in `betopff_init()`
CVE-2023-1074	KASLR leak in `inet_diag_msg_sctpasoc_fill()`
CVE-2023-1075	Type confusion in `tls_is_tx_ready()`
CVE-2023-1076	Incorrect UID assigned to tun/tap sockets
CVE-2023-1077	Type confusion in `pick_next_rt_entity()`
CVE-2023-1078	Heap OOB write in `rds_rm_zerocopy_callback()`

Table 2: CVEs assigned to the reported type confusion bugs.

C Static Analysis Rules

Table 4 shows the definitions for our static information flow analyses. For each pattern, we report the source where the dataflow starts from, the sinks that the dataflow searches, the path filters that inhibit the report (i.e., stop path exploration) when met, and additional checks that the analysis performs at a sink before reporting a potential container confusion.

Benchmark	baseline	UNCONTAINED	KASAN	UNCONTAINED overhead	KASAN overhead
Simple syscall	1.05 μs	1.21 μs	1.93 μs	16 %	84 %
Simple read	1.28 μs	1.64 μs	2.32 μs	28 %	82 %
Simple write	1.02 μs	1.24 μs	1.83 μs	21 %	79 %
Simple stat	8.34 μs	72.10 μs	37.59 μs	764 %	351 %
Simple fstat	5.01 μs	59.24 μs	21.24 μs	1083 %	325 %
Simple open/close	18.14 μs	86.89 μs	66.97 μs	379 %	269 %
Select on 10 fd's	2.05 μs	2.41 μs	3.68 μs	18 %	80 %
Select on 100 fd's	6.29 μs	6.79 μs	9.07 μs	08 %	44 %
Select on 250 fd's	13.38 μs	14.13 μs	18.06 μs	06 %	35 %
Select on 500 fd's	25.79 μs	29.10 μs	38.73 μs	13 %	50 %
Select on 10 tcp fd's	2.19 μs	2.55 μs	3.95 μs	17 %	81 %
Select on 100 tcp fd's	11.85 μs	12.74 μs	19.37 μs	07 %	63 %
Select on 250 tcp fd's	28.23 μs	29.83 μs	45.37 μs	06 %	61 %
Select on 500 tcp fd's	56.05 μs	61.16 μs	95.02 μs	09 %	70 %
Signal handler installation	1.32 μs	1.57 μs	2.46 μs	19 %	87 %
Signal handler overhead	4.75 μs	7.65 μs	14.51 μs	61 %	206 %
Pipe latency	16.58 μs	20.99 μs	39.54 μs	27 %	139 %
AF_UNIX sock stream latency	22.71 μs	38.03 μs	74.32 μs	67 %	226 %
Process fork+exit	627.32 μs	1076.48 μs	1869.73 μs	72 %	197 %
Process fork+execve	718.54 μs	1210.79 μs	2099.22 μs	69 %	191 %
Process fork+/bin/sh -c	2530.20 μs	5370.25 μs	6756.88 μs	112 %	167 %
UDP latency using localhost	44.56 μs	135.34 μs	106.43 μs	204 %	139 %
TCP latency using localhost	56.33 μs	113.18 μs	141.90 μs	101 %	152 %
TCP/IP connection cost to localhost	240.82 μs	494.53 μs	672.52 μs	105 %	179 %
geomean				74 %	126 %

Table 3: LMbench experiments: comparing the native execution baseline against UNCONTAINED and KASAN.

Bug Pattern	Direction	Source	Sink	Filters	Checks
❶ Statically Incompatible Containers	B	`container_of()` input	Origin object of input pointer		Mismatch between `container_of()` destination type and origin type
❷ Empty-list Confusion (rule 1)	F	`list_entry()` result	Any use	Conditional Checks	
❷ Empty-list Confusion (rule 2)	F	`list_entry()` result	Comparison with NULL	Flows with explicit NULL values	
❸ Mismatch on Data Structure Operators	F	Any list operation (e.g. `list_add()` or `list_entry()`)	Any list operation (e.g. `list_add()` or `list_entry()`)		Mismatch between member field/type used
❹ Past-the-end Iterator	F	Any iterator variable used in a loop over a list, e.g., `list_for_each_entry()`	Any use after the loop	Checks on *found*-like variables	
❺ Containers with Contracts (backwards part)	B	Arguments of `kobject_init_and_add()`	Collect containing structure of the `kobject` and `sysfs_ops` functions		
❺ Containers with Contracts (forward part)	F	`kobj` argument of collected `sysfs_ops` functions	`container_of()`		Mismatch between collected containing structure of the `kobject` and `container_of()` destination type

Table 4: Details of rules for the patterns defined for static analysis. Showing the direction (B for backwards dataflow, F for forward dataflow), source and sink matched, and eventual filters and/or additional checks. ❺ employs a single rule in two parts.