Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning

Navid Emamdoost
University of Minnesota
navid@cs.umn.edu

Qiushi Wu
University of Minnesota
wu000273@umn.edu

Kangjie Lu
University of Minnesota
kjlu@umn.edu

Stephen McCamant
University of Minnesota
mccamant@cs.umn.edu

***Abstract*—The kernel space is shared by hardware and all processes, so its memory usage is more limited, and memory is harder to reclaim, compared to user-space memory; as a result, memory leaks in the kernel can easily lead to high-impact denial of service. The problem is particularly critical in long-running servers. Kernel code makes heavy use of dynamic (heap) allocation, and many code modules within the kernel provide their own abstractions for customized memory management. On the other hand, the kernel code involves highly complicated data flow, so it is hard to determine where an object is supposed to be released. Given the complex and critical nature of OS kernels, as well as the heavy specialization, existing methods largely fail at effectively and thoroughly detecting kernel memory leaks.**

In this paper, we present K-MELD, a static detection system for kernel memory leaks. K-MELD features multiple new techniques that can automatically identify specialized allocation/deallocation functions and determine the expected memory-release locations. Specifically, we first develop a usage- and structure-aware approach to effectively identify specialized allocation functions, and employ a new rule-mining approach to identify the corresponding deallocation functions. We then develop a new ownership reasoning mechanism that employs enhanced escape analysis and consumer-function analysis to infer expected release locations. By applying K-MELD to the Linux kernel, we confirm its effectiveness: it finds 218 new bugs, with 41 CVEs assigned. Out of those 218 bugs, 115 are in specialized modules.

I. Introduction

An operating system (OS) kernel is part of the trusted computing base (TCB) in most modern software systems. Vulnerabilities in the OS kernel allow an adversary to bypass security measures and compromise the whole system. Much of the research in the security community has focused on memory-corruption bugs in the kernel [14, 19, 28, 47, 49]. At the same time resource exhaustion vulnerabilities in the kernel have been subject to less attention, though they too can have severe consequences on system stability and availability [5, 33, 35]. Based on our study, we found out that just 17 CVEs were assigned to Linux kernel memory leak bugs. 10 of these vulnerabilities were discovered in the past two years.

Memory is a primary resource available to a kernel, and it may be exhausted by memory leak vulnerabilities. A memory leak happens when an allocated memory region is not released even though it will never be used again. A memory region is definitely leaked when all the pointers to the allocated memory go out of scope or are overwritten. Leaked memory becomes unusable until the system reboots. Kernel memory is typically never swapped out, so a kernel memory leak reduces the physical memory available for any other purpose. Due to increased paging, a memory leak can hurt performance [11] and eventually exhaust all the available memory. Kernel memory leaks happen mostly on error-handling paths mainly because such paths are less exercised during tests. If a path associated with a memory leak gets executed frequently, the triggered memory leak eventually causes a denial of service. Furthermore, many small or infrequent memory leak bugs may lead to the same situation [4, 8]. Of course, even if a memory leak occurs only very rarely in normal operation of a system, a malicious user may find a way to trigger a leak with high frequency.

For example, Figure 1 depicts a memory leak bug that was discovered by our tool which was assigned `CVE-2019-19062`. Here the function `crypto_report()` at line 13 allocates the memory via `nlmsg_new` for a netlink message. Pointer to this message is stored in `skb`. The code at line 14 checks whether the allocation succeeded or not. If the allocation was not successful, execution returns at line 15. If the allocation was successful, the execution continues up to the line 19, where function `crypto_report_alg()` is called. The status of the call to `crypto_report_alg()` is checked at line 24. If it was successful, the `skb` is consumed in lines after 26, but if `crypto_report_alg()` fails, the function returns at line 25 without releasing `skb`, which is a memory leak.

Researchers have shown that an unprivileged user can exploit memory-leak vulnerabilities to cause denial-of-service. For example, Saha et. al [41] have shown cases of exploiting such a class of vulnerabilities in the Linux kernel causing memory exhaustion. As a result the system fails to allocate any new memory region which in turn leads to failure in unpredictable ways. The situation is not recoverable except via system reboot. Those memory leaks are related to error-handling paths that mistakenly miss releasing a dynamically allocated object in the Linux file system. An unprivileged user could exploit such bugs by triggering the error-handling path by providing invalid inputs to a system call or mounting a faulty file system.

Challenges. Two key challenges exist in static memory leak detection for an OS kernel. (1) *Specialized functions*. Monolithic OS kernels like the Linux kernel tend to contain tens of thousands of different modules developed by numerous vendors and programmers. We consider a function as specialized if it

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24416
www.ndss-symposium.org

is developed for a specific module to perform a customized flavor of generic operation (like allocation/deallocation of a network buffer). As a result, an OS kernel has a large number of specialized functions for memory allocation and deallocation. An effective detection requires identifying such specialized allocation functions and the corresponding deallocation functions. While there are only a handful of commonly used allocation functions, we found more than 800 specialized ones. (2) *Complicated and lengthy data flow*. A memory leak occurs when a memory object is not released at the end of the object life-cycle. In other words, an effective detection often has to analyze a lengthy data flow—from allocation to the end of life-cycle. More importantly, the lengthy data flow can be highly complicated in an OS kernel: Pointers to allocated memory object are also often copied between different data structures across functions. An effective detection must determine which location or function is responsible to release the memory object[1]. We say a memory pointer is an *escaping pointer* if it is passed to the caller of the current function and thus can be freed outside the scope of the current code. A *consumer function*, on the other hand is a callee of current function that takes the ownership of the memory object, therefore the current function should not try to release after returning from the consumer function. An analysis needs to accurately recognize escaping pointers and consumer functions to avoid false-positive memory leak reports.

Given the challenges for static leak detection in an OS kernel, one may wonder if a dynamic approach would yield better results. We argue that a static detection works better because there is no need to provide real or synthetic inputs to trigger all potential execution paths. Moreover, for specialized drivers, the specific hardware should be available to be able to exercise the driver code. Because of such inherent code coverage shortcomings of dynamic approaches, we opt for a static leak detection.

Researchers have attempted to statically detect kernel memory leaks; but their techniques have important limitations. In particular, general bug finding tools like Coccinelle [36] have limited effectiveness for detecting kernel memory leaks. Coccinelle does not implement any specific bug-finding policy but allows specifying patterns to search for potentially faulty code blocks in a function's CFG. For example, to find a memory leak a general pattern could be like: any allocation function should be followed by a deallocation. Such a high-level pattern yields a high rate of false positive. Because of the cost of manual rule specification, these tools have been applied just to general purpose allocation functions that have well-known deallocation counterparts (like `kmalloc-kfree`). In addition, previously proposed systems that detect resource release bugs [41, 48] either lack support for specialized allocations, or fail to effectively handle escaping pointers and consumer functions.

In this paper, we introduce K-MELD (**K**ernel **Me**mory **L**eak **D**etector), a static analysis tool, to detect kernel memory leaks. K-MELD not only identifies specialized allocation functions and the corresponding deallocation functions, but also answers where an object is supposed to be released by handling the complicated and lengthy data flow. K-MELD features

[1]Incorrectly placed releases introduce severe memory corruption bugs like use-after-free or double-free.

```
1  /* File: crypto/crypto_user_base.c */
2  static int crypto_report(struct sk_buff *in_skb,
3          struct nlmsghdr *in_nlh, struct nlattr **attrs)
4  {
5      struct crypto_dump_info info;
6      ...
7      alg = crypto_alg_match(p, 0);
8      if (!alg)
9          return -ENOENT;
10
11     err = -ENOMEM;
12     /* Memory is allocated here */
13     skb = nlmsg_new(NLMSG_DEFAULT_SIZE, GFP_KERNEL);
14     if (!skb)
15             goto drop_alg;
16     ...
17     info.out_skb = skb;
18     ...
19     err = crypto_report_alg(alg, &info);
20
21 drop_alg:
22     crypto_mod_put(alg);
23
24     if (err)
25             return err; /* Memory leaks here */
26      ...
27 }
```

Fig. 1: A new memory leak bug (CVE-2019-19062) detected by K-MELD: the path ending at line 25 needs to release `skb`.

multiple new techniques. First, K-MELD identifies specialized allocation functions by using a usage-driven and structure-aware analysis, then uses a context-aware and path-sensitive mining technique to detect corresponding release functions. More specifically, the initial set of allocation functions are populated based on the type and uses of the return value and the way such value is derived. Then, assuming the faulty dynamic allocation managements are outliers [10], the high-level intuition is modeling the common approach of memory releasing to effectively identify memory release operations even in specialized cases. In a big corpus of code like a kernel[2], there are many potentially long execution paths. It has been shown that error-handling paths get less of testing and coding review [2, 30]. Based on our observation from manually checking the sites of memory mismanagement bugs, error-handling codes were where we could spot the bugs. In addition, error-handling code are usually shorter than normal execution paths and are intended to restore the state of system from an error. That means we can expect to find the release functions in error-handling paths if the function is supposed to release memory. the number of correctly implemented error-handling paths is much more than erroneous ones. Therefore, modeling the common behavior of error-handling paths enables us to single out potentially buggy implementations.

Second, we develop an ownership reasoning mechanism to infer the release locations. K-MELD first uses enhanced escape analysis to determine when the ownership of an allocated object is transferred. We observed that kernels typically follow an informal discipline of "ownership" of dynamically allocated data (related to the concept codified in languages like Rust, but without type-system support). A function that allocates a memory object will either be responsible for deallocating that object itself, or if the object is made available to the caller via a return value or a pointer, the calling function also takes the responsibility to deallocate the object. Based on this pattern, our tool can avoid false positives by determining when

[2]For example the Linux kernel is over 27 MLOC.

objects can *escape* to a calling function: we do not expect a function to deallocate an object on a path where the object escapes. In such scenarios, the calling function is responsible to appropriately manage the allocated memory object. On the other hand, the allocating function may also pass a memory object down the call-graph to a callee, thus it is important to have an inter-procedural analysis to determine how the callee treats the memory object. We call a function *consumer* if it releases or the received memory object or allows it to escape on all or some of its execution paths. This is important because once returning from the callee, the allocating function should not try to release the memory object if already released. The analysis requires to handle conditional consumers as well. These are functions that consume the memory object under certain conditions: consider a socket packet-send function that consumes the allocated socket buffer only in the case of a successful send. In such case, the caller of packet-send has to release the buffer if sending fails. K-MELD employs an inter-procedural path-sensitive analysis to track memory object propagation and identify consumer functions.

We implemented our tool as multiple LLVM passes, rule mining, and rule application. We conservatively identified more than 800 memory allocation, most of which are specialized ones, and the associated release functions. After the ownership reasoning via inter-procedural analyses, and rule application, we detect 218 new memory leaks bugs even in many specialized modules. Our evaluation also confirmed that our ownership reasoning mechanisms significantly improves the accuracy.

Contributions We make the following contributions:

- **An approach for identifying specialized allocation functions.** We develop a usage- and structure-aware approach to effectively identify both common and specialized allocation functions. The approach automatically identified more than 800 specialized allocation functions.
- **A rule-mining approach for corresponding specialized deallocations.** We develop a context-aware approach for mining rules and identifying specialized release functions. The mining can differentiate erroneous and normal execution contexts. We also employ field- and path-sensitive static analyses that track data flows to enhance the rule mining. Among the bugs found, 115 are caused by missing specialized deallocation functions. Note that the identification of specialized allocations and deallocations is of independent interest to the detection of other classes of bugs such as NULL dereference, out-of-bound access, and use-after-free.
- **An ownership reasoning mechanism for kernel objects.** We propose an ownership reasoning mechanism to infer where an object is supposed to be released. We first develop an *path-sensitive pointer escape analysis* to determine if the ownership of an allocated memory object is transferred. This analysis reduces false positives significantly. Moreover, we develop an *efficient consumer function detection* to precisely determine if the allocating function is responsible to release the allocated memory. This analysis employs an inter-procedural data flow and control flow analysis to track the propagation of the allocated memory object downwards in the call-graph.
- **A scalable implementation and numerous new bugs.** We implement a scalable system, K-MELD, and apply it to the Linux kernel. K-MELD detects a large number (218) of new memory leak bugs with an acceptable false positive rate (52%). 41 of the new bugs have been assigned with CVEs, confirming the security impacts of kernel memory leaks. 115 of the bugs were located in specialized modules.

II. A Study of Kernel Memory Allocation and Leaks

In this section, we first take a look at the dynamic memory allocation mechanism in a kernel (using the Linux kernel for concreteness). Then we review the importance of memory leaks in the kernel and the challenges for leak detection.

A. Dynamic Memory Allocation in the Kernel

Types of memory allocation. Dynamic memory allocation in the kernel is not as straightforward as in user-space. The complication comes mainly from the fact that the kernel has much less physical memory available—generally kernel memory is not pageable, and often the allocation is required to be a physically continuous memory region and sometimes in specific address ranges. The kernel allocation sometimes needs to be atomic i.e. it should not sleep. Such delicacies make any mistake in kernel allocation have a higher impact on the whole system's stability.

OS kernels typically allocate only a relatively small stack per thread.[3] This limitation requires kernel developers to avoid allocating large structures on the stack but instead, to perform more heap-based allocation. `kmalloc` is the general-purpose allocation interface in the kernel. It takes two arguments: `size` and `flags`. Like user-space `malloc`, `size` specifies the allocation size in bytes. The `flags` argument controls the behavior of the allocation. On success `kmalloc` returns a pointer to the memory of `size` bytes, while in case of failure a NULL pointer is returned. The flags parameter in `kmalloc` tells the kernel how or where to perform the allocation.

As an example, the flag `GFP_ATOMIC` instructs the memory allocator never to block, i.e., it cannot go to sleep to free up the required memory. This is appropriate when the code holds a lock, or in interrupt handlers. On the other hand, the flag, `GFP_KERNEL`, indicates the normal kernel allocation which may go to sleep. As another example, flag `GFP_DMA` instructs the memory allocator to allocate from the physical address range which is accessible by the hardware through direct memory access. The header `<linux/gfp.h>` defines all the allocation flags.

`kmalloc` returns physically continuous memory. Such an allocation has two main advantages over virtual memory allocations. First, it can be used by the hardware as non-CPU devices use physical addressing. Second, physically continuous memory can be allocated within a single large page and as a result, be faster from memory translation perspective. However, allocation via `kmalloc` has a higher chance of failure for large sizes. Therefore if there is no need for physically continuous memory, or if the allocation size is large, `vmalloc` should be used. `vmalloc` uses page table manipulation to create a virtually continuous memory region. It also may block when allocating, and therefore cannot be used in an interrupt handler.

[3] On Linux, kernel thread stacks vary by architecture, but are commonly 1, 2, or 4 pages, so 4–16 KiB.

To avoid memory fragmentation, especially when many identical objects should be allocated, a slab cache can be used [3]. The cache should be set up via `kmem_cache_create` and the allocation from the cache is realized via `kmem_cache_alloc`. For example, Linux maintains separate caches for inode, dentries, and buffer heads [22].

The dynamically allocated memory should be explicitly released when it has no further usage. Otherwise, the memory is leaked and eventually, no further allocation will be possible. The allocations via `kmalloc` should be released via `kfree` which takes the pointer to the memory to be released and returns the memory to the kernel. Allocations via `vmalloc` should be released by `vfree`, and slab cache allocations via `kmem_cache_free`.

Specialized allocation. Various kernel modules have their own specialized allocators. Such allocators are responsible to allocate and sometimes initialize a specialized structure. The memory allocation is usually realized via a more primitive allocator, and after some initialization or extra operations, a pointer is returned to the caller. Such allocations require a specialized release and are not deallocated just by `kfree`. For example, the `netlink`[4] module uses `nlmsg_new` to allocate a new message and uses `nlmsg_free` to release such message. Such specialization particularly imposes challenges to the detection of memory leaks because the detection must identify allocators and the corresponding deallocators.

B. Memory Leaks in the Kernel

Memory-leak bugs in the kernel are considered security critical [35]. That is because the amount of memory available to the kernel is highly limited, and thus the kernel is susceptible to memory exhaustion, leading to denial-of-service (DoS) [33]. Worse, a DoS in OS kernels is typically unacceptable because it results in the whole system unavailable, which is particularly critical for long-running servers. The DoS may further result in other critical issues such as data losses or crash inconsistencies.

Memory-leak bugs become vulnerabilities when they are triggerable by attackers. By repeatedly triggering a memory-leak bug, attackers can eventually exhaust available kernel memory and hang the whole system.

In order to detect the bug in Figure 1, K-MELD first confirms the memory is successfully allocated via customized allocator `nlmsg_new`, then it confirms the assignment at line 17 does not cause `skb` to escape (because the struct `info` is a local variable). Then K-MELD checks if the call to `crypto_report_alg` is consuming `skb`. In this case, the memory object is not consumed, so it means any execution path following the line 19 should manage `skb` properly. This is not the case for the execution path ending at line 25, so K-MELD reports it as a bug.

III. OVERVIEW OF K-MELD

The goal of K-MELD is to thoroughly and precisely detect memory-leak bugs in kernel code with static analysis. To identify a memory-leak bug in a function, we model memory leak as a case satisfying the following conditions:

1) A function retains ownership of the allocated memory
2) The function finishes without releasing the allocated memory

Originally, the allocating function is the owner of the memory object. However, the ownership of the object would likely propagate to other functions, e.g., the pointer to the memory object is passed to other functions (e.g. via return or parameters). As long as a function owns the memory region, the function is supposed to release the memory region; failure to do so is a memory leak. According to the modeling, we structure K-MELD into three phases, as shown in Figure 2: allocation/deallocation identification, ownership propagation analysis, and missing deallocation detection.

As shown in Figure 2, K-MELD compiles the source code into LLVM IR bitcode files. Such bitcode files are first preprocessed to extract contextual and structural properties. More specifically, we collect return type, argument signature, definition and usages of functions. At multiple points of our analyses, a call graph is used to identify the set of callees or callers for each function. Therefore in the preprocessing step, the global call graph is constructed via maintaining a map of function and all of its callees. To resolve indirect calls, we use function signatures to match the target function from among functions whose address is taken. The preprocessing phase is performed once.

Phase 1: allocation/deallocation identification. we identify potential allocation functions by looking for pointer-returning functions that are followed by a null-check on the returned pointer. Additionally, we prune any function that is offsetting the returning pointer from an arguments. We also track the usage of the returned pointer to determine initialization before being de-referenced. Once the allocation functions are identified, deallocation functions are identified via rule mining. We refer to the current allocation function as the "function of interest" (FOI), while the function that calls the FOI is the "allocating function."

The high-level rationale behind using rule mining is that assuming the correct behavior is prevalent in a big code basis like the OS kernel, we can model such common behavior in the form of sequential patterns. Rule mining uses sequential pattern discovery techniques to identify the sequence of commonly used operations on the memory object. This way for a specific FOI we can identify the common sequence of operations that are used for deallocation. For example, considering *kmalloc* as FOI, by looking through error handling paths of *kmalloc* call-sites we can observe the function *kfree* is called in most cases before terminating the execution path. Thus one can conclude that *kfree* is the associated deallocation function.

Phase 2: ownership propagation analysis. Recalling the two conditions proposed earlier in this section, condition (1) needs to know if the subject call-site is the owner of the allocated memory object or not. On one hand, the escape analysis tracks the propagation of the memory object upwards in call-graph beyond the allocating function boundaries. On the other hand, the consumer function detection tracks the propagation of memory object downwards in call-graph. These analyses determine if the subject call-site is still the owner of memory object. Such analyses are realized via a customized data-flow analysis. They need to be path-sensitive, to differentiate the conditional escape or consumption. They also need to be

[4] include/net/netlink.h

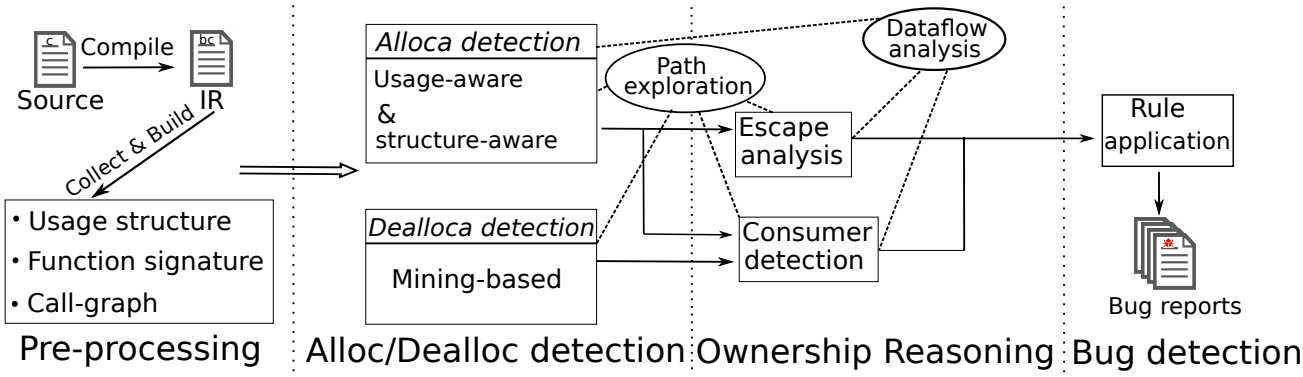

Fig. 2: Overview of K-MELD. Identifying specialized allocation functions and the corresponding deallocatoin is followed by inter-procedural escape and consumer function analysis to determine the ownership of each allocation. Rule application checks the presence of deallocation and reports the bugs.

field-sensitive to be able to track the pointer propagation via struct fields. Such analysis tracks the def-use chain of an allocated memory object on each execution path. The escape analysis determines if the memory pointer is copied beyond the allocating function (via a reference argument or global variable). The consumer function determines if any callees of the allocating function are releasing the memory object or causing it to escape. In either case, the allocating function is not the owner of the memory object anymore. Otherwise, the allocating function is responsible for appropriately releasing the memory object before returning.

Phase 3: missing deallocation detection. Once the previous phases finish, we can evaluate condition (2) for the specific FOI. That is any potential execution path missing appropriate release function is a potential memory leak bug. The context-aware Rule Mining identifies appropriate release functions associated with each FOI, and then escape analysis and consumer function detection analysis reason on the owner of allocated memory object. For each FOI, any absence of associated deallocation indicates a potential memory leak bug.

In the following sections we explain the core design components of K-MELD which are specialized allocation/deallocation function detection and ownership reasoning.

IV. Allocation and Deallocation Identification

A. Identifying Allocators (FOIs)

The first challenge K-MELD overcomes is identifying allocation functions, both generic and specialized ones. Manual identification of allocation functions in an OS kernel is not practical. That is because various kernel modules have their own specialized allocation functions. We need an automated way to be able to collect a set of allocations including the specialized ones.

Observations. We observe that memory allocation is a critical operation, which in case of success returns a pointer to the allocated memory region, whilst in case of failure it returns a NULL pointer. Because of this, the allocation functions are followed by a null-check on their return value. When a memory object is allocated, it requires initialization before being used effectively. An initialization is realized either via an `store` instruction or a call to `memcpy/memset` with the allocated pointer as destination. Such initialization must be performed before any read from the memory object.

To summarize, an allocation function has the following properties.

- It returns a pointer.
- The pointer is immediately followed by a NULL check.
- The pointer is not derived from another base pointer.
- The object is initialized before being used, e.g., being read.

Removing noisy functions. Besides the allocator functions, there is another class of functions that return a pointer and may be followed by a null-check. A *getter* function is a function that produces a pointer out of one of its pointer arguments either by indexing or accessing a field in struct. In order to exclude getter functions from the set of initial allocators, we first profile all pointer-returning functions in the kernel and mark those that their returning pointer is derived (calculated or accessed via `GetElementPtr`) from one of pointer arguments as base pointer.

Characterizing Pointers. We evaluate the aforementioned properties via use-finding and source-finding data flow. Use-finding is a forward data flow tracking that determines the operations on a pointer. For example use-finding helps to identify any null-check on a pointer. Source-finding is a backward data flow tracking that determines what is the source of pointer. This, for example helps in determining if a pointer is offset of a base pointer.

At this stage, our analysis tends to be more relaxed in terms of the number of null-check or initialization. More specifically, first we look for pointer returning functions which are not profiled as getter function (i.e. offset returning) and are followed by null-check and initialized in majority of their call-sites. This set will be further refined when deallocation detection is finished. As will be shown in §VII, our approach works well; While it identifies the common allocation functions we are aware of, it also identifies specialized allocation functions.

B. Context-aware Rule Mining for Deallocation Detection

After we identified allocation functions, we next identify the corresponding deallocation functions. To do so, we use sequential pattern mining on the error-handling operations.

Identifying and Collecting Error-Handling Paths. It is a convention that failure of an operation in an OS kernel is reflected in a return status, which in turn should be check by the caller. The error handling is responsible for recovering from errors and preventing the system from entering an unstable or undefined state. Error-handling paths are relatively short, and have clean and important operations to recover from the failure, this makes it a great place to identify paired operations (e.g. release, unlock or close).

To identify error handling paths at each FOI call-site, we first extract intra-procedural execution paths. To do so, we statically explore the control flow graph (CFG) in a depth-first fashion. Each path starts from the function's entry basic block and ends in a basic blocks with return instruction as terminator. Loops are unrolled for just one level. We also filter out the execution paths that do not go through the FOI call. In most cases the FOI return status is checked[5]. If the call to the FOI fails, the corresponding failure branch will not continue to use the resources (e.g., a pointer) returned by the FOI. Therefore, any paths following the FOI failure branch are not interesting to us because there will be no memory leaks. For example, the path going through line 15 of Figure 1 cannot lead to a memory leak of `skb`.

From the set of FOI success paths, we then identify error handling paths. If a path explicitly returns an error code (similar to those used in `errno` in user space, though kernel conventions use negative values) or NULL, it is an error-handling path. Not all error-handling paths are simple to identify. In most cases, the return status of an operation is checked, and if it turns out as an error code, the same error code is propagated up to the caller.

Referring to Figure 1, the return status of call to `crypto_report_alg()` is checked at line 24 and the same status code is returned. To identify such more complicated error-handling cases, we look for a critical check. We define a critical check as a check against zero or NULL that has no fall-through and leads to a return instruction with an integer or NULL parameter. A zero/non-NULL return status indicates that an operation was successful, while a non-zero (usually a negative errno) or NULL return status is an indication of the operation failure. Therefore, for each critical check, based on the check predicate we can determine which side of the check is taken when an error happens. This way we can identify non-explicit error handling paths like the one shown in Figure 1 line 24.

Sequential Pattern Mining. The mining is applied to the sequential patterns of operations on error-handling paths of a given FOI. This way we can extract the common patterns of error handling for each FOI. A key part of such a finding is which release functions are used to de-allocate the allocated memory object. In terms of memory management, it means when an allocation succeeds (i.e., the memory is allocated), if an error happens later, how the allocated memory should be released. This can be represented by a high-level rule in form of <*call FOI*, *check*, *release*, *return*>. Such a sequential pattern of operations is important because the OS kernel has different mechanisms and functions to allocate and release memory dynamically. Finding associated release functions for any FOI can be addressed via looking for the sequential pattern of operations at the moment of error handling. Many specialized allocation functions allocate and craft specialized memory objects, which in turn require specialized de-allocation. Using the rule mining technique we identify such specialized release functions associated with a given FOI.

Frequent pattern mining algorithms expect the inputs in the form of a sequence of operations. Once we extract the error handling paths as described in IV-B, each path is transformed into a representation of sequential opcodes. To do so, we retrieve the LLVM opcode of each instruction on the path and form the opcode sequence. For the case of call instructions, we also retrieve the callee function name.

For each FOI, the opcode sequences are fed into the frequent pattern mining algorithm. The result will be a set of opcode patterns associated with error handling paths of the FOI. We take the most general pattern which by definition will be covering the maximum number of error handling paths and use it to identify the associated release function. Such a function is identified by cross-checking the result of mining against the set of operations as the last operation on the allocated pointer. Use-finding has a key role in here as we identify all operations that use a copy of the allocated pointer and consider only such operations in rule mining.

V. Ownership Reasoning

Another key challenge we have to address to detect kernel memory leaks is to decide where an allocated object is supposed to be released. In K-MELD, we propose a new ownership reasoning mechanism to infer the release locations. The ownership reasoning mechanism includes two components: enhanced escape analysis and consumer function detection.

A. Enhanced Escape Analysis

Understanding how a function manages the ownership of allocated memory is a key factor in designing a precise memory leak detection technique. The second condition of resource release bugs (as mentioned in §III) refers to that the owner function of the allocated memory fails to release the allocated memory upon finishing the uses. If the allocating function passes on the ownership of the memory object, then there will be no leaking problem if the allocating function finishes without releasing the allocated memory. The ownership can be transferred either via returning the allocated pointer or assigning the allocated pointer to a global object or a reference argument. We call such ownership transfer as escaping the pointer.

For example Figure 3 shows a case of an escaping allocated pointer. Here the allocated pointer escapes at line 33 via reference argument `bounce_buf_ret`. Looking at the call graph reveals that the function `hgcm_call_preprocess_linaddr()` is called repeatedly by `hgcm_call_preprocess()`. Eventually the caller of `hgcm_call_preprocess()` which is `vbg_hgcm_call()` takes care of releasing `bounce_bufs` either in case of success or failure. But the code in Figure 3 is still leaking memory at line 27, and it is because the pointers does not escape on this path. This shows the importance of path-sensitive escape analysis that we are employing to determine how the ownership of the allocation is changed. The leak can be resolved by moving

[5]If such a check is missing, it can be a missing-check bug. Since such bugs are out of the scope of this paper, if FOI is not checked, we assume it was successful.

```
1  /* File: drivers/virt/vboxguest/vboxguest_utils.c */
2  static int hgcm_call_preprocess_linaddr(
3      const struct vmmdev_hgcm_function_parameter *src_parm,
4      void **bounce_buf_ret, size_t *extra)
5  {
6      void *buf, *bounce_buf;
7      bool copy_in;
8      u32 len;
9      int ret;
10
11     buf = (void *)src_parm->u.pointer.u.linear_addr;
12     len = src_parm->u.pointer.size;
13     copy_in = src_parm->type != VMMDEV_HGCM_PARM_TYPE_LINADDR_OUT;
14
15     if (len > VBG_MAX_HGCM_USER_PARM)
16         return -E2BIG;
17
18     /* Memory is allocated here */
19     bounce_buf = kvmalloc(len, GFP_KERNEL);
20     /* Check for allocation success */
21     if (!bounce_buf)
22         return -ENOMEM;
23
24     if (copy_in) {
25         ret = copy_from_user(bounce_buf, (void __user *)buf, len);
26         if (ret)
27                 return -EFAULT;
28     } else {
29         memset(bounce_buf, 0, len);
30     }
31
32     /* Allocation pointer is assigned to a reference argument */
33     *bounce_buf_ret = bounce_buf;
34     hgcm_call_add_pagelist_size(bounce_buf, len, extra);
35     return 0;
36 }
```

Fig. 3: An example of escaping pointer: The allocation pointer `bounce_buf` is escaping via a pointer argument at line 33. But it is not enough to the prevent memory leak at line 27 (CVE-2019-19048).

the assignment at line 33 to line 23. When an allocation pointer is escaping on a path, then we assume there is another place of code responsible to manage the allocation, and the current path is not further explored for leak finding. This avoids the path-explosion problem and the tracking of complicated data flows, which in turn reduces false positives.

As mentioned in §IV-A source-finding is a backward data flow tracking which is employed to identify the sources of any destination that the allocation pointer is copied into. Source-finding determines the destination of copy at line 33 is a function argument. We tend to identify escaping pointer as those are the pointers that their ownership is passed to other functions, so it is safe if the current function do not release them. To identify if an allocated pointer is escaping we should find any variable that such pointer is copied into (via use-finding) and then determine the kind of the destination variable (via source-finding). If the destination variable is used in a return instruction, is a global variable, or is a function argument, then we can conclude that the allocated pointer escapes.

Path-sensitive escape analysis. Such an escape analysis should be path-sensitive as not all the execution paths may reach to the escaping point (as in Figure 3). Therefore, after identifying error handling paths (as described in IV-B) we perform escape analysis on each path. If the allocated pointer escapes on a specific error handling path, then we exclude such a path from the rest of our analysis, because the ownership is changed and the allocating function is not the sole entity having a handle on the allocated memory. It means if the allocating function terminates without releasing the allocated memory, there are still other live pointers to the allocation and no memory leak yet has happened. Instead we collect the escaping pointer information and go after the callers of the allocating functions one by one, and look for potential memory leak. This requires an inter-procedural analysis to track the operations performed on the escaping pointer in the context of the callers.

B. Consumer Function Detection

Consumer functions are another place that the ownership of an allocated memory object is changed. An allocating function may pass the memory object to its callees, so K-MELD analyzes those receiving callees to make sure ownership is not changed once returning from such callees. If the callee is a consumer, then there will be no memory leak. It means any execution path that is going through a consumer function, should be disregarded for memory leak detection.

Figure 4 shows an example of consumer function. The specialized allocator `alloc_skb` allocates a new network buffer `skb` at line 12. At line 16 `skb` is passed to the function `t4_mgmt_tx` which in turn passes `skb` to `ctrl_xmit` at line 26. As the implementation of `ctrl_xmit` shows the buffer is consumed on all execution paths. More specifically, the `skb` is released at lines 36 and 47, and is escaped via being added to a queue at line 42. This confirms the code at line 18 should not release `skb`. This example shows how K-MELD requires an inter-procedural analysis to track `skb` across the function calls.

Remember at this stage K-MELD knows associated deallocations of the FOI. So a pointer receiving callee becomes a consumer when on all of its execution paths it either releases or escapes the allocated pointer. This is realized by applying escape analysis to the paths of the callee, and also tracking the propagation of the allocated pointer in the callee to determine if it reaches any deallocation function.

Conditional consumers. While analyzing consumer function candidates, it may be the case that the candidate consumes (releases or causes to escape) the memory object conditionally. For example in Figure 5 the allocated `nskb` is passed to `skb_put_padto` which tries to pad the buffer. If it fails to do so, the memory object is released (as in line 38). But if it succeeds (as in lines 23 and 35) the ownership is not changed and the allocating function has to handle `nskb` appropriately. K-MELD handles such cases by first identifying the critical check at line 11. Then via the condition predicate it determines if the current path is associated with success or failure of `skb_put_padto` as described in IV-B. For example line 12 is associated with the failure, then when processing `skb_put_padto`, K-MELD only considers failure execution paths (only paths ending at line 39). This way K-MELD determines the code at line 12 is not responsible to release `nskb`, and there is no leak.

C. Detecting Bugs Using Mined Rules

Now that we presented the techniques, all the ingredients are ready to check the satisfiability of the two memory leak conditions in section III: the function owns the allocated memory object, but fails to release it. As described in IV-A, we collect the initial set of FOIs and then extract the error-handling paths as explained in IV-B. These paths are fed into the rule

```c
/* File:  drivers/net/ethernet/chelsio/cxgb4/srq.c */
int cxgb4_get_srq_entry(struct net_device *dev,
            int srq_idx, struct srq_entry *entryp)
{
    ...
    struct adapter *adap;
    struct sk_buff *skb;
    ...
    adap = netdev2adap(dev);
    ...
    /* ALLOCATION */
    skb = alloc_skb(sizeof(*req), GFP_KERNEL);
    if (!skb)
            return -ENOMEM;
    ...
    t4_mgmt_tx(adap, skb);  /* CONSUMER */
    ...
    return rc;
}

/* File: drivers/net/ethernet/chelsio/cxgb4/sge.c */
int t4_mgmt_tx(struct adapter *adap, struct sk_buff *skb)
{
    int ret;
    ...
    ret = ctrl_xmit(&adap->sge.ctrlq[0], skb); /* CONSUMER */
    ...
    return ret;
}

static int ctrl_xmit(struct sge_ctrl_txq *q, struct sk_buff *skb)
{
    ...
    if (unlikely(!is_imm(skb))) {
            WARN_ON(1);
            dev_kfree_skb(skb);     /* RELEASE */
            return NET_XMIT_DROP;
    }
    ...
    if (unlikely(q->full)) {
    ...
        __skb_queue_tail(&q->sendq, skb);    /* ESCAPE */
        spin_unlock(&q->sendq.lock);
        return NET_XMIT_CN;
    }
    ...
    kfree_skb(skb);     /* RELEASE */
    return NET_XMIT_SUCCESS;
}
```

Fig. 4: Consumer function example: `t4_mgmt_tx` consumes the buffer `skb` unconditionaly.

```c
/* File: net/dsa/tag_ksz.c */
static struct sk_buff *ksz_common_xmit(struct sk_buff *skb,
    struct net_device *dev, int len)
{
    nskb = alloc_skb(NET_IP_ALIGN + skb->len +
            padlen + len, GFP_ATOMIC);
    if (!nskb)
        return NULL;
    ...
    /* CONDITIONAL-CONSUMER */
    if (skb_put_padto(nskb, nskb->len + padlen))
        return NULL;
    ...
    return nskb;
}
/* File: net/core/skbuff.c */
int __skb_pad(struct sk_buff *skb, int pad, bool free_on_error)
{
    int err;
    int ntail;
    if (!skb_cloned(skb) && skb_tailroom(skb) >= pad) {
        memset(skb->data+skb->len, 0, pad);
        return 0;
    }
    ...
    if (likely(skb_cloned(skb) || ntail > 0)) {
        err = pskb_expand_head(skb, 0, ntail, GFP_ATOMIC);
        if (unlikely(err))
            goto free_skb;
    }
    err = skb_linearize(skb);
    if (unlikely(err))
        goto free_skb;
    memset(skb->data + skb->len, 0, pad);
    return 0;
free_skb:
    ...
    kfree_skb(skb);
    return err;
}
```

Fig. 5: Conditional Consumer function example: `skb_put_padto` consumes the buffer `skb` on failure.

mining to detect associated deallocation functions as described in IV-B. We prune the escaping paths and those going through consumer functions. The remaining paths are used in a pattern-matching step to detect the paths that miss the release function. Such paths comprise K-MELD's memory leak reports.

VI. Implementation

We have implemented K-MELD as multiple LLVM passes, integrated with Python code to run the passes and rule mining, and then perform the pattern matching.

A. Potential Allocation Functions

As our ultimate goal is detecting memory leak bugs, we need to populate an initial set of potential memory allocation functions. Such functions will become the initial set of FOIs for the rest of our analysis. In order to scale to the large number of specialized allocation functions in the kernel, we need this phase to be automated.

Using a preprocessing LLVM pass over all the kernel, we identify any pointer-returning function. Then looking into the definition of that function, we make sure the returning pointer is not derived from any pointer argument, because allocation function is supposed to return a previously non-existent memory object. To do so, we use source-finding described in §IV-A to make sure the returning pointer is a not coming from a pointer argument. After that, we look at the call-sites of the candidate allocator and using use-finding analysis determine if the returned pointer is being null-checked and initialized or not. This can identify a caller function of a primitive allocator (like `kmalloc`) as a new allocator. Such design choice reduces the tracking of complicated and lengthy data flows.

For some reason not all memory allocations are null-checked. As an example if flag `GFP_NOFAIL` is passed to `kmalloc`, the allocation cannot fail [21]. Additionally, some primitive allocators like `kzalloc` or `kcalloc` zero-initialize the memory object. Therefore, at this stage it is enough a candidate allocator to be null-checked or initialized at least in 40% of cases (such threshold was selected based on empirical observations in common allocators). This initial list of functions later will be pruned when we apply release detection described in §IV-B. Any function that fails to yield at least one release function in rule mining will be discarded from the analysis results.

B. Path Exploration

To identify error handling paths, as described in §IV-B, we statically explore the CFG and determine if a path is error

handling or not. To avoid path explosion problem, in addition to loop unrolling, we set a hard limit to the number of paths we explore per function. We also set a limit on the path depth to handle deep paths like mutually recursive calls. In the current implementation, we set the both limits to 1000. Such a limit was selected empirically to balance the pass running time and the leak detection rate.

When exploring the paths, we keep track of the uses of the allocated pointer. For each explored path, we maintain a list of opcodes that work on FOI result; specifically, function calls that take the FOI result as an argument. Such call instructions, are used for consumer function detection. For each explored path, we also record the last call instruction taking the FOI result as an argument in a set called *last_foi_use*.

Pruning Infeasible Paths. We employ a simple but effective infeasible path pruning. K-MELD is interested in the execution paths where the allocation was successful. After detecting the initial null-check on the FOI result, any subsequent null-check on the same pointer will result in infeasible path if the condition is taken. Therefore K-MELD prunes a path if any branch, either in the allocating function, or in a consumer function, implies that the allocated pointer is null.

C. Context-aware Rule Mining

For the purpose of sequential pattern mining, we employed the CloFAST [12] algorithm. CloFAST is a fast algorithm for discovering closed sequential patterns from a set of sequences. A sequential pattern is a sub-sequence that appears in many input sequences, and intuitively a pattern is closed if it cannot be extended. More precisely the *support* of a sequential pattern is the ratio of the number of times the sequential pattern appears divided by the total number of input sequences. The input to the CloFAST is a sequence database and a user-defined minimum support *minsup*. The sequence database is a set of sequences where each sequence is a list of opcodes associated with error handling paths as described in §IV-B. The minimum support is a frequency threshold in terms of percentage which is used to recognize a frequent sequential pattern. A frequent sequential pattern is a pattern with a support level of no less than *minsup*. A closed sequential pattern is a frequent sequential pattern that is not a subset of any other pattern with the same support level. Discovering closed sequential patterns is more efficient than finding all sequential patterns while missing no information about the patterns [12]. Informally, as all sub-patterns of a frequent pattern are also frequent, therefore mining closed sequential patterns avoids generating unnecessary patterns and as a result yields savings of space and computational costs. In our implementation we empirically set the *minsup* to 0.6, as further described in §VIII. In our implementation we noticed a limitation of the mining algorithm with respect to the number of input sequences. Meaning that the algorithm was not able to process all execution paths even on a machine with a large amount of memory. We avoided this limitation by first applying escape analysis to prune uninteresting paths, and then feeding in the sequence of operations on each path as input to the mining algorithm.

Release Function Identification. Once the rule mining is finished, we will have a set of sequential patterns for the specific FOI. We are interested in the patterns that follow the correct behavior of memory release; meaning <*Call FOI*, *check*, *release*, *return*>. Such a pattern is used to identify the associated release function to the FOI. We cross-check the mined pattern against *last_foi_use* set populated at the path exploration phase. This confirms the candidate release is the last function working on FOI.

D. Ownership Reasoning

Path-sensitive Escape Analysis. We employ escape analysis to track the ownership of the allocated memory. K-MELD's use of escape analysis is inter-procedural in that if a pointer escapes from the current function f, it is never reported as a leak from f. However all the callers of f are tracked to check for leaking the pointer.

Consumer Function Path Profiling. For the purpose of consumer function detection, K-MELD analyzes any called function that takes the allocated pointer as an argument. It labels the callee paths as success or failure based on the return code. Then each caller path is associated with either of these path collections. If the caller path takes the success side of the consumer candidate, then callee's success paths are considered for releasing or escaping the allocated pointer, and vice versa.

Inter-procedural Data Flow. At multiple stages, K-MELD employs inter-procedural data flow analysis to track the allocated pointer. For each LLVM call instruction taking allocated pointer as an argument, we determine the argument index in the call instruction. Then in the definition of the callee we start tracking the argument at that index. This way K-MELD monitors the propagation of pointers across functions.

E. Pattern Matching and Error Finding

When we have identified the release functions associated with the specific FOI, K-MELD then goes through the error handling paths extracted for the FOI and applies pattern matching. The pattern are of the form <*call FOI*, *check*, *call release*, *return*> where any call to the identified release functions will match the *call release* item.

Any path failing to match in the previous step will be further investigated for potential consumer functions. K-MELD looks at the function calls on the allocated pointer and checks if the associated paths in the callee are consuming the pointer or not. If consuming, such a path is disregarded as a potential bug.

This way any error handling path deviating from the common approach of memory releasing will be found. Then such bug candidate paths are manually audited to confirm it is a memory leak bug. Once confirmed, we then prepare a patch to fix the bug and submit the patch to the code contributors.

VII. EVALUATION

The effectiveness and scalability of K-MELD is tested on the Linux kernel, version 5.2.13-stable. We compiled the kernel code into LLVM bitcode using `allyesconfig`, and got 18074 LLVM IR bitcode files. The experiments were carried out on an Intel Xeon CPU server with 48-cores and 256GB RAM, and runs Ubuntu-18.04 OS with LLVM v8.0 installed.

A. Scalability

LLVM bitcode generation for the whole kernel took 5 hours. Allocation function collection took less than 2 hours. Note that these two steps are a one-time process and can be reused. Sequential pattern mining and pattern matching for identifying deallocators took slightly more than one hour only. The detection for each FOI takes from seconds to 4 hours (with 3 minutes average). The most time-consuming case (4 hours) is for kmalloc with the largest number of callsites. Note that based on K-MeLD design, each FOI can be analyzed independently, making it possible to benefit from parallelization.

B. Set of Allocations and Associated Deallocations

Using the methods described in §IV-A we populated an initial set of 4621 candidate allocator functions. Furthermore, once the rule mining finished, those failing to produce at least one associated deallocation, are pruned. This left us with 807 allocation functions[6]

These functions are considered as FOI for our evaluation. These FOIs have a wide range of frequency (some with many call sites, some with just a few), and so demonstrate K-MELD's effectiveness for both general-purpose and specialized allocation functions. Withing this set of 807 functions, there are 4 generic allocation functions (like `kmalloc`) with over one thousand callsites, along with more specialized allocation functions with callsites down to 2 (like `charlcd_alloc`). K-MELD was able to find the associated release functions for even the most specialized allocation functions (with as few as 2 callsites). Looking at the FOIs and associated releases, we identified only 15 false positive cases (1.8%). Such cases are result of incorrectly paired functions in the mining algorithm (IV-B). In VIII we discuss the effect of minimum support parameter in the mining algorithm on the rate of false positive and false negative. Moreover, there were allocators like `fscache_alloc_retrieval` that were not selected as FOI, but were covered by K-MELD. These functions are either escaping the allocation through pointer arguments, or did not pair with a specific deallocator. But essentially they are using other primitive allocators like kzalloc and are processed as escaping functions. Looking into the internal of 807 FOIs, we found a small number of primitive allocators (21 FOIs, comprising 2% of all selected FOIs) that are used to perform the actual allocations.

This selection of allocation and associated deallocation functions are necessary for effectively detecting memory leak bugs. To the best of our knowledge, none of the previous detection techniques used such a rich set of allocation-deallocation functions.

C. Found Bugs

In total, K-MELD generated 458 leak warnings. After manual audit we confirmed 218 new memory leak bugs. This means K-MELD is bearing 52% false positive. We prepared patches and submitted for those bugs. To date, 106 patches have been confirmed by the Linux code maintainers and the rest are under review. Table V (in §A) lists all the found bugs, giving the source code file, function name, the FOI name, and whether the bug is confirmed or not yet. We reported the number of bugs based on the patches submitted, meaning that a single patch sometimes covers multiple memory leaks in the same function. We also requested CVE IDs for the leak bugs found, which to date we received 41 CVEs as listed in Table VI (in §A).

The bugs we found are spread among the most common allocation functions like `kmalloc` (with 9244 call-sites) and `kzalloc` (with 9926 call-sites); to the most specialized allocation functions like `sync_file_alloc`, `edac_mc_alloc()` and `nlmsg_new()` with 2, 17 and 333 call-sites, respectively.

From the perspective of specialized modules, among 218 detected bugs, 115 were related to the FOIs with 400 call-sites or less. These results demonstrate the utility of K-MELD to detect memory leak bugs even in specialized kernel modules.

D. Exploitability Analysis

Entry functions	Attacker	Count
System calls	Userspace	137
Ioctl handlers	Userspace	173
IRQ handlers	Hardware	173
Reachable from any entry		182 (83.9%)

TABLE I: The number of reachable bugs for different types of entry functions. IRQ = Interrupt request, Ioctl = I/O control.

As a first step toward automatically assessing the security impact of the bugs found by K-MELD, we have tested the control-flow reachability of each bug location from common attacker-controllable kernel entry points. Precisely determining the exploitability or even just reachability of bugs found statically is a hard problem, especially in a kernel. Dynamic approaches such as fuzzing or whole-kernel symbolic execution can confirm exploitability if they find a triggering input, but the search space is so large that these approaches can run indefinitely without showing the absence of an exploit, and code that interacts with hardware is challenging to execute in simulation. Detailed checking of path feasibility such as with static symbolic execution and constraint solving is also expensive and suffers from path explosion. Therefore, in this project, we weaken these conditions and try to find the reachable call stacks between entry points function to vulnerable functions.

To this end, we first analyzed the call relationship for every function in the Linux kernel. To improve the accuracy rate, we identify the indirect-calls and refine the indirect-call targets by using struct type matching [25, 26, 50]. Then, we can build a complete call-graph for the whole kernel. Based on this call-graph, we try to find the shortest path between every attacker-controllable functions to the functions that include memory leak bugs.

As the starting points of these paths, we identify three kinds of attacker-controllable functions in the Linux kernel: system calls, I/O control handler functions, and interrupt request handlers functions. Previous works such as [42] and [9] have

[6]The full list is available at: https://github.com/Navidem/k-meld/blob/main/results/FOIs.txt

used these functions as entry points to guide the kernel fuzzer, which show the effectiveness of these entry points. Finally, we find 83.9% of bugs found by K-MELD can be potentially reached from attacker-controllable points. Though this analysis is coarse-grained, it confirms our intuition that a majority of memory leaks in a kernel can likely be triggered by a local user or a misbehvaing hardware device. By contrast an example of a leak that cannot be triggered in this way is one that occurs during kernel boot. Table I shows the number of vulnerable functions that can be reached for different types of entry points.

E. False Positive Analysis

K-MELD has a false-positive rate of 52%, which we believe is acceptable for a static analysis tool applied to an OS kernel. We revisited the false warnings issued by K-MELD to get an understanding on the sources of such false positives.

Infeasible paths are one of the main sources of false warnings. As we use static path exploration and do not check for path feasibility except a few cases, K-MELD may assume some infeasible paths as potential leaks. Incorporating some more expensive analysis like under-constrained symbolic execution [38] can help avoid these cases. Customized device-managed allocations caused false warnings, too. These are driver-specific allocations that autonomously release all the allocated resources at the time of device detachment. Uncommon customized release functions which K-MELD fails to identify in the rule mining step are another source of false positives. These functions do not pass the minimum threshold of mining and are basically not a wrapper for the common release functions; therefore, it leads to false positive. For example, K-MELD correctly identified `kmem_cache_free()` as the release for `kmem_cache_alloc()`, but in 9 cases the allocated cache is released via `abort_creds()`.

F. False Negative Analysis

K-MELD misses memory leak bugs if the allocation function is not in the set of FOIs. Additionally, even though we incorporate inter-procedural escape and consumer function analysis, these analyses are not complete and there may be cases where pointer propagation is lost due to aliasing or complicated data structure assignments. In order to perform a false negative analysis we decided to use 17 previously detected memory leak bugs which were assigned CVEs as ground truth. These bugs were spread among multiple versions of the Linux kernel over the span of 10 years. Instead of compiling multiple versions of the kernel, we reproduced each bug by undoing the proposed patch.

Table II shows the results of this analysis. Out of 17 cases, it turned out one was false positive and the initially merged patch was reverted later[7]. K-MELD correctly identifies 9 of those memory leaks, and correctly identifies the reverted one as escaped pointer. Two of those bugs are out of scope, as the source of the bug is not missing a release function, but API confusion[8]. Four of the bugs were not reproducible due to the code structure changes between kernel versions. Finally, K-MELD missed one bug due to the complicated pointer propagation. More specifically, the allocation function causes the allocated pointer to escape by adding it to a list in a field of reference argument. K-MELD loses the track of the pointer when it gets to the caller.

CVE #	Reproducible	K-MELD Success/Failure
CVE-2019-8980	✓	Success
CVE-2019-9857	✓	Success
CVE-2019-16995	✓	Failure
CVE-2019-16994	✓	Success
CVE-2019-15916	✓	Success
CVE-2019-15807	✓	Success
CVE-2019-12379	✓	Success (correctly rejected)
CVE-2018-8087	✓	Success
CVE-2018-7757	✓	Success
CVE-2018-6554	X	—
CVE-2016-9685	✓	Success
CVE-2016-5400	✓	Success
CVE-2015-1339	X	—
CVE-2015-1333	X	—
CVE-2014-8369	✓	out-of-scope
CVE-2014-3601	✓	out-of-scope
CVE-2010-4250	X	—

TABLE II: False Negative Analysis results based on previous memory Leak CVEs

G. Comparison with Hector

Hector [41] is the closest detection tool to K-MELD. Hector detects resource release bugs in a function if some error-handling paths correctly release the allocated memory while some other do not. Thus Hector fails to detect bugs in a function where no path is correctly releasing the allocated memory. The presence of correct path is a hint to hector to decide the current function in the owner of the allocated pointer.

Unfortunately, Hector's source code is not available, so we cannot run it on the same kernel used in our main evaluation. Instead we compile the old kernel used by Hector to LLVM bitcode and run K-MELD on it. Hector reports bugs in version 2.6.34. Despite the challenges of compiling an old kernel using LLVM, we managed do so partially and produce 6861 bitcode files. On the other hand, we were able to obtain a raw table of outputs from Hector authors containing 4979 entries. By cross-referencing the table entries and the patches submitted by authors to the Linux kernel[9], we were able to retrieve 29 memory leak bugs identified by Hector. 16 of these bugs were in the sources that produced bitcode. K-MELD successfully detected all of these 16 bugs. Table IV in §A lists these bugs. This experiment also demonstrated how good K-MELD is in terms of detecting known bugs.

[7]CVE-2019-12379: commit 84ecc2f was reverted by 15b3cd8.

[8]The leak happens because `kvm_pin_pages()` expects size, but `kvm_unpin_pages()` expects the number of pages as argument.

[9]Linux Kernel Mailing List: https://lkml.org/

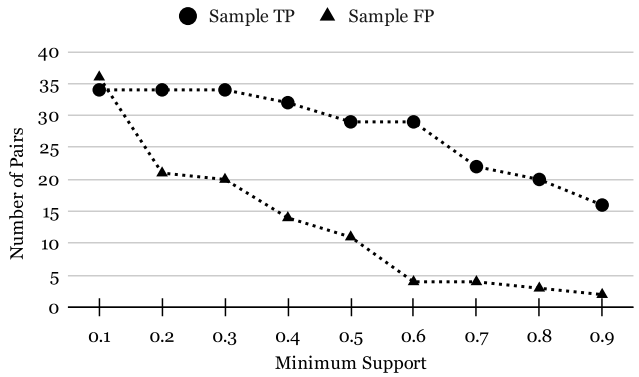

Fig. 6: The effect of different *minsup* on deallocation detection

H. Effectiveness of Escape and Consumer Function Analysis

K-MELD incorporates inter-procedural path-sensitive escape and consumer function analysis to reason on the ownership of the allocated pointer. The escape analysis affects whether an execution path should be further considered for potential leak finding or not. Consumer function analysis determines if the pointer is consumed once passed to a function. As a result, both of these analyses affect the number of reports K-MELD may generate for each FOI.

To evaluate the effectiveness of such analyses, we monitor the total number of reports generated by K-MELD with each of these analyses enabled and disabled. Our experiments demonstrated disabling escape analysis increases the number of reports to 1292, while disabling consumer function analysis produces 1386 reports. This results confirms the effectiveness of our path-sensitive escape and consumer function analyses to reduce the false positive rate. We manually audited 20 of these suppressed reports which were selected randomly and it did not reveal any missed bug.

VIII. DISCUSSION

In this section we describe some additional design choices in our implementation and evaluation.

Choosing Minimum Support. The rule mining algorithm uses the parameter *minsup* to determine if a sequence is frequent or not. This threshold affects the quality of identified release functions for a given FOI. A too-high `minsup` may lead to missing release functions of a given FOI, while a too-low `minsup` may lead to incorrectly identified releases.

To perform pattern matching we need at least one release function; if we miss a release function it contributes to false positive in leak warnings by K-MELD. If we incorrectly identify a release function it causes false negative. We prefer to choose a *minsup* that does not yield incorrect release functions to avoid false negative with the cost of some false positive.

To demonstrate the effect of different support levels on the release function detection we randomly selected 100 potential allocation functions and recorded the number of correct and incorrect pairs. Figure 6 shows the result. As it shows, *minsup=0.6* provides a good trade-off between release detection true positives and false positives.

`devm_kmalloc()`	`devm_kmalloc_array()`
`devm_kcalloc()`	`devm_kzalloc()`
`devm_kmemdup()`	`devm_kstrdup()`
`devm_kasprintf()`	`devm_kvasprintf()`
`devm_get_free_pages()`	`devm_free_pages()`

TABLE III: List of Managed Resource Allocations

Managed Resource Allocation. A feature known as managed device resources was introduced in kernel 2.6.21 [18]. In this model, each device driver maintains a list of allocated resources which will be released when the device is detached. It means a manged resource allocator adds the memory to the device list upon successful allocation. As documentation states, there is no explicit release needed for managed resource allocation. Such functions were not selected as potential FOIs because in the mining step they did not yield at least one release function. Table III lists such functions.

Reference Counting. The Linux kernel uses the technique of reference counting [32] for some objects. Using reference counting to manage the lifetime of an object is to have a counter which is incremented whenever a reference is taken, and decremented whenever a reference is released. When this counter reaches zero, then any resource (like the memory) used by the object can be freed. In our initial results we noticed a higher number of false positives for some general purpose allocators like `kmalloc` and `kzalloc`. After investigations it turned out that reference counting releases are a major factor of such false positives. We decided to perform another round of mining, but this time just using the set of paths that were not matched via `kfree` as release. Results were satisfying where we identified three reference counting releases: `kref_put`, `kobject_put`, and `put_device`. Therefore, we added these three functions to the set of release functions, whenever `kfree` was identified as release.

Escaped Pointers. The path-sensitive escape analysis helps with determining the locations of the code where the ownership of the allocated memory is transferred. One strategy could be tracking an escaping pointer in the context of the caller. If the pointer is escaped to the caller via return or a reference argument, then the caller is analyzed from the instruction following the call. Tracking the pointer propagation determines if it is being released or not. We tried this in K-MELD and for each escaping pointer, tracked any potential caller. It produces 131 reports, which auditing half of them did not reveal any bug. Therefore we decided to de-rank those reports. Our understanding is that complicated data flow used for escaping via reference arguments and more complex feasible paths on multiple levels of call-graph hinder the precision.

K-MELD does not track an allocation when it is added to a global list or queue; therefore it may miss any subsequent memory leaks. Our analysis shows that such cases of escape are mainly realized through variants of `list_add` and `queue_tail` functions.

Ranking Results. When exploring the execution paths as described in IV-B we do not check for complicated infeasible paths. As a result, if a release function is under a check there will be a path that will not go through the release no matter what is the check condition. One solution for this could be using

some heavyweight analysis like under-constrained symbolic execution [38]. To avoid such expense, we took a different approach. An infeasible path if not pruned, introduces false positive, so we de-prioritize such path. More specifically, we determine whether the release candidate function (last function call using the allocated memory) is under a check, and also whether the check is not critical (as defined in IV-B), then any path going through such check is de-prioritized when generating leak warnings.

Other Classes of Bugs. In the process of detecting kernel memory leak bugs, we also found other types of security bugs and submitted patches to the Linux kernel. As K-MELD already searches to determine if an FOI result is null-checked or not (IV-B), we found a handful of missing null-check bugs. The call to an allocation function may fail and therefore a null pointer may be returned. Therefore, any missing check bug on an allocation operation is a potential null pointer dereference. Use-after-free is another class of bugs, where the pointer to an already released memory is used to read or write. We explicitly looked for simple use-after-free bugs by looking for any subsequent use of the pointer to the release function, and interestingly found one such bug. We received `CVE-2019-18814`[10] for the bug where a pointer to an already released memory was dereferenced in an error-handling path.

The general approach of rule mining employed in this paper can be applied to any resource release bugs. Another common case is locks: when a function acquires a lock, any potential the error-handling path must release the lock. We leave such extensions for future work. K-MELD is also extendable to other OS kernels like FreeBSD. It requires compiling the source code into LLVM IR and then running K-MELD on it to collect leak reports, and then auditing the reports.

IX. Related Work

Static Memory Leak Detection. Hector [41] has a similar motivating intuition to K-MELD, and has also been applied to the Linux kernel, but it is limited to finding inconsistencies within a single function. Hector warns on situations where a resource is released on some paths through a function and not on others. It also checks whether a resource is returned from a function, but not if it escapes via a pointer.

Leak Checker [48] tries to detect memory leaks via static path sensitive pointer analysis. It reduces the memory leak detection problem to a Boolean satisfiability problem, and then uses a SAT-solver to identify potential bugs. Other static memory leak detection techniques have been proposed [7, 13, 20, 34, 39, 43], which to the best of our knowledge were not scalable to OS kernel.

Deviation-based Analysis. K-MELD's use of mining is similar to Engler et al.'s [10] proposal to infer bugs by looking for deviations from commonly observed behavior. More specifically, they looked for NULL-pointer inconsistency through the OS kernel and were able to detect multiple bugs.

Pattern mining has been employed in previous research for the purpose of program analysis. In [24] the authors used mining on code revision history to find paired functions. Found pairs are then checked at the run-time to find violations. The approach is not flow sensitive, and relies on user input to enhance pattern matching. The authors of [16, 46] applied sequential pattern mining for specification mining by focusing on Java exception-handling code. PR-Miner [23] uses frequent item-set mining to infer programming rules without using user-defined templates. It does not consider the sequence of operations. MUVI [29] uses a similar approach to detect concurrency bugs. CHRONICLER [37] integrates sequential pattern mining with a path-sensitive data flow analysis to identify precedence rules for a function call. To the best of our knowledge none of these works were applicable to a large system like OS kernel.

Error-handling based Detection. Using error-handling code to detect bugs was employed previously is many works. LRSan [44] and Crix [26] find classes of missing-check bugs in the Linux kernel via employing error-handling code to identify critical variables. Other techniques analyzing error-handling code within the Linux kernel include [2, 17, 40] where the error propagation is evaluated to detect potential bugs. These techniques rely on explicit `errno` returning to identify error-handling code. Based on our study, non-explicit error-handling cases are common, and missing those, causes false negative. Such error-handling cases are covered via critical check identification in K-MELD as described in IV-B.

Kernel Vulnerability Analysis. Because of the inherent complexity of the OS kernel, analyzing the whole kernel is challenging. Therefore, first compiling the whole kernel into LLVM IR became a practical approach. This facilitates the analysis by employing LLVM passes. K-Miner [14] performs inter-procedural analysis by extracting execution paths starting from system-calls to detect memory corruption vulnerabilities. Without the ownership reasoning mechanism and the identification of specialized allocators/deallocators, K-Miner has to analyze complicated data flows globally, which is very hard. As a result it only focuses on the paths starting from syscalls. Dr. Checker [31] finds vulnerabilities in the Linux kernel drivers via static data flow analysis. KINT [45] and UniSan [27] employ taint analysis to find integer overflow and information leakage, respectively. SLAKE [6] provides an automated method to exploit vulnerabilities in the Linux kernel by extending LLVM for its static analysis, in tandem with the fuzzer Syzkaller [15]. DCUAF [1] proposes a static analysis approach to detect use-after-free bugs in the Linux device drivers.

X. Conclusion

Memory leak bugs are a serious security vulnerability in critical systems like OS kernels. In this paper we presented K-MELD, an effective and scalable static memory leak detection for kernels that may consist of many modules. K-MELD can detect memory leak bugs not only on general allocation functions, but also on specialized allocation functions with only a handful of call sites. K-MELD first identifies allocation functions via a structure- and usage-aware approach, then associated deallocations are determined by using a sequential pattern mining technique. To detect memory leak bugs, K-MELD reasons on the ownership of the allocated memory object to determine the location of expected dealocation call. Such reasoning is realized via inter-procedural and path-sensitive escape and consumer-function analysis. Our application of K-MELD to the Linux kernel found 218 new memory leak

[10] security/apparmor/audit.c:191

bugs among which the maintainers so far confirmed 106. We also received 41 new CVEs for the detected memory leak bugs.

ACKNOWLEDGMENT

We would like to thank our shepherd, Mathias Payer, and the anonymous reviewers for their insightful suggestions and comments. We also thank Julia Lawall for providing us with Hector results, and Linux maintainers for providing prompt feedback on patching the Linux kernel. This research was supported in part by the NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in Linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 255–268.

[2] J.-J. Bai, Y.-P. Wang, J. Yin, and S.-M. Hu, "Testing error handling code in device drivers using characteristic fault injection," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 635–647.

[3] J. Bonwick *et al.*, "The slab allocator: An object-caching kernel memory allocator." in *USENIX Summer*, vol. 16. Boston, MA, USA, 1994.

[4] B. M. Cantrill, "Method and apparatus for post-mortem kernel memory leak detection," Feb. 18 2003, US Patent 6,523,141.

[5] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 2011, p. 5.

[6] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019, pp. 1707–1722.

[7] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 480–491.

[8] J. K. Chittigala, "System and method for finding kernel memory leaks," Jul. 16 2013, US Patent 8,489,842.

[9] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2123–2138.

[10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502041

[11] C. Erickson, "Memory leak detection in embedded systems," *Linux Journal*, vol. 2002, no. 101, p. 9, 2002.

[12] F. Fumarola, P. F. Lanotte, M. Ceci, and D. Malerba, "CloFAST: closed sequential pattern mining using sparse and vertical id-lists," *Knowledge and Information Systems*, vol. 48, no. 2, pp. 429–463, 2016.

[13] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for C programs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 459–470.

[14] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-Miner: Uncovering memory corruption in Linux," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[15] Google, "syzkaller - kernel fuzzer," 2019, https://github.com/google/syzkaller.

[16] C. Goues and W. Weimer, "Specification mining with few false positives," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, ser. TACAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 292–306. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00768-2_26

[17] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "EIO: Error handling is occasionally correct." in *FAST*, vol. 8, 2008, pp. 1–16.

[18] T. Heo, "Devres - managed device resource," 2019, https://www.kernel.org/doc/Documentation/driver-model/devres.txt.

[19] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *USENIX Security Symposium*, 2009, pp. 383–398.

[20] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *Proceedings of the 7th international symposium on Memory management*. ACM, 2008, pp. 131–140.

[21] Kernel.org, "kmalloc," 2019, https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html.

[22] ——, "Slab allocator," 2019, https://www.kernel.org/doc/gorman/html/understand/understand011.html.

[23] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 306–315.

[24] B. Livshits and T. Zimmermann, "Locating matching method calls by mining revision history data," in *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*. ACM, 2005, pp. 296–305.

[25] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1867–1881.

[26] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in *28th USENIX Security Symposium*, 2019, pp. 1769–1786.

[27] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 920–932.

[28] K. Lu, M. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack

spraying," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.
[29] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 103–116.
[30] lwn.net, "Injecting faults into the kernel," 2019, https://lwn.net/Articles/209257/.
[31] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for Linux kernel drivers," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.
[32] P. E. McKenney, "Overview of Linux-kernel reference counting," 2007, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2167.pdf.
[33] MITRE, "CWE-400: Uncontrolled resource consumption," 2019, http://cwe.mitre.org/data/definitions/400.html.
[34] M. Orlovich and R. Rugina, "Memory leak analysis by contradiction," in *International Static Analysis Symposium*. Springer, 2006, pp. 405–424.
[35] OWASP, "Memory leak," 2019, https://www.owasp.org/index.php/Memory_leak.
[36] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *EuroSys*, 2008.
[37] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 240–250.
[38] D. A. Ramos and D. Engler, "Under-Constrained Symbolic Execution: Correctness Checking for Real Code," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.
[39] D. Rayside and L. Mendel, "Object ownership profiling: a technique for finding and fixing memory leaks," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 194–203.
[40] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *ACM Sigplan Notices*, vol. 44. ACM, 2009, pp. 270–280.
[41] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.
[42] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary." in *NDSS*, 2019.
[43] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 254–264.
[44] W. Wang, K. Lu, and P. Yew, "Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.
[45] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving Integer Security for Systems with KINT," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.
[46] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 461–476. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31980-1_30
[47] J. Xiao, H. Huang, and H. Wang, "Kernel data attack is a realistic security threat," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 135–154.
[48] Y. Xie and A. Aiken, "Context-and path-sensitive memory leak detection," in *Proceedings of ESEC-FSE*. ACM, 2005, pp. 115–125.
[49] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 414–425.
[50] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "PeX: A permission check analysis framework for Linux kernel," in *28th USENIX Security Symposium*, 2019, pp. 1205–1220.

APPENDIX

source:lineno
net/key/af_key.c:3471
net/wireless/wext-core.c:763
fs/btrfs/volumes.c:3217
fs/autofs4/dev-ioctl.c:645
drivers/net/wan/farsync.c:2482
drivers/video/intelfb/intelfbdrv.c:524
drivers/block/cpqarray.c:1216
drivers/scsi/lpfc/lpfc_sli.c:9581
drivers/scsi/lpfc/lpfc_sli.c:9685
drivers/scsi/lpfc/lpfc_sli.c:9878
drivers/scsi/aic94xx/aic94xx_hwi.c:222
drivers/scsi/scsi_debug.c:3238
drivers/edac/i3200_edac.c:331
drivers/usb/misc/usbtest.c:1924
drivers/usb/misc/usbtest.c:1934
drivers/staging/comedi/comedi_fops.c:241

TABLE IV: List of bugs reproduced from Hector [41]

Allocating Function	FOI	Missing Release	Allocating Function	FOI	Missing Release
xcan_rx	alloc_can_skb	consume_skb	bcm_sysport_probe	alloc_etherdev_mqs	free_netdev
__test_case_3	alloc_extent_map	free_extent_map	test_case_2	alloc_extent_map	free_extent_map
__test_case_4	alloc_extent_map	free_extent_map	test_case_1	alloc_extent_map	free_extent_map
s3fwrn5_i2c_read	alloc_skb	kfree_skb	cxgbit_setup_conn_pgidx	alloc_skb	kfree_skb
mt7601u_burst_write_regs	alloc_skb	kfree_skb	mt76x02u_mcu_wr_rp	alloc_skb	kfree_skb
cxgbit_setup_conn_digest	alloc_skb	kfree_skb	mt7601u_write_reg_pairs	alloc_skb	kfree_skb
__sys_accept4	alloc_skb	kfree_skb	htc_config_pipe_credits	alloc_skb	kfree_skb
create_cq	alloc_skb	kfree_skb	htc_connect_service	alloc_skb	kfree_skb
ath9k_wmi_cmd	alloc_skb	kfree_skb	htc_setup_complete	alloc_skb	kfree_skb
cgx_probe	alloc_workqueue	destroy_workqueue	mmc_blk_probe	alloc_workqueue	destroy_workqueue
mlxsw_emad_init	alloc_workqueue	destroy_workqueue	bnxt_re_create_srq	atomic_inc	atomic_dec
bsg_sg_io	blk_get_request	blk_put_request	generate_filter	bpf_prog_alloc	__bpf_prog_free
cfctrl_linkup_request	cfpkt_create	cfpkt_destroy	cifs_setlk	cifs_lock_init	kfree
rsi_send_beacon	dev_alloc_skb	kfree_skb	do_lookup_dcookie	find_dcookie	free_dcookie
fuse_create_open	fuse_iget	d_splice_alias	hso_create_net_device	hso_create_device	kfree
ib_cm_insert_listen	ib_create_cm_id	kfree	davinci_timer_register	ioremap	iounmap
fsl_ifc_nand_probe	kasprintf	kfree	sja1105_static_config_upload	kcalloc	kfree
ti_dra7_xbar_probe	kcalloc	kfree	alloc_sgtable	kcalloc	kfree
adis_update_scan_mode	kcalloc	kfree	adis_update_scan_mode_burst	kcalloc	kfree
rtl_usb_probe	kcalloc	kfree	usbduxfast_auto_attach	kmalloc	kfree
ca8210_probe	kmalloc	kfree	kmemleak_test_init	kmalloc	kfree
__ubifs_node_verify_hmac	kmalloc	kfree	nfs4_try_migration	kmalloc	kfree
ccp_run_sha_cmd	kmalloc	kfree	af9005_identify_state	kmalloc	kfree
cx24117_load_firmware	kmalloc	kfree	read_znode	kmalloc	kfree
predicate_parse	kmalloc_array	kfree	getname_flags	kmem_cache_alloc	kmem_cache_free
dwc3_qcom_probe	kmemdup	kfree	unittest_data_add	kmemdup	kfree
imx_pd_bind	kmemdup	kfree	mlx5_fw_fatal_reporter_dump	kvmalloc	kvfree
hgcm_call_preprocess_linaddr	kvmalloc	kvfree	btrfs_mount_root	kvzalloc	kvfree
btrfsic_mount	kvzalloc	kvfree	mlx5_fpga_conn_create_cq	kvzalloc	kvfree
do_shmat	kzalloc	kfree	btrfs_mount_root	kzalloc	kfree
sg_io	kzalloc	kfree	cifs_sb_tlink	kzalloc	kfree
btrfs_qgroup_trace_extent	kzalloc	kfree	btrfs_add_delayed_data_ref	kzalloc	kfree
subscribe_event_xa_alloc	kzalloc	kfree	orangefs_mount	kzalloc	kfree
v9fs_mount	kzalloc	kfree	fastrpc_dma_buf_attach	kzalloc	kfree
vmw_cmdbuf_res_add	kzalloc	kfree	netlbl_unlabel_defconf	kzalloc	kfree
sdma_init	kzalloc	kfree	bfad_im_get_stats	kzalloc	kfree
rpmsg_eptdev_write_iter	kzalloc	kfree	qrtr_tun_write_iter	kzalloc	kfree
sof_dfsentry_write	kzalloc	kfree	cx23888_ir_probe	kzalloc	kfree
komeda_wb_connector_add	kzalloc	kfree	sof_set_get_large_ctrl_data	kzalloc	kfree
i40e_setup_macvlans	kzalloc	kfree	v3d_idle_axi	kzalloc	kfree
i2400m_op_rfkill_sw_toggle	kzalloc	kfree	nfp_flower_spawn_phy_reprs	kzalloc	kfree
nfp_flower_spawn_vnic_reprs	kzalloc	kfree	dce100_create_resource_pool	kzalloc	kfree
dce100_clock_source_create	kzalloc	kfree	__ipmi_bmc_register	kzalloc	kfree
bcm2835_timer_init	kzalloc	kfree	ttc_setup_clockevent	kzalloc	kfree
davinci_timer_register	kzalloc	kfree	cpufreq_dbs_governor_init	kzalloc	kfree
dce110_clock_source_create	kzalloc	kfree	dce110_create_resource_pool	kzalloc	kfree
dce112_create_resource_pool	kzalloc	kfree	dce112_clock_source_create	kzalloc	kfree
dce120_create_resource_pool	kzalloc	kfree	dce120_clock_source_create	kzalloc	kfree
dce80_clock_source_create	kzalloc	kfree	dce80_create_resource_pool	kzalloc	kfree
dce81_create_resource_pool	kzalloc	kfree	dce83_create_resource_pool	kzalloc	kfree
dcn10_create_resource_pool	kzalloc	kfree	dcn10_clock_source_create	kzalloc	kfree
dcn20_clock_source_create	kzalloc	kfree	nouveau_bo_new	kzalloc	kfree
aspeed_video_probe	kzalloc	kfree	vsp1_dl_list_alloc	kzalloc	kfree
fastrpc_buf_alloc	kzalloc	kfree	arc_mdio_probe	mdiobus_alloc	kfree
xge_mdio_config	mdiobus_alloc	kfree	e100_loopback_test	netdev_alloc_skb	consume_skb
nf_tables_addchain	nla_strdup	kfree	nfnl_cthelper_get	nlmsg_new	nlmsg_free
ctnetlink_conntrack_event	nlmsg_new	nlmsg_free	cttimeout_get_timeout	nlmsg_new	nlmsg_free
cttimeout_default_get	nlmsg_new	nlmsg_free	ctnetlink_get_conntrack	nlmsg_new	nlmsg_free
crypto_reportstat	nlmsg_new	nlmsg_free	crypto_report	nlmsg_new	nlmsg_free
ap_flash_init	of_find_matching_node	of_node_put	ti_tscadc_probe	of_get_child_by_name	of_node_put
fimc_is_probe	of_get_child_by_name	of_node_put	create_sysclk	of_get_child_by_name	of_node_put
st_dwc3_probe	of_get_child_by_name	of_node_put	vpif_probe	of_graph_get_next_endpoint	of_node_put
imxfb_probe	of_parse_phandle	of_node_put	ti_pipe3_get_sysctrl	of_parse_phandle	of_node_put
of_msi_get_domain	of_parse_phandle	of_node_put	snd_rk_mc_probe	of_parse_phandle	of_node_put
qcom_smd_parse_edge	of_parse_phandle	of_node_put	mxs_saif_probe	of_parse_phandle	of_node_put
dvic_probe_of	of_parse_phandle	of_node_put	of_nvmem_cell_get	of_parse_phandle	of_node_put
hns_roce_v1_reset	of_parse_phandle	of_node_put	smsm_parse_ipc	of_parse_phandle	of_node_put
smp2p_parse_ipc	of_parse_phandle	of_node_put	dsa_port_parse_of	of_parse_phandle	of_node_put
rsnd_ssiu_probe	rsnd_ssiu_of_node	of_node_put	rtsx_probe	scsi_host_alloc	scsi_host_put
myrb_detect	scsi_host_alloc	scsi_host_put	macsec_encrypt	skb_copy_expand	kfree_skb
macsec_encrypt	skb_unshare	consume_skb	macsec_decrypt	skb_unshare	consume_skb
at91_rtc_probe	syscon_node_to_regmap	of_node_put	tm6000_start_stream	usb_alloc_urb	usb_free_urb
usbduxfast_auto_attach	usb_alloc_urb	usb_free_urb	rtl819xU_tx_cmd	usb_alloc_urb	usb_free_urb
gs_can_open	usb_alloc_urb	usb_free_urb	ath10k_usb_hif_tx_sg	usb_alloc_urb	usb_free_urb
rtl8192_tx	usb_alloc_urb	usb_free_urb	kmemleak_test_init	vmalloc	vfree

TABLE V: List of new memory leaks found in the Linux kernel 5.2.13-stable

Allocating Function	FOI	Missing Release
acpi_ut_create_package_object	acpi_ut_create_internal_object	acpi_ut_remove_reference
ocfs2_initialize_super	alloc_ordered_workqueue	destroy_workqueue
adf7242_probe	alloc_ordered_workqueue	destroy_workqueue
nicvf_probe	alloc_ordered_workqueue	destroy_workqueue
i40iw_setup_cm_core	alloc_ordered_workqueue	destroy_workqueue
ca8210_dev_com_init	alloc_ordered_workqueue	destroy_workqueue
rsxx_dma_ctrl_init	alloc_ordered_workqueue	destroy_workqueue
kvaser_pci_add_chan	alloc_sja1000dev	free_sja1000dev
mwifiex_add_card	alloc_workqueue	destroy_workqueue
mwifiex_reinit_sw	alloc_workqueue	destroy_workqueue
ath10k_tm_cmd_wmi	ath10k_wmi_alloc_skb	consume_skb
create_key_field	create_hist_field	destroy_hist_field
__qedi_probe	create_singlethread_workqueue	destroy_workqueue
ttm_mem_global_init	create_singlethread_workqueue	destroy_workqueue
nandsim_debugfs_create	debugfs_create_file	debugfs_remove
mwifiex_pcie_alloc_cmdrsp_buf	dev_alloc_skb	kfree_skb
mwifiex_pcie_init_evt_ring	dev_alloc_skb	kfree_skb
rsi_send_block_unblock_frame	dev_alloc_skb	kfree_skb
lm36274_parse_dt	device_for_each_child_node	fwnode_node_put
lm3692x_probe_dt	device_get_next_child_node	fwnode_node_put
iwl_pcie_ctxt_info_gen3_init	dma_alloc_coherent	dma_free_coherent
s3c_fb_probe_win	framebuffer_alloc	framebuffer_release
chtls_recv_sock	inet_csk_route_child_sock	dst_release
psxpad_spi_probe	input_allocate_polled_device	input_free_polled_device
kimage_crash_copy_vmcoreinfo	kimage_alloc_control_pages	kfree
btrfs_add_delayed_data_ref	kmem_cache_alloc	kmem_cache_free
__btrfs_add_free_space	kmem_cache_zalloc	kmem_cache_free
tcp_accept_from_sock	kmem_cache_zalloc	kmem_cache_free
etnaviv_gem_prime_import_sg_table	kvmalloc_array	kvfree
alcor_pci_sdmmc_drv_probe	mmc_alloc_host	mmc_free_host
nl80211_get_ftm_responder_stats	nlmsg_new	nlmsg_free
ocfs2_acl_chmod	ocfs2_get_acl_nolock	posix_acl_release
meson_mx_socinfo_init	of_find_matching_node	of_node_put
of_flash_probe_versatile	of_find_matching_node_and_match	of_node_put
usbhs_rza1_hardware_init	of_find_node_by_name	of_node_put
mscc_ocelot_probe	of_get_child_by_name	of_node_put
rtl8366rb_setup_cascaded_irq	of_get_child_by_name	of_node_put
of_get_devfreq_events	of_get_child_by_name	of_node_put
max77620_initialise_fps	of_get_child_by_name	of_node_put
dwc3_qcom_of_register_core	of_get_child_by_name	of_node_put
axp20x_regulator_parse_dt	of_get_child_by_name	of_node_put
gpiod_get_from_of_node	of_get_named_gpiod_flags	gpiod_put
of_fwnode_graph_get_port_parent	of_get_parent	of_node_put
ath10k_qmi_setup_msa_resources	of_parse_phandle	of_node_put
dwc3_pci_probe	platform_device_alloc	platform_device_put
spi_gpio_probe	spi_alloc_master	spi_controller_put
meson_mx_socinfo_init	syscon_node_to_regmap	of_node_put
npcm7xx_ehci_hcd_drv_probe	syscon_regmap_lookup_by_compatible	of_node_put
meson_mx_socinfo_init	syscon_regmap_lookup_by_compatible	of_node_put
clps711x_keypad_probe	syscon_regmap_lookup_by_compatible	of_node_put
lpc18xx_dwmac_probe	syscon_regmap_lookup_by_compatible	of_node_put
fsl_sai_probe	syscon_regmap_lookup_by_compatible	of_node_put
spi_clps711x_probe	syscon_regmap_lookup_by_compatible	of_node_put
rk3036_codec_platform_probe	syscon_regmap_lookup_by_phandle	of_node_put
syscon_reboot_probe	syscon_regmap_lookup_by_phandle	of_node_put
gemini_sata_probe	syscon_regmap_lookup_by_phandle	of_node_put
rk3x_i2c_probe	syscon_regmap_lookup_by_phandle	of_node_put
rk3328_platform_probe	syscon_regmap_lookup_by_phandle	of_node_put
rk_spdif_probe	syscon_regmap_lookup_by_phandle	of_node_put
gsbi_probe	syscon_regmap_lookup_by_phandle	of_node_put
imx_init_from_tempmon_data	syscon_regmap_lookup_by_phandle	of_node_put
pistachio_clksrc_of_init	syscon_regmap_lookup_by_phandle	of_node_put
zynq_fpga_probe	syscon_regmap_lookup_by_phandle	of_node_put
create_event_toplevel_files	tracefs_create_dir	tracefs_remove_recursive
usbduxsigma_alloc_usb_buffers	usb_alloc_urb	usb_free_urb
rtl8xxxu_submit_int_urb	usb_alloc_urb	usb_free_urb
emmaprp_probe	video_device_alloc	v4l2_device_unregister
xfrm_bundle_lookup	xfrm_policy_lookup	xfrm_pols_put

TABLE V: List of new memory leaks found in the Linux kernel 5.2.13-stable (continued)

CVE #	File:LineNumber
CVE-2019-18807	drivers/net/dsa/sja1105/sja1105_spi.c:405
CVE-2019-18808	drivers/crypto/ccp/ccp-ops.c:1758
CVE-2019-18809	drivers/media/usb/dvb-usb/af9005.c:965
CVE-2019-18810	drivers/gpu/drm/arm/display/komeda/komeda_wb_connector.c:151
CVE-2019-18811	sound/soc/sof/ipc.c:571
CVE-2019-18812	sound/soc/sof/debug.c:137
CVE-2019-18813	drivers/usb/dwc3/dwc3-pci.c:234
CVE-2019-19043	drivers/net/ethernet/intel/i40e/i40e_main.c:7187
CVE-2019-19044	drivers/gpu/drm/v3d/v3d_gem.c:541
CVE-2019-19045	drivers/net/ethernet/mellanox/mlx5/core/fpga/conn.c:460
CVE-2019-19047	drivers/net/ethernet/mellanox/mlx5/core/health.c:570
CVE-2019-19048	drivers/virt/vboxguest/vboxguest_utils.c:219
CVE-2019-19050	crypto/crypto_user_stat.c:315
CVE-2019-19051	drivers/net/wimax/i2400m/op-rfkill.c:85
CVE-2019-19052	drivers/net/can/usb/gs_usb.c:587
CVE-2019-19053	drivers/rpmsg/rpmsg_char.c:226
CVE-2019-19054	drivers/media/pci/cx23885/cx23888-ir.c:1165
CVE-2019-19056	drivers/net/wireless/marvell/mwifiex/pcie.c:1028
CVE-2019-19057	drivers/net/wireless/marvell/mwifiex/pcie.c:684
CVE-2019-19058	drivers/net/wireless/intel/iwlwifi/fw/dbg.c:535
CVE-2019-19059	drivers/net/wireless/intel/iwlwifi/pcie/ctxt-info-gen3.c:73
CVE-2019-19060	drivers/iio/imu/adis_buffer.c:80
CVE-2019-19061	drivers/iio/imu/adis_buffer.c:33
CVE-2019-19062	crypto/crypto_user_base.c:203
CVE-2019-19063	drivers/net/wireless/realtek/rtlwifi/usb.c:1034
CVE-2019-19065	drivers/infiniband/hw/hfi1/sdma.c:1522
CVE-2019-19066	drivers/scsi/bfa/bfad_attr.c:267
CVE-2019-19068	drivers/net/wireless/realtek/rtl8xxxu/rtl8xxxu_core.c:5435
CVE-2019-19069	drivers/misc/fastrpc.c:494
CVE-2019-19071	drivers/net/wireless/rsi/rsi_91x_mgmt.c:1575
CVE-2019-19072	kernel/trace/trace_events_filter.c:436
CVE-2019-19073	drivers/net/wireless/ath/ath9k/htc_hst.c:151
CVE-2019-19074	drivers/net/wireless/ath/ath9k/wmi.c:308
CVE-2019-19075	drivers/net/ieee802154/ca8210.c:3149
CVE-2019-19077	drivers/infiniband/hw/bnxt_re/ib_verbs.c:1406
CVE-2019-19078	drivers/net/wireless/ath/ath10k/usb.c:414
CVE-2019-19079	net/qrtr/tun.c:83
CVE-2019-19080	drivers/net/ethernet/netronome/nfp/flower/main.c:515
CVE-2019-19081	drivers/net/ethernet/netronome/nfp/flower/main.c:400
CVE-2019-19082	drivers/gpu/drm/amd/display/dc/dce100/dce100_resource.c:1088
CVE-2019-19083	drivers/gpu/drm/amd/display/dc/dce100/dce100_resource.c:660

TABLE VI: List of new memory leak CVEs