MLEE: Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernels

Wenwen Wang, *University of Georgia*

https://www.usenix.org/conference/atc21/presentation/wang-wenwen

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

MLEE: Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernels

Wenwen Wang
University of Georgia

Abstract

Memory leaks in operating system (OS) kernels can cause critical performance and security issues. However, it is quite challenging to detect memory leaks due to the inherent complexity and large-scale code base of real-world OS kernels. In this work, inspired by the observation that software bugs are often hidden in rarely-tested program paths, we focus on detecting memory leaks on *early-exit (E-E) paths* in OS kernels. To this end, we conduct a systematic study of memory management operations involved on E-E paths in OS kernels. Based on the findings, we design a novel leak detector for OS kernels: MLEE, which intelligently discovers memory leaks on E-E paths by *cross-checking* the presence of memory deallocations on different E-E paths and normal paths. MLEE successfully reports 120 *new* memory leak bugs in the Linux kernel. It is the first time these memory leaks are uncovered by a leak detector for OS kernels.

1 Introduction

Memory leaks are a common class of memory management bugs and wide spread in many critical software systems. The accumulation of leaked memory objects can eventually exhaust the limited memory resource, leading to a significant influence on system response time and throughput [6, 18, 37], as well as shortened battery life on mobile devices [43]. Besides, memory leaks have been exploited to launch security attacks, e.g., CVE-2019-12379 [39] and CVE-2019-8980 [40].

Over the past decades, significant efforts have been devoted to detecting memory leaks [4, 7, 14, 16, 22, 27, 31, 32, 44–46]. However, most of them are designed based on the structures and features of user-level applications and thus cannot be applied directly to operating system (OS) kernels. Compared to user applications, OS kernels are much more complicated in terms of program logic and semantic [33], as they need to handle various exceptional statuses, respond to arbitrary interrupts from peripheral devices, perform security checks on untrusted data sources, and etc. Besides, given the gigantic code base of a typical OS kernel, e.g., 25 million source code lines of the Linux kernel with hundreds of new lines added per hour, it is extremely challenging to complete the leak detection for the entire kernel within an acceptable time.

```
1  /* mm/mempool.c */
2  int mempool_resize(mempool_t *pool, int new_min_nr) {
3    ...
4    spin_lock_irqsave(&pool->lock, flags);
5    if (new_min_nr <= pool->min_nr) {
6      spin_unlock_irqrestore(&pool->lock, flags);
7      kfree(new_elements); // A memory deallocation.
8      return 0;
9    }
10   ...
11   return 0;
12 }
```

Figure 1: An example of E-E path in the Linux kernel.

In this paper, we focus on detecting memory leaks on *early-exit paths* (or E-E paths for short) in OS kernels. This is inspired by the observation that software bugs often lurk in rarely-tested program paths [7, 44]. In general, an E-E path is designed to exit from a kernel routine as early as possible. But, before the routine is exited from the E-E path, some extra work usually needs to be completed, e.g., deallocating a memory object, as shown by the example in Figure 1. Hence, if a memory deallocation is required but *missed* on the E-E path, it constitutes a memory leak. Though recent work also attempts to find software bugs related to E-E paths [19, 21, 35], their schemes are limited to a specific type of E-E paths, i.e., error handlers, and thus cannot cover a broad range of buggy E-E paths. Our study reveals that, besides error handlers, E-E paths in OS kernels are used in many *previously-unknown* scenarios, which inherently render existing approaches ineffective to detect memory leaks on general E-E paths.

There are two typical reasons why memory leaks are particularly common on E-E paths. First, the major purpose of an E-E path is to exit from a kernel routine as soon as possible when a special system status is encountered. With this in

mind, it is fairly easy for a developer to compose an E-E path with some required work missed on the path, especially when the developer does not have a comprehensive knowledge of the required work. Second, in practice, many E-E paths are added to the kernel code base not during the development but after the deployment due to various reasons, e.g., correcting the processing logic, achieving better performance/reliability, or enhancing the security. This inevitably leads the E-E paths to the lack of complete and thorough testing, which has been demonstrated by previous work [17].

To understand how E-E paths are used in OS kernels and how memory management operations are involved on E-E paths, we systematically investigate a substantially large portion, i.e., around 1 million lines, of the source code of two popular OS kernels, Linux [2] and FreeBSD [1]. The study uncovers some interesting findings on common usage scenarios of E-E paths and general principles of memory deallocations on E-E paths. In particular, we observe that the presence *inconsistencies* of memory deallocations on different E-E paths and normal paths usually indicate potential memory leaks. Inspired by this observation, we design MLEE to intelligently detect memory leaks on E-E paths through *cross-checking* the presence of memory deallocations on different paths. MLEE also employs several novel static analysis techniques to balance the analysis efficiency and detection accuracy.

We have implemented MLEE based on LLVM [24], which is a popular compiler infrastructure. We evaluate MLEE by applying it to the Linux kernel. MLEE is able to complete the analysis for the entire Linux kernel in around half an hour. This shows the analysis efficiency and scalability of MLEE. By manually analyzing the report produced by MLEE, we finally confirm 120 new leaks. It is the first time these bugs are uncovered by a leak detector. Most of them have been acknowledged by Linux developers and fixed using our patches. For the others, we are working closely with the kernel developers to finalize the patches.

In summary, this paper makes the following contributions:

- We conduct a comprehensive study of E-E paths in OS kernels and involved memory management operations. The study discovers some interesting and inspiring findings. To the best of our knowledge, this is the first systematic study of memory management operations on E-E paths.
- We design MLEE, which employs effective and scalable static program analysis techniques to identify E-E paths and analyze memory deallocations for memory leak detection. We believe the proposed analysis techniques will also benefit similar bug-detection systems.
- We find 120 new memory leak bugs in the Linux kernel with the help of MLEE. It is the first time that these bugs are reported by a leak detector. Most of these bugs have been acknowledged by Linux maintainers and fixed by our patches. This demonstrates the capability of MLEE to detect memory leaks in a real-world OS kernel.

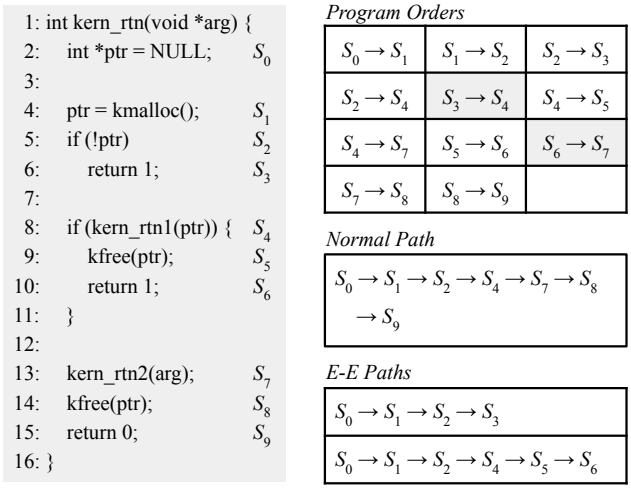


Figure 2: An example of the definition of E-E paths.

2 Background and Motivation

This section gives a formal definition of E-E paths and elaborates common usage scenarios of E-E paths in OS kernels.

2.1 What is E-E Path?

Intuitively, an E-E path intends to exit from a kernel routine *earlier* than normal program paths, by skipping some of the program code in the routine without execution. Hence, an E-E path generally has less program statements than a normal program path. We next give a definition of E-E paths.

A Formal Definition. We use a two-tuple to denote a kernel routine: $R =< S, O >$, where S is a set of statements: $S_0, ..., S_n$, and O is a set of program orders between two statements: $S_i \rightarrow S_j, 0 \leq i, j \leq n$. Each statement represents a basic and concrete operation, e.g., assigning a value to a kernel variable or invoking a callee routine. The program orders specify how the statements can be executed according to the orders in which they show up in the source code. For example, $S_i \rightarrow S_j$ means that S_j can be executed after the execution of S_i because, for example, S_j shows up in the source code immediately after S_i. In this paper, we call S_j a *successor* statement of S_i. It is worth noting that a return statement may also have a successor statement under O, as it may show up in the middle of a routine. This is the key point of E-E paths.

Given $R =< S, O >$, a *program path* of R is an ordered list of statements: $P = S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_m$, where

(i) $S_i \in S, 0 \leq i \leq m$;

(ii) $S_i \rightarrow S_{i+1} \in O, 0 \leq i \leq m-1$;

(iii) S_m is a return statement.

We enforce a return statement at the end of a program path to facilitate the following definition of E-E path. In case there

```c
/* ipc/mqueue.c */
static void mqueue_evict_inode(struct inode *inode)
{
  ...
  clear_inode(inode);
  if (S_ISDIR(inode->i_mode))
    return;
  /* Evict the inode from the message queue. */
  ...
}
```

Figure 3: An E-E path for irrelevant kernel state bypassing.

is no return statement in a kernel routine, we can append an implicit return statement at the end of the routine source code. P is an E-E path if the following condition satisfies:

$$\exists S_n \in S, \text{s.t. } S_m \rightarrow S_n \in O \tag{1}$$

The rationale behind this definition is that the return statement of an E-E path P is *not* the last statement of the routine, i.e., it has a successor statement under O. Hence, if the routine exits from P, the successor statement and the following statements will not be executed, leading to an *early* exit.

An Example. We next use the example in Figure 2 to explain the above definition. The left side of the figure shows the source code of the kernel routine, which has ten statements with two if branches: S_2 and S_4, and three returns: S_3, S_6, and S_9. The top right side of the figure shows the program orders. Note that the two return statements S_3 and S_6 have successor statements, as highlighted in the figure. This routine has three possible program paths and two of them are E-E paths. For example, $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$ is an E-E path, as S_3 has a successor statement under the program orders: $S_3 \rightarrow S_4$.

2.2 Usage Scenarios of E-E Paths

Though E-E paths are often used to handle unexpected errors, e.g., memory allocation failures and invalid function arguments, our study on popular OS kernels, i.e., Linux and FreeBSD, reveals that E-E paths are also composed in other usage scenarios. We next describe each scenario in detail.

Error/Exception Handling. In general, an OS kernel needs to deal with various exceptional system statuses and unexpected program errors. For example, when a permission check fails, the kernel should not continue to perform the following privileged and safety-critical operations. Otherwise, potential security risks will be raised. Therefore, E-E paths are extensively used in OS kernels to terminate unsafe and invalid execution when an error/exception happens.

Typically, an E-E path in this usage scenario returns an *error code* to indicate that an error is triggered. This is also the reason why existing approaches leverage error codes to detect error handlers [12, 19]. However, our study shows that this is not always true. Specifically, we find that 52% (out of 5910) and 25% (out of 1700) E-E paths in Linux and FreeBSD respectively do not return any error code even if they handle errors/exceptions. In fact, whether an E-E path returns an error code primarily depends on what error/exception is handled by the E-E path. Thus, it is insufficient to detect E-E paths merely relying on whether an error code is returned.

```c
/* fs/fat/misc.c */
void fat_time_unix2fat(struct msdos_sb_info *sbi, ...)
{
  ...
  if (tm.tm_year < 1980 - 1900) {
    *time = 0;
    *date = cpu_to_le16((0 << 9) | (1 << 5) | 1);
    return;
  }
  if (tm.tm_year > 2107 - 1900) {
    *time = cpu_to_le16((23 << 11) | (59 << 5) | 29);
    *date = cpu_to_le16((127 << 9) | (12 << 5) | 31);
    return;
  }
  ...
}
```

Figure 4: Two E-E paths for kernel functionality extension.

Irrelevant Kernel State Bypassing. In this scenario, the E-E paths are used to bypass specific kernel states, as it is not necessary for the following program code to process these states. The specific kernel states are often represented in the form of flags, modes, sizes, capacities, properties, etc. Figure 3 shows such an example. Here, if `inode` indicates not a regular file but a directory, it is unnecessary to evict it from the message queue. Thus, an E-E path is created to skip such inodes.

It is worth pointing out that this usage scenario is *not* same as the previous scenario. The major difference is that the E-E paths in this scenario handle *legal* and *valid* kernel states, while the E-E paths in the previous scenario tackle *unexpected* kernel errors/exceptions. That is, the E-E paths in this scenario may show up in a routine even if no error/exception needs to be handled. Besides, if an E-E path in this scenario returns a value, it usually returns the same value as normal paths. In contrast, an E-E path in the previous scenario typically returns a value (if any) different from those of normal paths.

Kernel Functionality Extending. An E-E path in this usage scenario extends existing program logic in a kernel routine to incorporate additional functionalities. Figure 4 shows an example. Here, the routine converts a UNIX date to a (time, date) pair in the FAT file system. However, FAT only supports years between 1980 and 2107. Hence, two E-E paths are created for years outside of this range. E-E paths in this scenario often contain extra statements to extend the functionalities. A representative example of E-E paths in this scenario is *fast paths*, which aim to accelerate the processing of some special cases. Previous research shows that fast paths are very likely to introduce bugs [17], which aligns with our observations.

2.3 Memory Management on E-E Paths

On the surface, memory management has no relationship with E-E paths. However, our experience shows that more than 60% kernel routines that have memory management operations contain at least one E-E path. So a further study is necessary to understand how memory management operations are involved on E-E paths. We next report our observations.

Observation 1. *If a memory object is allocated in a kernel routine, it usually needs to be deallocated on the following E-E paths in this routine.* In principle, after a memory object is allocated, it should be used in the following computations. However, in reality, it is possible that an E-E path is reached before the object is actually used. Hence, the object has to be cleaned up on the E-E path, otherwise a memory leak may be introduced. This observation implies that memory deallocation is an important component of the cleanup work on E-E paths. In fact, 58% and 41% E-E paths investigated by our study in Linux and FreeBSD respectively contain at least one memory deallocation. On the other side, if an allocated memory object is used before an E-E path is taken, it is probably not required to deallocate this object on the E-E path. But, we indeed witness some cases where memory objects are deallocated on the following E-E paths even if they are used before. This demonstrates the close correlation between memory deallocations and E-E paths.

Observation 2. *If a memory object is deallocated on the normal paths of a kernel routine, it usually also needs to be deallocated on the E-E paths in this routine.* This observation shows the consistency between normal paths and E-E paths on deallocating memory objects. If an object is deallocated on normal paths, it implies that the live range of this memory object is limited to the current kernel routine, so it should be deallocated on E-E paths as well. We find that 50% and 48% of kernel routines in Linux and FreeBSD respectively have similar deallocations on E-E paths and normal paths. This provides a strong evidence for the analyses in MLEE, as we will see in Section 4. In particular, MLEE pays special attention to memory deallocations on normal paths and employs effective and scalable static analyses to check whether these deallocations are also required on E-E paths. This allows MLEE to detect required but missed deallocations on E-E paths.

Observation 3. *If a memory object is deallocated on an E-E path in a kernel routine, it usually needs to be deallocated on the following E-E paths with same/similar return values (if any) in the routine.* If two E-E paths in a routine return same/similar values, it usually indicates that these two paths are constructed for same/similar purposes, e.g., in the same usage scenario. Particularly, if a memory object is deallocated on an E-E path, it means the caller routine is irresponsible to deallocate this object under the return value of the E-E path, and thus, this object should be deallocated on the following E-E paths with same/similar return values. Therefore, MLEE creates scalable and precise static analyses to detect memory deallocations present on some E-E paths but missed on others.

Observation 4. *If multiple memory objects are allocated in a kernel routine, all objects allocated before an allocation failure usually need to be deallocated on the E-E path corresponding to this allocation failure.* One of the common reasons for multiple memory allocations in the same routine is to allocate *semantically-correlated* memory objects. For example, the allocation of a memory object with a subfield pointing to a child object is often realized by firstly allocating the object itself and then allocating the child object. Thus, if the allocation for the child object fails, the previously-allocated object has to be deallocated. This observation drives MLEE to thoroughly inspect kernel routines with multiple memory allocations to detect memory leaks. If one of the memory objects is deallocated on an E-E path but not on a following E-E path, it probably indicates a memory leak.

3 Issues and Challenges of MLEE

The design of MLEE is inspired by the key insight that an *inconsistency* between the presence of a memory deallocation on an E-E path and its presence on a normal path or another E-E path in the same kernel routine very likely indicates a potential memory leak. Therefore, it is possible to detect memory leaks by *cross-checking* memory deallocations on different E-E paths and normal paths in the same routine. Though the idea is intuitive, there are several technical challenges.

How to identify early-exit paths? Given the vast amount and the complicated logic of the source code in an OS kernel, e.g., more than 25 million lines in Linux, it is quite challenging to precisely identify E-E paths in a wide range of kernel modules with significant semantic diversities. Besides, the various usage scenarios of E-E paths, as described in Section 2.2, may lead E-E paths to exhibit different characteristics at the source code level. A simple solution for this issue is to enumerate all possible program paths of a kernel routine and exhaustively examine each of them to see whether it satisfies the definition of E-E paths in Section 2.1. However, this solution is obviously impractical due to the poor scalability caused by the well-known path explosion problem.

To address this challenge, we further investigate the structures and semantics of numerous E-E paths, in particular, with detecting memory leaks in mind. We find that the key for MLEE to detect memory leaks on an E-E path is to find out the crucial features of this E-E path that distinguish it from normal paths and other E-E paths in the same routine. Furthermore, such features are often reflected in a specific portion of the E-E path, i.e., the last several statements at the end of the E-E path. Recall the example in Figure 2. Compared to S_0, S_1, and S_2, the statement S_3 is more representative of the first E-E path, because S_3 is manifested *uniquely* on this E-E path. In a similar way, S_5 and S_6 are more unique for the second

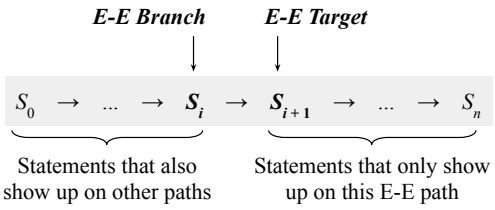


Figure 5: E-E branch and E-E target of an E-E path.

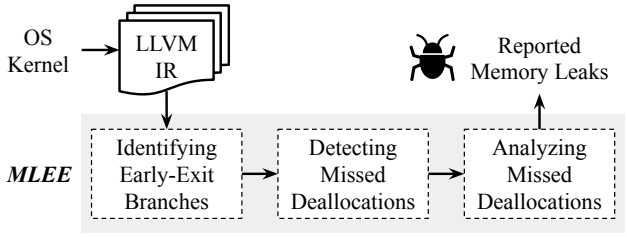


Figure 6: The work flow of MLEE.

E-E path, as they only show up on this E-E path and thus can be used to distinguish this E-E path from other paths.

In addition, we find out that such statements often follow a conditional branch statement, e.g., an if statement. In this paper, we call such a conditional branch as an ***E-E branch***, which is essential to differentiate an E-E path from normal paths and other E-E paths. Similarly, we call the target of the E-E branch that is included by the E-E path as an ***E-E target***, which initiates the following statements after the E-E branch on the E-E path. Figure 5 shows the basic structure of an E-E path. It is worth noting that an E-E branch typically show up on both E-E paths and normal paths. Also, it is possible that an E-E path encompasses more than one E-E branch. Here, we primarily consider the *last* E-E branch on the E-E path because it is this one that leads to the unique statements.

In summary, to detect memory leaks on E-E paths, MLEE first collects E-E branches in a kernel routine and then utilizes them as anchor points of following analyses. This allows MLEE to compose precise and scalable static analyses to identify E-E paths and avoid the path explosion issue.

How to analyze memory deallocations? After the E-E branches in a kernel routine are recognized, MLEE moves forward to analyze memory deallocations in this routine to identify the inconsistencies between their presence on different E-E and normal paths. A simple analysis scheme is to examine each deallocation in the routine and check whether it shows up on an E-E path, in particular, after the E-E branch. If not, MLEE then reports a memory leak on this E-E path. Though this scheme sounds reasonable, it may produce plenty of false positive cases and require significant manual efforts to screen them. The major cause of false positives is that the deallocation may not be required on a specific E-E path. For instance, the memory object to be deallocated may be used, e.g., as the return value, on the E-E path and thus should not be deallocated.

To address this issue, MLEE firstly checks whether a missed memory deallocation is required to show up on an E-E path. This is realized in three steps. First, MLEE ensures that the deallocated object is live on the E-E path via a classical context-sensitive and flow-sensitive liveness analysis. Second, MLEE guarantees the validity of the deallocated object on the E-E path, i.e., the object is successfully allocated and remains allocated. Finally, MLEE confirms that the deallocated object is not used on the E-E path, particularly after the E-E branch (see Section 4 for more details). This way, MLEE can infer whether the deallocation is required on the E-E path or not.

4 MLEE System Design

In this section, we firstly present an overview of how MLEE works and then describe the technical details of MLEE.

Figure 6 illustrates the high-level work flow of MLEE. Essentially, MLEE takes as the input the LLVM IR of the target OS kernel and reports potential memory leaks on E-E paths in each kernel routine. We choose to start from LLVM IR because LLVM has plenty of static analysis passes in place, such as control/data-flow and alias analysis. Besides, LLVM IR has quite comprehensive debugging information (generated with the "`-g`" option), which can map the IR back to the corresponding source code. This allows MLEE to report the source code locations of the detected memory leaks to facilitate further manual inspection and bug fixing. Even though the implementation of MLEE leverages the LLVM infrastructure to reduce engineering effort, it is worth pointing out that the static analyses used by MLEE are *not* provided by LLVM. These analyses are a primary contribution of our work.

For each kernel routine, MLEE firstly analyzes all conditional branches in this routine to identify E-E branches. Next, MLEE gathers memory deallocations in this routine. For each pair of an E-E branch and a memory deallocation, MLEE then checks whether the memory deallocation is *missed* on the E-E paths associated with the E-E branch, i.e., the memory object freed by the deallocation operation is not deallocated after the E-E branch. MLEE focuses the analysis particularly on the statements after the E-E branch, including the E-E target and its following statements, because they are unique and thus more representative of the E-E paths. This also allows MLEE to delimit the analysis scope and scale up the analysis. Besides, it is possible that a memory object is deallocated implicitly after the E-E branch, e.g., in a callee routine. Hence, MLEE creates an inter-procedural analysis to cover all statements in callee routines invoked after the E-E branch.

If the result of the above step is yes, it means the memory deallocation is not present on the associated E-E paths. MLEE then further infers whether the missed deallocation should show up on the E-E path, i.e., whether the correspond-

ing memory object should be deallocated after the E-E branch. To this end, MLEE gathers the liveness, validity, and usage information of the memory object after the E-E branch to guide the analysis. Also, MLEE leverages additional information, e.g., the relative locations of the memory deallocation and the E-E branch in the source code of the kernel routine, to assist the necessity inference, according to the observations in Section 2.3. If the analysis concludes that the memory object needs to be deallocated after the E-E branch, MLEE will report a memory leak bug along with the source code locations of the E-E branch and the missed memory deallocation.

4.1 Identifying E-E Branches

Typically, a conditional branch comes with two branch targets, which are taken when the specified condition is satisfied or not, respectively. The major difference of an E-E branch, compared to a regular conditional branch, is that one of the branch targets is an E-E target, which only shows up on E-E paths, and thus leads to the last several statements on an E-E path, Hence, MLEE determines whether a conditional branch is an E-E branch based on the following two conditions:

- **Condition-1:** The program paths starting from one target always reach to a return statement that has a successor statement under the program orders of the routine.
- **Condition-2:** The program paths starting from another target always have a possibility to reach to a return statement that has no successor statement under the program orders of the routine.

A conditional branch is identified as an E-E branch if its two branch targets satisfy the above two conditions. Here, it is not hard to understand **Condition-1**, because one branch target of an E-E branch should always lead to E-E paths. In fact, the branch target that satisfies **Condition-1** is the E-E target of the E-E branch. The purpose of **Condition-2** is to emphasize the possibility of an E-E branch to show up on normal paths. In essence, an E-E branch divides the following execution space of the current routine into two *disjoint* sets. One of the sets comprises executions only ending with early exits, while another set includes at least one execution ending with normal exit. Therefore, **Condition-2** is crucial to distinguish an E-E branch from a conditional branch both of whose branch targets merely lead to E-E paths.

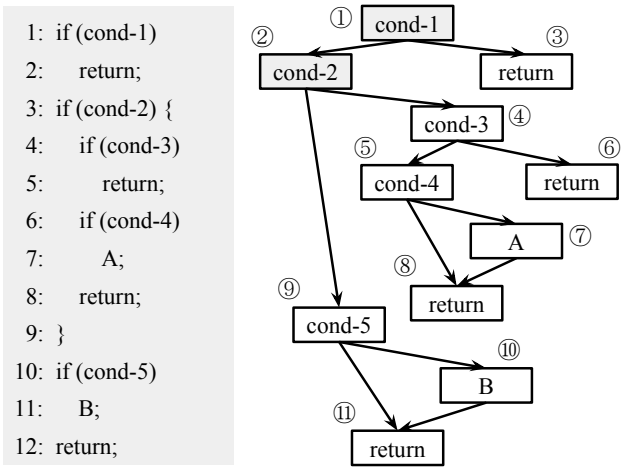
According to these two conditions, MLEE creates an effective static analysis to identify E-E branches by carefully checking branch targets of each conditional branch. Algorithm 1 shows the details of the static analysis. Overall, for each conditional branch in the kernel routine, MLEE firstly collects its two branch targets, i.e., the two first statements executed immediately after the conditional branch is taken and not taken, respectively, and then checks whether they satisfy the two conditions. If yes, MLEE then identifies this conditional branch as an E-E branch and places it into the set of identified E-E branches. Otherwise, MLEE continues to check next conditional branch.

Algorithm 1: Identification of Early-Exit Branches

Input: R - a kernel routine
Output: $EEBSet$ - the set of early-exit branches in R

```
1  EEBSet ← ∅;
2  CBSet ← Collect_Conditional_Branches (R);
3  for CB ∈ CBSet do
4      BTSet ← Collect_Branch_Targets (CB);
5      for T1 ∈ BTSet do
6          RetSet1 ← Collect_Reachable_Returns (T1);
7          Flag ← TRUE;
8          for Ret ∈ RetSet1 do
9              if Ret has no successor statement then
10                 Flag ← FALSE;
11                 break;
12             end
13         end
14         if Flag ≠ TRUE then
15             continue;
16         end
17         for T2 ∈ BTSet \ {T1} do
18             RetSet2 ← Collect_Reachable_Returns (T2);
19             for Ret ∈ RetSet2 do
20                 if Ret has no successor statement then
21                     EEBSet ← EEBSet ∪ {CB};
22                     break;
23                 end
24             end
25         end
26     end
27 end
28 return EEBSet;
```

To determine if a branch target satisfies one of the two conditions, MLEE traverses the control-flow graph (CFG) of the kernel routine to gather all return statements that are reachable from the branch target. A potential issue in this traversing process is how to handle loop structures. Since MLEE mainly intends to visit all reachable nodes starting from the branch target on the CFG, MLEE iterates a loop as many times as necessary until all of its nodes are visited. This allows MLEE to exhaustively capture all return statements in a loop, which actually are quite common in OS kernels. Once the reachable return statements are collected for the branch target, MLEE next checks each of them to confirm whether it has a successor statement under the program orders of the routine. Given that the static analysis works at the LLVM IR level, MLEE needs to resolve the source location of a return statement. To this end, MLEE leverages the debugging information embedded into the LLVM IR to establish the mapping between the LLVM IR and the corresponding source code [41]. This way, MLEE can determine whether the branch target satisfies one of the two conditions mentioned above.



(a) Sample code (b) The CFG of the code in (a)

Figure 7: Conditional branch analysis for E-E path detection.

Now, let us use the example in Figure 7 to understand how MLEE identifies E-E branches. The left side of the figure presents the source code and the right side shows the corresponding CFG. From the figure, we can see that there are five conditional branches and four return statements in this example. Besides, each return statement has a successor statement under the program orders except the one at line 12.

MLEE consecutively analyzes each conditional branch to determine whether it is an E-E branch. Take the conditional branch of `cond-1` at line 1 as an example. The *true* target of this conditional branch can only reach to the return statement at line 2, i.e., ① → ③ in the CFG. On the other hand, the *false* target of this conditional branch may reach to the return statement at line 12, e.g., through the program path ① → ② → ⑨ → ⑪ in the CFG. As a result, MLEE identifies this conditional branch as an E-E branch. In a similar way, the conditional branch of `cond-2` is also identified as an E-E branch by MLEE. However, the conditional branches of `cond-3`, `cond-4`, and `cond-5` are not recognized as E-E branches. This is because MLEE figures out that their branch targets do not satisfy the above two conditions after the analysis. For example, the branch targets of the conditional branch of `cond-3` reach to two different return statements at line 5 and line 8, respectively. But both of these two return statements have a successor statement under the program orders. Hence, the conditional branch of `cond-3` is not considered as an E-E branch by MLEE.

4.2 Detecting Missed Memory Deallocations

Given an identified E-E branch EEB, the next task of MLEE is to detect memory deallocations that are potentially missed on E-E paths associated with EEB.

To this end, MLEE first collects memory deallocations in the same kernel routine as EEB. Next, for each deallocation D, which is suppose to deallocate a memory object M, MLEE checks whether M is always deallocated before EEB. MLEE achieves this by traversing the CFG to search for another deallocation D' that also deallocates M. More importantly, D' dominates EEB in the CFG, which means D' is on all program paths from the entry of the routine to EEB and executed before EEB. In other words, if there exists such a D', it implies that M has already been deallocated before EEB and therefore, D should not be considered as a missed deallocation on the E-E paths of EEB. Otherwise, MLEE continues to check whether M is deallocated by the program statements after the E-E target on the E-E paths. If not, MLEE treats D as a missed deallocation of EEB and further checks whether it is necessary on the E-E paths of EEB.

It is possible that M is deallocated in a callee routine of the current kernel routine. Thus, MLEE devises a path-sensitive and context-sensitive inter-procedural analysis to achieve the above goal. Specifically, MLEE analyzes every backward reachable statement starting from the return statement on the E-E paths corresponding to EEB. In case the statement is a call statement, MLEE further includes the statements in the callee routine into the analysis.

MLEE pays special attention to each memory deallocation statement during the analysis, because MLEE needs to validate whether it is used to deallocate M. A memory object is often accessed through *pointers*, which hold the address of the object. Given that an object can be accessed through different pointers, it is *incomplete* to determine whether two objects freed by two memory deallocations are same or not by simply checking whether the two deallocations use the same pointer. To solve this issue, MLEE leverages the alias analysis in LLVM to discover the potential alias relationship between two pointers. This allows MLEE to differentiate memory deallocations for different objects.

4.3 Analyzing Missed Memory Deallocations

Once a missed memory deallocation D^M is identified for an E-E branch EEB, the final step of MLEE is to check whether D^M is necessary on the E-E paths corresponding to EEB, i.e., the memory object M deallocated by D^M should also be deallocated on the E-E paths.

MLEE firstly composes an effective static analysis to validate the liveness and validity of M at the point of EEB, as it may cause unexpected program errors to deallocate an out-of-scope memory object. Specifically, MLEE relies on the analyses in LLVM to determine whether EEB is covered by the liveness range of M. Regarding the validity, MLEE needs to verify the possibility for M to be in an "allocated" state when EEB is reached. This is because EEB may be on a program path that does not allocate M. Hence, MLEE firstly conducts a backward slicing on M starting from D^M to find out the statement that successfully allocates M. Then MLEE performs a reachability analysis from the allocation statement

to EEB by traversing the CFG.

MLEE needs to take care of two special cases during the above analysis process: 1) No allocation statement is found for M because, for example, M is allocated in another kernel routine. To solve this issue, MLEE treats the entry point of the current routine as a *nominal* allocation point of M. 2) Multiple allocations are found for M, due to, for example, different allocation mechanisms adopted by the kernel to allocate M. MLEE deals with this case soundly by collecting all allocation statements that can reach EEB.

In addition to checking the liveness and validity of M, MLEE also needs to ensure that M is not used by the statements after EEB on the related E-E paths. To this end, MLEE employs a forward slicing on M starting from the E-E target of the E-E branch to confirm that M is not used by the following statements. The analysis stops when the return statement on the E-E paths of EEB is reached. Some common usage examples include but not limited to passing M to a callee routine as an argument, accessing a memory location inside of M, calculating effective memory addresses based on the start address of M, and etc. If M is indeed used, MLEE will skip the following analysis on D^M and EEB.

Finally, MLEE infers the necessity of D^M for EEB. To achieve this, MLEE creates a set of *checking rules* mainly based on the observations described in Section 2.3. The rules check various factors of EEB and D^M to heuristically determine whether D^M is required on E-E paths corresponding to EEB. To give an example, one of the rules checks the order of D^M and EEB in which they appear in the source code. This is inspired by the observation that if D^M shows up earlier than EEB in the source code, it typically implies that D^M is executed before EEB and therefore, it is very likely that D^M is not necessary on the corresponding E-E paths. In contrast, if the order is reversed, it is probably essential for the corresponding E-E paths to deallocate M. The output of each rule is either 1 or 0, meaning the deallocation is required or not on the E-E paths, respectively. MLEE assigns an empirical weight in (0, 1) for each rule and calculates the final result using the following formula:

$$f(EEB, D^M) = \sum_i w_i R_i(EEB, D^M) \quad (2)$$

where w_i is the weight of the ith rule: $\sum_i w_i = 1$, and $R_i(EEB, D^M)$ is the output of the ith rule for EEB ad D^M. If the final result exceeds a predefined threshold 0.5, MLEE determines that D^M is required for EEB and reports a memory leak with the detailed information of EEB and D^M.

5 Implementation

We have implemented MLEE as an LLVM (version 8.0.0) tool composing of multiple analysis passes. This section reports the issues we encountered during the implementation of MLEE as well as the solutions we adopted to address them.

Compiling Linux to LLVM IR. Currently, LLVM is not fully compatible with the Linux kernel. To solve this issue, we compose a Python script to automatically extract and adapt the compilation commands to compile the kernel source code to LLVM IR. We simply skip a source file if LLVM cannot compile it. To include as many kernel modules as possible, we use the "allyes" option to configure the kernel. At last, our compilation covers around 11K kernel modules.

Global Call Graph. MLEE conducts the path-sensitive and context-sensitive inter-procedural analyses on a global call graph. To build the graph, MLEE incrementally compose a child call graph for each kernel module when it is loaded and parsed by LLVM [42], and eventually combines them together to obtain the global call graph. Note that MLEE does not need to link all kernel modules together to produce a single LLVM IR file and thus avoids potential linking errors in this process. MLEE also employs a type-based analysis to identify all possible call targets for an indirect call [29, 36].

Alias Analysis. MLEE relies on the alias analysis in LLVM to analyze the alias relationship between two memory pointers. This is achieved by querying the LLVM alias analysis framework with the two memory pointers along with the size of the memory locations. There are four possible relationship outcomes for such a query: `MustAlias`, `PartialAlias`, `MayAlias`, and `NoAlias`. For accuracy consideration, MLEE considers two pointers are alias pointers only when `MustAlias` or `PartialAlias` is returned by the alias analysis framework.

Handling Goto Statements. Some Linux routines use goto statements to realize sophisticated control logic. This entails difficulties for MLEE to detect E-E paths, as the return statement of an E-E path implemented using a goto statement may be placed at the end of a kernel routine. To deal with this issue, MLEE creates a source transformation tool to remove goto statements by replacing them with the target statements they jump to. Note that MLEE only replaces *forward* goto statements because backward goto statements are mostly used to implement loops. Fortunately, the Linux kernel does not use indirect goto statements [11], which take as input label variables and can complicate the transformation.

```
1     if (...)
2       goto out;
3     ...
4     return 0;
5   out:
6     return 1;
```

6 Experimental Study

This section evaluates MLEE by applying it to the Linux kernel (version 5.0, which was the most recent version when we finished the implementation of MLEE). Due to time limitations, MLEE is not evaluated with other OS kernels, e.g., FreeBSD, and non-kernel applications. This is a direction for future work. The evaluation presented in this section would like to answer two research questions about MLEE: i) *Effectiveness*: can MLEE discover new memory leaks in a real-

Table 1: Detection statistics of MLEE. "KEE": kernel routines that have E-E paths. "KMD": kernel routines that have memory deallocations. "KEM": kernel routines that have both E-E paths and memory deallocations. "MMD": E-E branches that are reported by MLEE to have missed memory deallocations.

Kernel Routine				Mem	E-E Branch	
Total	KEE	KMD	KEM	Dealloc	Total	MMD
887877	121829 13.7%	14540 1.6%	7685 0.9%	20751	297451	126 0.04%

world OS kernel? ii) *Efficiency*: can MLEE complete the analysis of an entire OS kernel in an acceptable time?

Table 1 shows the detection statistics. As shown in the table, MLEE finds out that 0.9% kernel routines comprise of both E-E paths and memory deallocation operations. After the analysis, MLEE reports that 126 suspicious E-E branches have missed memory deallocations. These E-E branches span across various kernel modules and account for 0.04% of the total E-E branches identified by MLEE. With further manual analysis and investigation, we can finally confirm memory leak bugs on the corresponding E-E paths.

Finding New Bugs. It takes around 3∼5 minutes for a researcher to manually investigate one suspicious E-E branch. Finally, we confirm 120 memory leaks related to 103 buggy E-E branches. Note that one buggy E-E branch may correspond to multiple missed deallocations. All of these memory leaks are *new* bugs. This demonstrates the capability of MLEE to discover unknown memory leaks in a real-world OS kernel.

Table 2 shows the details of the found memory leaks. We have reported these bugs along with the patches to Linux developers. During the communication with the developers, we were surprised that the developers responded to our bug reports promptly, e.g., in half an hour or even less, and actively worked with us to revise and update the patches. Given the developers' heavy maintenance workload, this once again shows the high priority and severity of fixing memory leaks in OS kernels. In summary, at the time of the paper submission,

- 89 bugs (74.2%) have been fixed in the latest version of the Linux kernel using our patches;
- 16 bugs (13.3%) have been fixed in the latest version of the Linux kernel using others' patches;
- 15 bugs (12.5%) have been confirmed and we are working with the kernel developers to finalize the patches.

Table 2 also shows that a vast majority of memory leaks are detected in the driver (60%), sound (17.5%), and file system (16.7%) directories. This means memory leaks are a general type of software bugs across various kernel modules. We find that 74% missed deallocations are also found on other E-E paths. This shows that E-E paths in the same kernel routine often exhibit similar behaviors in memory management

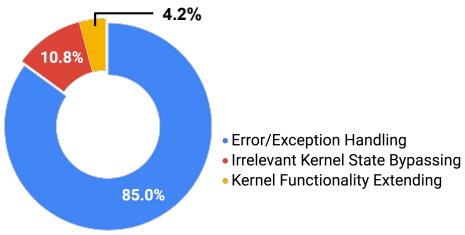


Figure 8: The usage-scenario distribution of the E-E paths on which the memory leaks are detected by MLEE.

```
/* drivers/net/ethernet/mellanox/mlx4/en_rx.c */
int mlx4_en_config_rss_steer(struct mlx4_en_priv *priv)
{
  ...
  err = mlx4_qp_alloc(mdev->dev, priv->base_qpn, ...);
  if (err) {
    en_err(priv, "Failed to allocate RSS ...\n");
    goto rss_err; // rss_map->indir_qp is leaked
  }               // on this early-exit path.
  ...
  kfree(rss_map->indir_qp);
  rss_map->indir_qp = NULL;
rss_err:
  ...
  return err;
}
```

Figure 9: A real memory leak in Linux detected by MLEE.

operations. On the other side, this implies memory deallocations on normal paths may also be required on E-E paths.

An interesting observation on the E-E paths of the E-E branches in Table 2 is that these buggy E-E paths are used in not only the scenario of error/exception handling, but also two other scenarios described in Section 2.2. Figure 8 shows the detailed distribution of the scenarios. This demonstrates that MLEE can detect buggy E-E paths in different usage scenarios. However, existing bug-detection schemes that focus only on error handlers may fail to detect these memory leaks.

To understand when and how the found memory leaks are introduced into the kernel code base, we further investigate the code revision histories of the kernel routines in which the memory leaks are found. We find that many memory leaks were actually brought in at the first time when the corresponding E-E paths were added into the kernel source code. This was also acknowledged by some Linux maintainers. Taking the Bug#100 in Table 2 as an example, when we submitted it to the Linux patch mailing list, a kernel maintainer quickly commented our bug report and said that "*this bug exists till its first commit for v2.6.39*." One possible reason behind this phenomenon is that the author of an E-E path may have a partial and incomplete understanding of the work that are necessary on the E-E path. In addition, this also reveals that many E-E paths in OS kernels are error-prone due to the lack of extensive testing when they are committed to the kernels.

Table 2: New memory leaks in the Linux kernel discovered by MLEE. "O": the bug has been fixed in the latest version using our patch. "L": the bug has been fixed in the latest version using others' patch. "C": the bug has been confirmed.

	Kernel Routine	E-E Branch	Missed Deallocation			Kernel Routine	E-E Branch	Missed Deallocation	
1	block/...:bio...prep	if (ret...)	kfree(buf)	O	61	drivers/...:i24...it	if (res...)	kfree(opt...)	O
2	drivers/...:cm...	if (*ppos...)	kfree(buf)	O	62	drivers/...:e...am	if (e1...)	kfree(tx_old)	O
3	drivers/...:acpi...ble	if (!ac...)	kfree(entry)	O	63	drivers/...:e...am	if (e1...)	kfree(rx_old)	O
4	drivers/...:fs_open	if (!to)	kfree(vcc)	O	64	drivers/...:ath...te	if (!sta...)	kfree(arsta...)	O
5	drivers/...:f...en	if (DO...)	kfree(vcc)	O	65	drivers/...:pr...en	if (ai->...)	kfree(file...)	O
6	drivers/...:f...en	if (!tc)	kfree(vcc)	O	66	drivers/...:pr...en	if (down...)	kfree(file...)	O
7	drivers/...:read...refs	if (xen...)	kfree(req)	O	67	drivers/...:lbs...ate	if (lbs...)	kfree(cmd)	C
8	drivers/...:read...refs	if (xen...)	kfree(ind...)	O	68	drivers/...:mw...d	if (mwi...)	kfree(hos...)	L
9	drivers/...:read...refs	if (xen...)	kfree(seg...)	O	69	drivers/...:mw...d	if (!skb)	kfree(hos...)	L
10	drivers/...:devfreq...	if (err...)	kfree(dev...)	L	70	drivers/...:mw...d	if (err)	kfree(hos...)	L
11	drivers/...:ti_...	if (of_...)	kfree(rsv..)	O	71	drivers/...:fr...pvc	if (*get...)	kfree(pvc)	C
12	drivers/...:omap...	if (de...)	kfree(c)	O	72	drivers/...:ya...tl	if (ym...)	kfree(ym)	C
13	drivers/...:dr...	if (WA...)	kfree(pl...)	C	73	fs/...:btrfs...mod	if (IS...)	kfree(ref)	O
14	drivers/...:md...	if (p_v...)	kfree(dsi...)	C	74	fs/...:btrfs...mod	if (be...)	kfree(ref)	O
15	drivers/...:md...	if (!dsi...)	kfree(dsi...)	C	75	fs/...:btrfs...mod	if (be...)	kfree(ra)	O
16	drivers/...:md...	if (md...)	kfree(dsi...)	C	76	fs/...:btrfs...mod	if (exi...)	kfree(ref)	O
17	drivers/...:c...it	if (pl...)	kfree(pk...)	O	77	fs/...:btrfs...mod	if (act...)	kfree(ref)	O
18	drivers/...:i...ed	if (de...)	kfree(de...)	O	78	fs/...:parse...ket	if (*new...)	kmem...(...)	O
19	drivers/...:fault...te	if (co...)	kfree(data)	O	79	fs/...:ecryp...ging	if (!ecr...)	kfree(ecr...)	O
20	drivers/...:fault...te	if (un...)	kfree(data)	O	80	fs/...:ker...file	if (byt...)	vfree(*buf)	L
21	drivers/...:fault...d	if (un...)	kfree(data)	O	81	fs/...:jffs2...block	if (jffs2...)	kfree(su...)	O
22	drivers/...:mlx4...fs	if (!tu...)	kfree(tun...)	O	82	fs/...:__br...se	if (!ctx)	kmem...(...)	O
23	drivers/...:mlx4...fs	if (ib_...)	kfree(tun...)	O	83	fs/...:_nfs...copy	if (hand...)	kfree(res...)	O
24	drivers/...:srp...in	if (!po...)	kfree(addr)	L	84	fs/...:_nfs42...py	if (pro...)	kfree(res...)	O
25	drivers/...:led...set	if (trig...)	kfree(event)	O	85	fs/...:nfs4...tion	if (nfs4...)	__free...(...)	O
26	drivers/...:led...set	if (dev...)	kfree(event)	O	86	fs/...:nfs4...tion	if (nfs4...)	kfree(loc...)	O
27	drivers/...:ra...r	if (rs...)	kfree(rs)	O	87	fs/...:om...map	if (blo...)	kfree(sbi...)	C
28	drivers/...:dvb...ty	if (!dv...)	kfree(dv...)	O	88	fs/...:re...mount	if (sb_...)	kfree(new...)	C
29	drivers/...:su...bs	if (!us...)	kfree(ca...)	O	89	fs/...:re...mount	if (sb_u...)	kfree(new...)	C
30	drivers/...:cx...re	if (ret...)	vfree(p_...)	O	90	fs/...:re...mount	if (!sb...)	kfree(new...)	C
31	drivers/...:cx...re	if (ret...)	vfree(p_...)	O	91	fs/...:__ub...ac	if (err)	kfree(hmac)	O
32	drivers/...:cx...re	if (i2c...)	kfree(buf)	O	92	fs/...:read_znode	if (ubifs...)	kfree(idx)	O
33	drivers/...:dib...on	if (rx0...)	kfree(rx)	O	93	kernel/...:p...t	if (ret...)	kfree(path)	L
34	drivers/...:dib...on	if (rx0...)	kfree(tx)	O	94	kernel/...:tr...te	if (!p...)	kfree(par...)	O
35	drivers/...:dib...on	if (rx1...)	kfree(rx)	O	95	kernel/...:tr...te	if (!pid...)	kfree(par...)	O
36	drivers/...:dib...on	if (rx1...)	kfree(tx)	O	96	lib/...:test...init	if (__te...)	kfree(test...)	O
37	drivers/...:h...ch	if (saa...)	kfree(he...)	O	97	net/...:com...ace	if (WA...)	vfree(ent...)	O
38	drivers/...:f...n	if (ctx...)	kfree(ctx)	O	98	net/...:eth...stats	if (ret < 0)	vfree(data)	L
39	drivers/...:vp...up	if (ma...)	kfree(buf...)	O	99	net/...:bpf...filter	if (!new...)	kfree(addrs)	C
40	drivers/...:vp...up	if (sub...)	kfree(buf...)	O	100	sound/...:iso...it	if (WA...)	kfree(b->...)	O
41	drivers/...:vp...up	if (ti...)	kfree(buf...)	O	101	sound/...:sn...dec	if (snd_...)	kfree(cod...)	O
42	drivers/...:sm...ne	if (sm...)	kfree(zo...)	O	102	sound/...:snd...ed	if (w->r...)	kfree(w)	O
43	drivers/...:na...t	if (ch...)	vfree(buf)	O	103	sound/...:snd...ed	if (w->p...)	kfree(w)	O
44	drivers/...:spi...it	if (spi...)	kfree(dw...)	O	104	sound/...:snd...ed	if (w->clk)	kfree(w)	O
45	drivers/...:o...q	if (oct...)	vfree(oct...)	O	105	sound/...:soc...te	if (se==...)	kfree(priv..)	L
46	drivers/...:bl...te	if (bit...)	kvfree(t)	O	106	sound/...:soc...te	if (se==...)	kfree(name)	L
47	drivers/...:ix...32	if (ix...)	kfree(mask)	O	107	sound/...:soc...te	if (se==...)	kfree(dval...)	L

Table 2: (Continued)

48	drivers/...:ix...32	if (ix...)	kfree(input)	O	108	sound/...:soc...te	if (se==...)	kfree(dtex...)	L
49	drivers/...:ix...32	if (ix...)	kfree(jump)	O	109	sound/...:soc...te	if (strnle...)	kfree(priv...)	L
50	drivers/...:ml...er	if (ml...)	kfree(rss...)	O	110	sound/...:soc...te	if (strnle...)	kfree(name)	L
51	drivers/...:m...be	if (st...)	kfree(mg...)	O	111	sound/...:soc...te	if (strnle...)	kfree(dval...)	L
52	drivers/...:q...start	if (qe...)	kfree(p_...)	O	112	sound/...:soc...te	if (strnle...)	kfree(dtex...)	L
53	drivers/...:q...start	if (qe...)	vfree(p_...)	O	113	sound/...:sou...it	if (index...)	kfree(s)	O
54	drivers/...:q...info	if (rc)	kfree(cd...)	C	114	sound/...:hif...init	if (hif...)	kfree(rt)	O
55	drivers/...:cx...nd	if (!time)	kfree(voi...)	O	115	sound/...:hif...init	if (hif...)	kfree(buffer)	O
56	drivers/...:k...dr	if (kal...)	kfree(us...)	O	116	sound/...:hif...init	if (snd_...)	kfree(buffer)	O
57	drivers/...:k...dr	if (kal...)	kfree(us...)	O	117	sound/...:par...unit	if (!kctl)	kfree(nam...)	O
58	drivers/...:l...be	if (reg...)	kfree(buf)	O	118	sound/...:add...ctl	if (err < 0)	kfree(elem)	C
59	drivers/...:l...be	if (lan...)	kfree(buf)	O	119	sound/...:add...ctl	if (err < 0)	kfree(elem)	C
60	drivers/...:i24...init	if (ss...)	kfree(op...)	O	120	sound/...:snd...c3	if (!pd)	kfree(fp->...)	O

Comparing with Existing Leak Detectors. A further study of the memory leaks reported by MLEE shows that many of the leaks actually cannot be detected using existing static detection approaches due to their fundamental limitation. We use the Bug#50 in Table 2 as an example to explain this limitation. Figure 9 shows the source code of this bug. Here, the memory object `rss_map->indir_qp` is not deallocated when `mlx4_qp_alloc()` fails. MLEE successfully reports this memory leak by figuring out that the memory deallocation at line 11 is missed on the E-E path of the E-E branch at line 6.

However, most of existing static detectors are established based on a basic assumption that a memory object should be deallocated at the end of the kernel routine, in which it is allocated, if it does *not* escape from this routine. This assumption itself has no problem, but it cannot cover memory leaks involving *escaped* objects, which are fairly common in OS kernels. Obviously, in this example, `rss_map->indir_qp` is still alive after the routine is returned. Hence, it is extremely hard for existing detectors to reason about whether `rss_map->indir_qp` should be deallocated or not for the buggy E-E path.

False Negatives. For a bug detection tool like MLEE, it is important to study false negatives of the detection results. However, given the large code base of the Linux kernel, it is hard to study false negatives of the detection results of MLEE at the scale of the entire kernel, because of the lack of the ground truth. Nevertheless, during the implementation and evaluation of MLEE, we conducted several false negative experiments to understand the completeness of the static analysis techniques developed in MLEE. For example, we manually examined many kernel routines to verify whether MLEE can identify all E-E paths in these routines, especially when the E-E paths are used in different usage scenarios. In addition, we intentionally modified several randomly selected source files in the kernel to introduce some known memory leak bugs by commenting out memory deallocations on related E-E paths and checked whether MLEE is able to detect them. Our experimental results showed that MLEE can successfully identify all E-E paths and report the injected memory leaks without any false negative. This demonstrates, to some extent, the completeness of the analysis techniques in MLEE on detecting memory leaks on E-E paths in Linux.

False Positives. As a static leak detector, MLEE inevitably suffers from false positives. Among the reported 126 suspicious E-E branches, 23 (18%) are confirmed having no memory leaks. Generally, a false positive is reported by MLEE due to two categories of reasons. First, the missed memory deallocation is required but invoked after the E-E path is completed, i.e., the current kernel routine is returned. Second, the missed memory deallocation is not required on the E-E path due to various corner cases not handled by MLEE.

We summarize the detailed reasons in each category and the associated percentages in Table 3. The most accountable reason is #5, where the target E-E path is composed for a different semantic purpose compared to the path(s) on which the missed deallocation is found. For example, a memory deallocation may be necessary on E-E paths that handle exceptional kernel statuses but unnecessary for E-E paths that handle expected statuses. To filter out these false positives, we can extend MLEE to capture the semantic differences between paths in the same routine. For other false positive reasons, e.g., #2, we need to develop scalable techniques to continue the tracking of the path after the E-E path is completed.

Analysis Efficiency. The evaluation platform is equipped with an Quad-Core Intel Xeon E5-1620 v4 CPU at 3.50 GHz and 32 GB main memory. The OS is Ubuntu 16.04 with Linux-4.4. The platform was exclusively occupied by MLEE during the analysis process. It took around half an hour for MLEE to complete the analysis of the Linux kernel. Given that the Linux kernel contains 25M lines of source code, we believe this analysis performance is acceptable.

Table 3: Detailed reasons of false positives reported by MLEE and the percentage of each reason.

Required	1	The memory object is deallocated by another kernel thread after the E-E path.	8%
	2	The memory object is deallocated by the same kernel thread in a callback routine after the E-E path.	18%
Not Required	3	The missed deallocation itself is redundant as the allocated object can be reused.	8%
	4	The E-E path does not satisfy a specific condition.	13%
	5	The E-E path is not used for the same semantic purpose as the path(s) of the missed deallocation.	53%

7 Related Work

Static Leak Detection. Static detection approaches analyze programs to discover memory leaks without running the programs. Clouseau [15] identifies program statements inconsistent with an ownership model as memory leaks. Hackett and Rugina [13] propose a region-based shape analysis to detect memory leaks. Xie et al. [44] represents computations as boolean constraints for memory leak detection. Orlovich et al. [31] disprove the presence of memory leaks through a backward data-flow analysis. FastCheck [7] reduces memory leak detection to a reachability problem over a guarded graph. LeakChecker [46] identifies memory leaks through a common code pattern of memory leaks. SMOKE [9] develops the use-flow graph to detect leaks in two stages.

MLEE is also a static memory leak detector. But, MLEE is inspired by the observations and findings from our study of memory management on E-E paths in OS kernels. This enables MLEE to invent effective and scalable static analyses to detect memory leaks on E-E paths. The detection results show that MLEE is a practical leak detector. In contrast, it is quite difficult to directly apply most of existing static mechanisms to OS kernels, due to the extreme complexity of the kernel code base and the widespread use of E-E paths in various scenarios. Although Xie et al. [44] applied the proposed leak detector to the Linux kernel, it took around one day to complete the analysis and, among the reported 123 leaks, only 2 were actually confirmed and fixed. Moreover, the code base of the Linux kernel has expanded significantly since then.

Dynamic Leak Detection. Dynamic detectors identify memory leaks by running the programs. Purify [14] adapts garbage collection techniques to detect memory leaks in C/C++ programs. SWAT [8] and Sleigh [4] identify a *stale* object as a memory leak if it has not been accessed for a long time. Cork [22] uses a dynamic heap-summarization technique to detect memory leaks. Rayside et al. [32] leverage object ownership profiling to find instances of memory management anti-patterns in object-oriented programs. Xu et al. [45] propose a heap-tracking technique to detect leaks in Java programs. Hound [30] organizes the heap layout to facilitate the staleness tracking of heap objects for leak detection. Li et al. [26] develop a dynamic technique to validate and categorize memory leak warnings reported by static detectors. GC assertions [3] allow developers to diagnose leaks through a system interface. Maxwell et al. [28] apply a graph grammar mining approach to detect leaks. Sniper [23] utilizes hardware performance monitoring units to track object staleness for leak detection. Lee et al. [25] train a machine learning model during the software testing stage and apply the model to the production stage for leak detection. MemInsight [20] performs a lifetime analysis on objects to detect leaks in JavaScript programs.

Compared to static detectors, dynamic approaches suffer from the coverage issue, i.e., only the exercised paths are examined. How to generate sufficient test inputs to cover rarely-executed paths, e.g., E-E paths, is an interesting but hard problem. Also, dynamic detectors deployed in production runs may incur performance overhead. MLEE may utilize dynamic techniques to further enhance the detection accuracy.

Others. Some research work aims to automatically fix or tolerate memory leaks. LeakSurvivor [34] and Melt [5] swap out leaked objects to disks to free up memory resources. LeakFix [10] checks each allocation and inserts a deallocation if necessary. BLeak [38] provides a leak debugging framework for web applications. Generally, MLEE can collaborate with them to provide integrated solutions for memory leaks.

8 Conclusion

Memory leaks can lead to critical performance and security issues, especially in OS kernels. Inspired by the observation that memory leaks often lurk in rarely-tested program paths, e.g., E-E paths, we firstly conduct a comprehensive and in-depth study on memory management operations involved on E-E paths in OS kernels. With the findings derived from the study, we then design MLEE, which aims to intelligently detect memory leaks on E-E paths. MLEE employs novel static analysis techniques to automatically identify E-E paths and infer whether a missed memory deallocation is required on an E-E path. The static analyses in MLEE are effective and scalable. With the help of MLEE, we discover 120 new memory leaks in the Linux kernel and most of these memory leaks have been fixed using our patches.

Acknowledgments

We are very grateful to our shepherd, Eric Schkufza, and the anonymous reviewers for their valuable feedback and comments. This work is supported in part by a faculty startup funding of the University of Georgia.

References

[1] The freebsd project. https://www.freebsd.org.

[2] The linux kernel archive. https://www.kernel.org.

[3] Edward E. Aftandilian and Samuel Z. Guyer. Gc assertions: Using the garbage collector to check heap properties. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 235–244, New York, NY, USA, 2009. ACM.

[4] Michael D. Bond and Kathryn S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 61–72, New York, NY, USA, 2006. ACM.

[5] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 109–126, New York, NY, USA, 2008. ACM.

[6] MySQL bug #83047. Memory usage gradually increases and brings server to halt, 2016. https://bugs.mysql.com/bug.php?id=83047.

[7] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 480–491, New York, NY, USA, 2007. ACM.

[8] Trishul M. Chilimbi and Matthias Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 156–164, New York, NY, USA, 2004. ACM.

[9] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 72–82, Piscataway, NJ, USA, 2019. IEEE Press.

[10] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 459–470, Piscataway, NJ, USA, 2015. IEEE Press.

[11] GCC. Labels as values. https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html.

[12] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. Eio: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, USA, 2008. USENIX Association.

[13] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 310–323, New York, NY, USA, 2005. ACM.

[14] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[15] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 168–181, New York, NY, USA, 2003. ACM.

[16] David L. Heine and Monica S. Lam. Static detection of leaks in polymorphic containers. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 252–261, New York, NY, USA, 2006. ACM.

[17] Jian Huang, Michael Allen-Bond, and Xuechen Zhang. Pallas: Semantic-aware checking for finding deep bugs in fast path. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 709–722, New York, NY, USA, 2017. ACM.

[18] Chromium issue 816002. High memory usage in youtube, 2018. https://bugs.chromium.org/p/chromium/issues/detail?id=816002.

[19] Suman Jana, Yuan Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 345–362, USA, 2016. USENIX Association.

[20] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. Meminsight: Platform-independent memory debugging for javascript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 345–356, New York, NY, USA, 2015. ACM.

[21] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, Ji Wang, Xiaodong Liu, and Yunhuai Liu. Detecting error-handling bugs without error specification input. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 213–225. IEEE Press, 2019.

[22] Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 31–38, New York, NY, USA, 2007. ACM.

[23] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 825–836, New York, NY, USA, 2014. ACM.

[24] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[25] Sangho Lee, Changhee Jung, and Santosh Pande. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 814–824, New York, NY, USA, 2014. ACM.

[26] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. Dynamically validating static memory leak warnings. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 112–122, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 911–922, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 115–124, New York, NY, USA, 2010. ACM.

[29] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 577–587, New York, NY, USA, 2014. ACM.

[30] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 397–407, New York, NY, USA, 2009. ACM.

[31] Maksim Orlovich and Radu Rugina. Memory leak analysis by contradiction. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 405–424. Springer-Verlag, 2006.

[32] Derek Rayside and Lucy 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*, ASE '07, pages 194–203, New York, NY, USA, 2007. ACM.

[33] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[34] Yan Tang, Qi Gao, and Feng Qin. Leaksurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 307–320, Berkeley, CA, USA, 2008. USENIX Association.

[35] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 752–762, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[37] Nginx ticket #1482. Memory leak in error handling block in ngx_stream_geo_block method, 2018. `https://trac.nginx.org/nginx/ticket/1482`.

[38] John Vilk and Emery D. Berger. Bleak: Automatically debugging memory leaks in web applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 15–29, New York, NY, USA, 2018. ACM.

[39] Common Vulnerabilities and Exposures. Cve-2019-12379, 2019. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12379`.

[40] Common Vulnerabilities and Exposures. Cve-2019-8980, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8980.

[41] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Enhancing cross-isa dbt through automatically learned translation rules. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 84–97, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, Hollywood, CA, 2012. USENIX.

[43] Mingyuan Xia, Wenbo He, Xue Liu, and Jie Liu. Why application errors drain battery easily?: A study of memory leaks in smartphone apps. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 2:1–2:5, New York, NY, USA, 2013. ACM.

[44] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '05, pages 115–125, New York, NY, USA, 2005. ACM.

[45] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 151–160, New York, NY, USA, 2008. ACM.

[46] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. Leakchecker: Practical static memory leak detection for managed languages. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 87:87–87:97, New York, NY, USA, 2014. ACM.