Gen Zhang, Peng-Fei Wang, Xu Zhou *et al.* MEBS: Uncovering Memory Life-cycle Bugs in Operating System Kernels. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY, August 2021. This is a pre-print of an article published in JCST. The final authenticated version is available online at: DOI: 10.1007/s11390-021-1593-4.

MEBS: Uncovering Memory Life-cycle Bugs in Operating System Kernels

Gen Zhang[1,], Peng-Fei Wang[1], Tai Yue[1], Xu Zhou[1], and Kai Lu[1], *Member, CCF*

[1] *College of Computer, National University of Defense Technology, Changsha 410073, China*

E-mail: {zhanggen, pfwang, yuetai17, zhouxu, kailu}@nudt.edu.cn
Received ; revised .

Abstract Allocation, dereferencing, and freeing of memory data in kernels are coherently linked. Real cases widely exist where the correctness of memory is compromised. This incorrectness in kernel memory brings about significant security issues, e.g., information leaking. Though memory allocation, dereferencing, and freeing are closely related, previous work failed to realize this point. In this paper, we study the life-cycle of kernel memory, which consists of allocation, dereferencing, and freeing. Errors in them are called memory life-cycle (MLC) bugs. We propose an in-depth study of MLC bugs and implement a memory life-cycle bug sanitizer (MEBS) for MLC bug detection. Utilizing an inter-procedural global call graph and novel identification approaches, MEBS can reveal memory allocation, dereferencing, and freeing sites in kernels. By constructing a modified define-use chain and examining the errors in the life-cycle, MLC bugs can be identified. Moreover, the experiment results on the latest kernels demonstrate that MEBS can effectively detect MLC bugs, and MEBS can be scaled to different kernels. More than 100 new bugs are exposed in Linux and FreeBSD, and 12 common vulnerabilities and exposures (CVE) are assigned. The prototype of MEBS will be released to contribute to the research in this field.

Keywords software security, operating systems, memory life-cycle, static analysis, vulnerability detection

1 Introduction

Linux kernels have numerous functions for heap memory allocation, e.g., `kmalloc`(). A memory allocation return value (MARV) is the memory pointer returned by a memory allocation function. Generally speaking, a MARV is allocated by a source function, then dereferenced, and at last freed by a sink function.

Therefore, we can summarize allocation, dereferencing, and freeing as the life-cycle of kernel memory. These three steps are closely linked and form an inseparable entity. This inseparability comes in three aspects, including allocation-dereferencing, dereferencing-freeing, and allocation-freeing. The relationship between allocation and dereferencing is apparent. Only correctly allocated memory can be dereferenced, and before dereferencing the memory, allocation failure should be checked. Furthermore, the freeing operation is also related to allocation and dereferencing. After dereferenced, the allocated memory should be freed with the correct freeing function corresponding to the allocation function. By thoroughly considering the relationships between the inseparable life-cycle, we introduce memory life-cycle (MLC) bugs.

Regular Paper

This work is supported by National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), Natural Science Foundation of China (61902405), Natural Science Foundation of China (61902412), PDL Research Foundation 6142110190404, and the Research Project of National University of Defense Technology (ZK20-17).

©Institute of Computing Technology, Chinese Academy of Sciences & Springer Nature Singapore Pte Ltd. 2020

```
 1    mnt_opts = selinux_alloc_mnt_opts(len);
 2    ...
 3    val = kstrdup_const(s, GFP_KERNEL);
 4    /* unchecked dereferencing */
 5 +  if (!val) {
 6      /* lacking freeing */
 7 +    selinux_free_mnt_opts(*mnt_opts);
 8 +    return -ENOMEM;
 9 +  }
10    /* incorrect allocation-freeing */
11 -  kfree(val);
12 +  kfree_const(val);
13    selinux_free_mnt_opts(*mnt_opts);
14    ...
15    /* use-after-free */
16 -  rc = selinux_add_opt(token, val, mnt_opts);
```

Fig.1. Example patch file.

To begin with, we use an example to show MLC bugs. In Fig. 1, `val` is improperly freed by `kfree`(). The sink function should be `kfree_const`() according to the documents. This is an issue in the allocation-freeing relationship, where a sink function does not match the source function. Furthermore, the patch enforces a check for `val` after the allocation operation, which is related to the allocation-dereferencing relationship, where verification of the MARV is required. Moreover, `mnt_opts` is freed in the patch to prevent potential memory leaking. In the original code, we can see that the dereferencing-freeing relationship is ignored in memory life-cycle, in which a freeing operation is missing after the dereferencing operation. In addition, it is not allowed to use `mnt_opts` after being freed to prevent use-after-free bugs. Therefore, line 16 is eliminated.

We call these bugs MLC bugs, including five types of bugs: incorrect allocation-freeing (IAF), unchecked dereferencing (UD), lacking freeing (LF), use-after-free (UAF), and double-free (DF). If the allocation and freeing functions are used incorrectly, an incorrect allocation-freeing case is matched. An unchecked dereferencing bug happens when a MARV is dereferenced without a null check. In addition, when a MARV is unfreed, we can identify a lacking freeing bug. When we dereference a MARV after freeing it, a use-after-free bug is going to happen. A double-free bug is triggered when a freed MARV is freed again.

The reason why we concentrate on these bugs is as follows: 1) MLC bugs are common in kernels because of commonly seen memory allocations in operating system (OS) kernels. For instance, there are 35,359 allocation operations in Linux according to our experiments. More operations indicate a higher possibility to trigger MLC bugs. Second, MLC bugs are common in kernels because all memory space goes through allocation, dereferencing, and freeing. Any error in the life-cycle results in MLC bugs. 2) MLC bugs can cause severe consequences and security problems, such as denial-of-service and information leakage. For instance, we have 12 common vulnerabilities and exposures (CVE) assigned based on the detected MLC bugs. Among them, we get high common vulnerability scoring system (CVSS) [1] scores. The scores indicate high security risks.

Identifying MLC bugs can be challenging. First, consisting of more than 27 million lines of code and supporting various architectures, the Linux kernel is certainly a complex software system. Analyzing kernels demands customized techniques to handle unforeseen circumstances. Second, to detect MLC bugs, we need a clear and specific definition. Previous work on memory errors [1–4] cannot be applied in our situation. Detecting MLC bugs requires precise reconstruction of the allocation, dereferencing, and freeing sites of the life-cycle. Detecting MLC bugs is more complicated than detecting common memory errors. Moreover, there are challenges in implementing the related rules, e.g., recognizing the customized source functions. There is no available approach to identifying a customized source function in kernels.

To overcome the challenges mentioned above and

[1]Common vulnerability scoring system. https://www.cvedetails.com/, Aug. 2021

detect MLC bugs, we first improve the concept of memory life-cycle proposed by Zhang [5]. Based on the concept, a formal definition of life-cycle-related rules and MLC bugs is given. In addition, we implement a memory life-cycle bug sanitizer (MEBS) to expose MLC bugs. To begin with, MEBS takes LLVM (low level virtual machine) IR (intermediate representation) of kernel source code as the input. Then, we construct a global call graph and conduct pointer analysis. Next, all the source functions, sink functions, and MARVs in OS kernels are identified by novel analysis techniques. Moreover, MEBS constructs modified define-use chains of the MARVs. By examining the chains, MLC bugs are detected.

In addition, we perform evaluations on Linux and FreeBSD. The entire analysis is finished in several minutes, and more than 100 new MLC bugs are exposed. Patches are submitted to the maintainers, and 12 CVEs are assigned. The experiment results demonstrate that MLC bugs widely exist in kernels, and we should focus on them to prevent security issues. Furthermore, the results demonstrate that MEBS can effectively detect MLC bugs.

The most critical point in MEBS is that we treat allocation, dereferencing, and freeing as an indivisible entity. This entity contains relationships between allocation-dereferencing, dereferencing-freeing, and allocation-freeing. Therefore, we can thoroughly and completely examine the relationships, and any incomplete or absent inspection is treated as breaking the indivisibility. Generally speaking, our work in MLC bugs is distinguishable from the related work in API misuse [6–9], missing check [10–12], and memory management model checkers [13–15]. For example, as a memory checker, MCChecker [13] checked allocation, dereferencing, and freeing of the kernel memory. However, compared with MEBS, MCChecker failed to follow the inseparability of memory life-cycle in the following aspects. First, MCChecker only checked commonly used functions, e.g, `kmalloc`(). Consequently, incomplete identification of source functions and sink functions did not ensure that MCChecker covered all the above-mentioned relationships. Second, MCChecker only considered the dereferencing-freeing relationship in the error paths, which left the non-error paths unchecked. Third, MCChecker ignored the allocation-freeing relationship. In conclusion, due to either insufficiency or absence of examination of the relationships, MCChecker did not maintain the inseparability of memory life-cycle. On the contrary, MEBS solves these problems by identifying all the source functions and sink functions, completely checking the paths on the define-use chain, and thoroughly examining the relationships. Experiments in Section 5 demonstrate that MEBS can identify more bugs than MCChecker. The results prove the importance of the life-cycle inseparability.

In conclusion, we make the following contributions in this paper:

- We propose a new perspective of memory life-cycle. This paper proposes a lofty vision, that memory bugs need to be reasoned about holistically concerning the life-cycle of memory. In addition, we improve the MLC bug definition, including five sub-classes. The most critical point in MEBS compared with other tools is that we treat allocation, dereferencing, and freeing as an indivisible entity. Therefore, we can thoroughly and completely examine the life-cycle, and any incomplete or absent inspection is treated as breaking the indivisibility.

- We propose new techniques to detect MLC bugs. We perform inter-procedural call graph construc-

tion and pointer analysis. Most importantly, we propose novel methods to detect the source functions, sink functions, and MARVs. Compared with previous methods, our identification techniques have higher precision. In addition, we build define-use chains based on the identified functions and MARVs to find MLC bugs. Compared with precious approaches, our bug detection methods are both effective and scalable.

- We detect 168 MLC bugs and 12 CVEs in Linux and FreeBSD. These bugs are capable of compromising the entire system and causing security problems. None of the detected bugs was discovered by other tools before.

In the rest of this paper, we discuss about the background of MLC bugs in Section 2. The design of MEBS is in Section 3. Section 4 contains the implementation details. The experiment results are in Section 5. Section 6 is the discussion about MEBS. Section 7 contains the related work. Section 8 concludes this paper.

2 MLC Bugs

2.1 Memory Life-cycle

Zhang first proposed the concept of MLC bug in MLCSan [5]. Following the concept of MLCSan, we introduce the life-cycle of a MARV as follows:

- Allocation. A MARV is allocated by a source function.
- Dereferencing. Before using this MARV, a null check should be enforced. Next, the MARV is dereferenced.
- Freeing. Finally, MARV is freed with the correct sink function. Any using or freeing is not allowed after the freeing site.

2.2 Customized Source Functions and Sink Functions in Kernels

Programmers use source functions to allocate kernel memory and use sink functions to free the memory. Besides the well known `kmalloc`()-`kfree`(), there are numerous customized source functions and sink functions. For example, direct memory access (DMA) allows an input-output (IO) device to send or receive data to the main memory directly. `dma_pool_alloc`() gets a block of consistent memory for DMA, and `dma_pool_free`() frees the block back into the DMA pool. In this case, `dma_pool_alloc`() and `dma_pool_free`() can be called customized source function and sink function, respectively.

Table 1. List of Features of Source Functions

Feature	Description
Pointer type	The return value of a source function should be a pointer value pointing to the memory space.
New pointer	A source function usually returns a pointer value, and this assignment should be the first operation on this pointer after the initial declaration.
Null check	After the allocation of the pointer, it should be checked to prevent allocation failure.
Memory access	After the null check, the pointer is used to access memory (memory dereferencing).

2.2.1 Source Functions

For the customized source functions in OS kernels, we study the common usage of them and extract the most frequent features in Table 1. There are four features listed in the table, and most of the source functions in the kernel fit these features. Our analysis utilizes these features to identify the customized kernel source functions in Section 3.

```
1  A = alloc1();
2  if (!A)
3    return -ENOMEM;
4  ...
5  A->B = alloc2();
6  if (!A->B)
7    goto error;
8  ...
9  error:
10   free1(A);
11   return -ENOMEM;
```

Fig.2. Error path example.

2.2.2 Sink Functions

OS kernels contain millions of error paths and security checks [16]. As shown in Fig. 2, `A` is allocated first, and then `A` $\rightarrow$ `B` is allocated. When the null check for `A` $\rightarrow$ `B` fails, `A` should be freed with the corresponding sink function in the error path. By identifying the error paths in kernels, we can extract all the customized sink functions located in the error paths.

2.2.3 Source-sink Pairs

In addition, by traversing the program paths in a backward direction from the freeing site to the allocation site, source-sink pairs can be identified. These pairs indicate the correct usage of the customized source functions and sink functions.

3 Design of MEBS

3.1 Overview

Fig. 3 shows the primary workflow, including three key stages: preprocessing, analysis, and bug reporting.

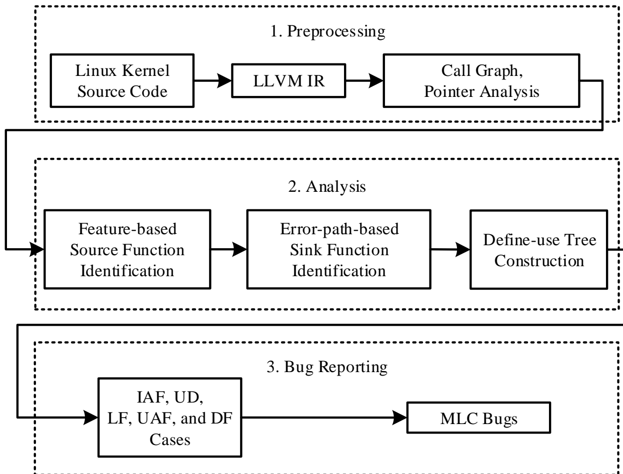


Fig.3. Primary workflow of MEBS.

1) A global call graph is built in Stage 1. In addition, we perform pointer analysis to help the following analysis, e.g., alias analysis. 2) Stage 2 contains source function identification, sink function identification, and define-use chain construction. 3) MLC bugs are reported in Stage 3.

3.2 Preprocessing

3.2.1 Call Graph Construction

To conduct inter-procedural analysis, a global call graph is built, including direct and indirect calls between functions.

The call graph is used in MEBS to detect MLC bugs across multiple procedures or functions. For example, we want to identify whether a variable is used to access memory. The variable is allocated and used in different functions. In this situation, the call graph may tell us one function calls another, and we can decide this variable is allocated and then used to access memory in these two functions.

```
1  void *true_alloc(args) ...
2  void *false_alloc(args) ...
3  struct true_struct T1 = {
4    .ind_alloc = true_alloc; /* true target */
5  };
6  struct false_struct T2 = {
7    .ind_alloc = false_alloc; /* false target */
8  };
9  struct true_struct indirect ...
10 indirect->ind_alloc(args);/* indirect call site */
```

Fig.4. Indirect call example.

3.2.2 Indirect Call Analysis

We incorporate the argument information with the structure data type (struct for short) containing address-taken functions to solve indirect call analysis. The struct is common in OS kernels. Lu et al. [12] claimed over 85% address-taken functions in Linux were initially stored in a pointer field of a struct. Therefore, we can use the type information of the struct with the argument information to increase the accuracy of indirect call analysis. Fig. 4 shows a general situation, where we want to find the target of `ind_alloc(args)` in line 10. If we only match the argument information `args`, we detect two targets: `true_alloc` in line 4 and

`false_alloc` in line 7. However, when we use the struct type of `indirect` in line 10 to match the result, we can discover that `true_alloc` is the target because the type of `indirect` is `true_struct`, which is the same as the type of `T1` in line 3.

Our approach is to match the struct type at the indirect call site with the struct type at the initialization site. A match can be further verified with the argument number and type to confirm the targets. When the struct type does not match anyone, we use only the argument number and type to identify the targets, and address-taken functions not initialized in the struct field can be analyzed in this way.

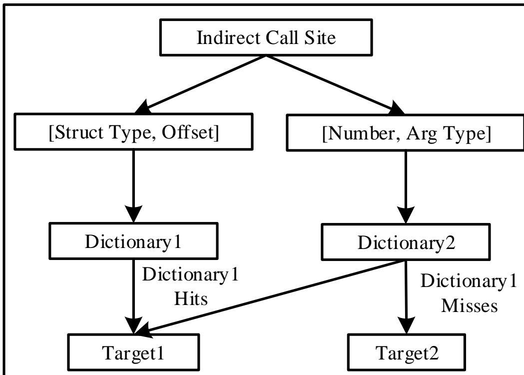


Fig.5. Indirect call analysis in MEBS.

In the beginning, we traverse the kernel to identify initialization sites and function declarations to construct two two-key dictionaries. The key of the first dictionary contains the struct type, offset of the field, and the key of the second one is the argument number with type. In general, we can collect [struct type, offset] as the key of the first dictionary and [num, arg type] as the key of the second one. The values of the dictionaries are the targets of indirect calls. Next, when an indirect call site is met, we extract the struct type and the GEP (GetElementPtr in LLVM, for calculating pointer address) instruction offset to look up in the first dictionary. If there is a value matching this key ([struct type, offset]), the argument information ([num, arg type]) is used in the second dictionary to confirm the result. Otherwise, Dictionary2 directly identifies the targets with the argument information. Fig. 5 illustrates the procedure of our indirect call analysis with the two dictionaries.

3.2.3 Pointer Analysis

Pointer analysis is conducted to detect the aliases of MARVs. It is fundamental in identifying MLC bugs because one MARV may propagate to other variables. These variables should also be tracked to reveal all the operations on the MARV. MEBS provides the alias results through the following steps. 1) First, we use the built-in alias analysis pass (AliasAnalysis) in LLVM. This pass provides four types of alias results, including “Must”, “Partial”, “May” and “No”. To reduce inaccuracy, we consider “Must” and “Partial” as aliased, “May” and “No” as not-aliased. 2) In addition, we consider the performance overhead of pointer analysis. Our initial version of the implementation shows that a few objects with many aliases mainly cause the overhead, and we solve this problem by limiting the number of aliases of these particular variables. Generally speaking, this limitation only affects less than 50 objects when the limit is 1,000, and the performance overhead is reduced to an acceptable level.

Algorithm 1 Analysis and Bug Reporting

```
1: function SOURCE_IDENTIFICATION()
2:     {SC} = ∅ /* Source functions */
3:     f_t = 0
4:     f_s = 0
5:     for F in Kernel do
6:         f_t = f_t + 1
7:         if features match then
8:             f_s = f_s + 1
9:         end if
10:    end for
11:    for F in Kernel do
12:        if Score(f_s, f_s/f_t) >= Threshold then
13:            {SC} =F∪{SC}
14:        end if
15:    end for
16: end function
17: function SINK_IDENTIFICATION(ErrorPaths)
18:    {SK} = ∅ /* Sink functions */
19:    {PA} = ∅ /* Source-sink pairs */
20:    for P in ErrorPaths do
21:        if struct pointer allocated by F1 in {SC} And freed by F2 then
22:            {SK} =F2∪{SK}
23:            {PA} =(F1, F2)∪{PA}
24:        end if
25:    end for
26: end function
27: function MARV_PROPAGATION({SC}, AliasResult)
28:    {MARV} = ∅
29:    for F in {SC} do
30:        if F returns V then
31:            {MARV} = {V∪alias(V)}∪{MARV}
32:        end if
33:    end for
34: end function
35: function CONSTRUCT_DEFINE_USE_TREE({SC},{SK},{MARV})
36:    Output: {DUT}
37: end function
38: function BUG_REPORTING({MARV}, {DUT}, {PA})
39:    Output: {IAF}, {UD}, {LF}, {UAF}, {DF}
40: end function
```

3.3 Feature-based Source Function Identification

There are numerous customized source functions in OS kernels, e.g., `dma_pool_alloc()`. They are essential components of kernel sub-systems for customized memory allocation. Identifying them is critical not only for MLC bug identification but also for deeply understanding memory allocation in the kernel. However, there is no existing approach to accomplishing this task effectively. For example, we cannot use natural language processing to identify source functions because there is no kernel document describing them in detail. To solve this problem, we manually study the known customized source functions in kernel sub-systems. Then, we extract the most frequent and representative features of the source functions in Table 1. The features can describe the usage of a source function precisely.

In this step, we use these four features and statistical methods to identify the customized source functions in OS kernels. For every call site of a kernel function, we examine its behaviors to match the four features. We track the operations on the return value to identify whether it is a new pointer. Forward flow analysis is then conducted to detect the null check and memory access of this return value. If this call site of the function matches the features, we treat it as a source function behavior.

Next, we utilize some statistical approaches to further improving the precision of source function identification. When a certain function F is called in the kernel, we record the number of call sites of F (total frequency f_t) and the number of source function behaviors among these call sites (source function frequency f_s). After traversing the whole kernel, we have the source function frequency f_s and the frequency rate $\frac{f_s}{f_t}$ of every kernel function. The idea of calculating both f_s and $\frac{f_s}{f_t}$ is reasonable. For example, F occurs only once in the kernel, and this occurrence is incorrectly identified as a source function behavior due to unavoidable false positives (FP). In this case, the f_s of F is one and the $\frac{f_s}{f_t}$ is 100%. However, this high frequency rate cannot be directly used to treat F as a source function.

By assigning the function a score based on f_s with $\frac{f_s}{f_t}$ and eliminating candidates below a certain threshold, we can effectively improve the precision and reduce the false positives. The full score is 100, and we give portioned weights to f_s and $\frac{f_s}{f_t}$. The simplified equation is $Score = w_1 \times f_s + w_2 \times \frac{f_s}{f_t}$, where w_1 and w_2 are the weights. We thoroughly discuss the weights and the values of thresholds in Section 5. Lines 1 - 16 in Algorithm 1 show the procedure of feature-based source function identification.

3.4 Error-path-based Sink Function Identification

Sink functions are used to free memory. Identifying them is a prerequisite to detect kernel memory leaking, use-after-free, and double-free bugs. However, customized sink functions are also not sufficiently studied yet. By observing the kernel error paths, we find out that they are ideal origins for sink function detection. First, error paths often contain the sink functions in the error handling code. Next, error paths widely exist in OS kernels (2 million in Linux), and this is sufficient for statistical methods.

As shown in Fig. 2, freeing a struct pointer containing another pointer in the error path is commonly seen in kernels. Therefore, we can extract numerous sink functions in the millions of error paths in OS kernels. This method is born to have high detection precision because the rules are more direct compared with the feature check of source function identification, and experiment results in Section 5 also prove this observation.

We use error path results from Lu et al. [16], and the results contain CFGs (control flow graph) for every kernel function with error paths marked, including some detailed information, e.g., the branch instruction to the error path. By examining the checked value of a branch instruction, we can identify whether this value is a pointer inside a struct pointer, i.e., `A` $\rightarrow$ `B`, and whether this pointer is allocated by our identified source functions. Then, we go forwards to the error path to identify function calls operating on this struct pointer, i.e., `A`. A hit can be treated as a sink function behavior. Similar to source function identification, we record the number of hits (sink function frequency f) for every kernel function by traversing the whole kernel. By directly judging this f with a threshold, we can get the kernel sink functions with acceptable precision.

Furthermore, after identifying the sink function and the freed struct pointer, we traverse the program paths in a backward direction to locate the allocation of this pointer. In this way, we can extract the source-sink pairs, which imply the correct usage of the source functions and sink functions. However, due to the precision of backward flow analysis, some false positives may be introduced that some source functions and sink functions may be incorrectly matched. We also solve this problem with statistical methods. By recording the most frequent pair for a certain function, we obtain hundreds of source-sink pairs from the kernel, and the false positive rate is relatively low, which is shown in Section 5. For example, we can extract `alloc1()-free1()` as a source-sink pair from the error path in Fig. 2. Lines 17 - 26 in Algorithm 1 show the procedure of error-path-based sink function identification and source-sink pair identification.

3.5 Define-use Chain Construction

Algorithm 2 Define-use Tree Construction

```
Require: {SC},{SK},{MARV}
 1: function CONSTRUCT_DEFINE_USE_TREE({SC},{SK},{MARV})
 2:     {S_1} = ∅
 3:     {S_2} = ∅
 4:     for i in {MARV} do
 5:         {S_1i} = {alias(i)}∪{S_1i}
 6:         {S_2} = {S_1i}∪{S_2}
 7:     end for
 8:     {DUT} = ∅
 9:     for i in S_2 do
10:         {DUC_i} = define_use_chain(i)
11:         {DUT_i} = {DUC_i} + branches(i)+{S_1i}
12:         {DUT} = {DUT_i}∪{DUT}
13:     end for
14: end function
Ensure: {DUT}
```

3.5.1 MARV Propagation

MARVs are identified through a propagative identification technique. First, we already identify all the source functions in a set SC, and we initially consider all the return values of the source functions as MARVs. Second, since a MARV tends to propagate to other variables, we need to track all the related ones and check

the errors in our predefined life-cycle. To do this, we incorporate the alias results from Stage 1. Whenever a MARV propagates to another variable, MEBS can track and analyze it. In this propagative manner, MEBS can form a complete MARV set and detect potential issues. A MARV i and its aliases are first collected in a one-dimensional set $S_{1\mathrm{i}}$. $S_{1\mathrm{i}}$ contains all variables propagated from the same MARV i, and line 5 in Algorithm 2 shows this process. $S_{1\mathrm{i}}$ is then inserted into a two-dimensional set S_2, which is shown in line 6 in Algorithm 2. S_2 records all the identified MARVs. Adopting these $S_{1\mathrm{i}}$ sets can maintain all the alias information.

3.5.2 From Chain to Tree

The define-use chain [2] is widely used in the data-flow analysis. Given a MARV set, we should examine all the operations on every element. Our analysis is flow-sensitive because the operations on MARVs, such as checking and dereferencing, are strictly ordered. The `value.users()` approach provided by LLVM returns a set of all the using sites of a variable, which cannot be directly used in our method for its disorder. On the contrary, we build a modified chain by traversing the CFG, which extracts the results of `value.users()` in order and appends the corresponding instruction to the chain. Constructing the define-use chain is shown in line 10 in Algorithm 2.

Theoretically, we should construct a complete define-use chain for all the elements in S_2. A single-branch define-use chain cannot fit some situations. One case is the conditional branch in the program path. For different paths, examining a single-branch chain is insufficient. Because a single-branch chain cannot represent multiple program paths with branches. Another situation is that some MARVs are the aliases of others. They share the same chain before the propagation site. Creating a new single-branch chain for every alias can cause unnecessary performance overhead.

We solve the above issues by modifying a single-branch define-use chain to a multi-branch one, which is a define-use tree. When a conditional branch is hit, we add a new branch to track the remaining operations in this path. As for the second situation, a new branch is added at the propagation site to track the aliases. As shown in Fig. 6, a new branch is appended at the conditional site, and the remaining operations are correctly recorded. MARV2 is propagated from a Store instruction of MARV1. Moreover, this is where we use the $S_{1\mathrm{i}}$ set. Elements in a set are aliases, and they are added to the branches of the corresponding define-use tree. This process is shown in lines 11 - 12 in Algorithm 2.

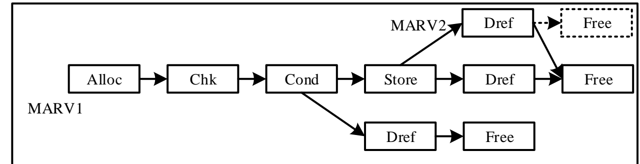


Fig.6. Branch of a define-use chain.

There is a corner case of define-use tree construction that needs to be taken care of. The conditional statements resulting in new branches are mostly error handling code. The error path and the non-error path both need to free the MARV at the end. However, we do not free its aliases when we already free a certain MARV. For example, as shown in Fig. 6, when we already identify the freeing operations of MARV1, we mark all the aliases as freed. In this figure, the end of MARV2 is joined to the end of MARV1.

[2]Wikipedia. Define-use chain. `https://en.wikipedia.org/wiki/Use-define_chain`, Aug. 2021.

3.6 MLC Bug Reporting

Algorithm 3 Bug Reporting

```
Require: {MARV}, {DUT}, {PA}
 1: {IAF} = {UD} = {LF} = {UAF} = {DF} = ∅
 2: {P1}= {PA}
 3: {P2}= pairs({DUT})
 4: for E2 in P2 do
 5:     for E1 in P1 do
 6:         if E1 not matches E2 then
 7:             {IAF} =E2∪{IAF}
 8:         end if
 9:     end for
10: end for
11: for Dref in {DUT} do
12:     if no Chk then
13:         {UD} =Dref∪{UD}
14:     end if
15: end for
16: for i in {MARV} do
17:     if no Free then
18:         {LF} =i∪{LF}
19:     end if
20: end for
21: for i in {MARV} do
22:     if Free then Dref then
23:         {UAF} =i∪{UAF}
24:     end if
25:     if Free then Free then
26:         {DF} =i∪{DF}
27:     end if
28: end for
Ensure: {IAF}, {UD}, {LF}, {UAF}, {DF}
```

In this stage, we need to report MLC bugs. According to memory life-cycle, any error in the life-cycle is reported by MEBS. First, by comparing the correct source-sink pairs P1 extracted from the error paths with P2 in the define-use tree, incorrect allocation-freeing cases can be exposed. We enumerate elements in P1 and P2, e.g., E1 and E2. If the source functions in E1 and E2 match, and the sink functions do not match, we assume an incorrect allocation-freeing bug happens, and vice versa. Reporting incorrect allocation-freeing is shown in lines 2 - 10 in Algorithm 3. Next, by examining "Chk" and "Dref" in the define-use tree, MEBS can identify cases where a MARV is dereferenced without verification. Since the operations are recorded in order, the cases where there is no "Chk" before "Dref" can be detected. This process is shown in lines 11 - 15 in Algorithm 3.

As for lacking freeing cases, MEBS traverses the define-use tree of a MARV, and if there is no sink function to free the memory, a lacking freeing case is detected. In addition, we discover that there are some common cases where lacking freeing may happen, and we add them to our detection strategy.

First, for a MARV, there is no sink function to free it. This case can be detected by checking the end of the define-use tree. Second, when an unchecked dereferencing bug happens, we check all the allocation before this site in the same function, e.g., in Fig. 1, when unchecked dereferencing of `val` happens, we detect that `mnt_opts` is allocated before this site in this function. Therefore, we identify there is a lacking freeing case and add `selinux_free_mnt_opts`() to free `mnt_opts`. Examining the CFG from the function entry to the unchecked dereferencing site can identify whether there is another allocation that needs to be freed. Lines 16 - 20 in Algorithm 3 describe the detection of lacking freeing bugs.

Furthermore, use-after-free and double-free bugs are more security-critical. By traversing forwards to examine the operations on a MARV after the freeing site, we can detect potential use-after-free and double-free bugs. If a freed MARV is dereferenced or freed again in the define-use tree, a use-after-free or double-free bug is triggered. This part of bug reporting is effective due to our high detection precision of the sink functions. Lines 21 - 28 in Algorithm 3 show the process of detecting use-after-free and double-free bugs.

4 Implementation Details

In MEBS, we utilize LLVM/Clang 9.0.1. In total, MEBS contains approximately 5,000 lines of code and six LLVM passes.

4.1 Compiling LLVM IR

First, the source code is compiled to LLVM IR as the input. Older versions of Linux can be successfully compiled to IR. However, asm-goto is enforced in newer

Linux versions. To the time of our experiments, Clang does not support asm-goto yet. To solve this problem, we use the techniques proposed by Xu et al. [17] to generate IR. Moreover, we succeed in compiling FreeBSD by following the steps in wllvm [3].

4.2 Unrolling Loops

In general, unrolling loops is a widely used strategy in static analysis to prevent analyzing abundant program paths. In this paper, we unroll loops by converting statements, such as `for` and `while`, to `if`. In LLVM IR, a loop is composed of several basic blocks. The entry of a loop is called the header, and the block jumping out of a loop is the latch. We modify the jump instruction in the latch to the successor after the loop, rather than to the header.

4.3 Deciding Memory Access (Dereferencing)

Our approaches need to judge whether a pointer accesses memory, and we use the following rules. First, GetElementPointer (GEP) is a pointer calculation instruction in LLVM IR. The GEP instructions are taken as memory dereferencing operations in our methods for these reasons. In GEP, the offset of the target is calculated, and the target is loaded. This process is similar to pointer dereferencing in C code. However, we do not mean GEP is a memory dereferencing instruction, and memory dereferencing requires GEP coupled with Load and Store. We use GEP for simplicity. Second, when a pointer or a MARV is passed to a function, there may be memory access in this function. In this case, we use a memory-access function set collected previously to judge this situation, e.g., `kmemdup`() is collected in the set. If a function is not in the set, we use a lightweight inter-procedural analysis to identify whether memory access exists inside the function.

4.4 Propagation Termination

For propagative MARV identification, we add a termination condition to stop the MARV propagation. Because it can cause considerable performance overhead, and it may cause false positives if a MARV propagates to uncontrollable scenarios. In MEBS, the propagation is designed to suspend when a user-space variable, a global variable, and similar cases are hit. It is challenging to perform more in-depth analysis in these cases.

We weigh the advantages and disadvantages of this problem. On the positive side, propagating these variables can cover more MARVs, which makes our analysis more complete. On the other hand, tracking these variables is more complex than tracking other kernel objects, e.g., the declaration and use of a global variable and a local one are different. Consequently, there may be unavoidable performance overhead when analyzing them. Moreover, the behaviors of these variables are more complicated, and analyzing them may cause false positives. Therefore, we decide to sacrifice MARV coverage to balance the performance overhead and false positives, and we terminate the propagation when these values are hit.

5 Evaluation

In this section, we show the evaluation results according to the following aspects.

- What is the precision of source function identification, sink function identification, and pair identification? (Subsections 5.3, 5.5, and 5.6)
- What is the effectiveness of MEBS in identifying MLC bugs? (Subsection 5.4)
- How is the portability and scalability of MEBS? (Subsection 5.3)

[3]Whole-program-llvm. Steps to build bitcode version of FreeBSD 10.0 world and kernel. `https://github.com/travitch/whole-program-llvm/blob/master/doc/tutorial-freeBSD.md`, Aug. 2021

- Can MEBS outperform other tools? (Subsection 5.4)

5.1 Setup

Our experiment is performed on a server with 32 cores (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz), 128 GB RAM, with Ubuntu 16.04 LTS with kernel 4.4. All the kernels are compiled with LLVM/Clang 9.0.1.

Table 2. Information of Kernels

Version	Date	LOC	Number of Files	IR Size	Evaluation Time(s)
Linux	2019.09	10.4M	18,074	4.4GB	310
FreeBSD	2019.07	1.3M	1,511	557MB	35

Table 2 shows the evaluation time in second and related information. LOC means the number of lines of code. MEBS accomplishes the bug detection task in about five minutes. In comparison with previous LLVM-based kernel static analysis tools (LRSan [18] analyzed Linux within four hours, and CRIX [12] spent more than one hour in analyzing Linux.), the analysis time of MEBS is at an acceptable level.

5.2 Indirect Call Analysis Results

We adopt two two-key dictionaries to identify the targets of the indirect calls. As discussed above, not all of them can hit the first dictionary. In our evaluation, we identify 79.1% indirect calls hit Dictionary1 in Linux and 81.6% in FreeBSD. Similarly, previous tools, such CRIX [12] and Pex [19], all had an identification rate of around 80% when using the struct type information.

The average number of targets for an indirect call is a benchmark to evaluate the precision of indirect call analysis. A smaller average number means higher precision. We examine this indicator in Dictionary1-hit cases. The average number of targets is 9.2 in Linux and 8.3 in FreeBSD. When adopting only the argument number and type in Dictionary2, the average number of targets is 173.9 in Linux and 155.1 in FreeBSD. This result demonstrates that the struct type provides more accurate information in identifying the indirect call targets. Moreover, when the first dictionary is hit, we use the second one to confirm the results. In this part, we have a confirmation rate of over 90% both in Linux and FreeBSD.

5.3 Precision of the Analysis Phase

Table 3. Results of Source Functions Judging on Source Function Frequency

f_s	Number	Precision
12.6 (Average)	750	49.8%
2 (Mode)	6,153	6.3%
2 (Median)	6,153	6.3%
0	10,276	3.5%

Table 4. Results of Source Functions Judging on Source Function Frequency Rate

$\frac{f_s}{f_t}$	Number	Precision
1.0	3,563	10.9%
0.9	3,779	10.1%
0.8	4,250	8.9%
0.7	4,710	8.1%
0.6	5,475	6.9%
0.5	7,019	5.3%
0.4	7,356	5.0%
0.3	8,155	4.5%
0.2	9,002	4.1%
0.1	9,769	3.7%
0	10,276	3.5%

5.3.1 Frequency and Frequency Rate of Source Functions

We discuss how to balance between the frequency f_s and the frequency rate $\frac{f_s}{f_t}$. We study the source function results (Linux) judging on f_s and $\frac{f_s}{f_t}$ in Table 3 and Table 4, respectively.

We select four thresholds for f_s, including the average source function frequency of all candidates, the mode, the median, and zero. By eliminating the candidates whose frequencies are strictly smaller than the thresholds, we collect the number of results and the precision, respectively, e.g., we have 750 source functions

with source function frequency greater than or equal to 12.6, and 49.8% of them are true positives. Similarly, we list the results judging on $\frac{f_s}{f_t}$ in Table 4. When we collect functions with a frequency rate equal to 1.0, we get 3,563 functions, and the precision is 10.9%. Furthermore, by studying the reason of this low precision, we find out that functions with $\frac{f_s}{f_t} = \frac{1}{1}$ account for most of the false positives. They occur only once, and this occurrence is identified as a source function behavior. We eliminate these false positives by giving different weights to $\frac{f_s}{f_t}$ and f_s when scoring a function.

By considering the precision of $\frac{f_s}{f_t}$ and f_s in judging source functions, we design a scoring strategy as below

$$f_s = \begin{cases} f_s & \text{if } f_s < 12.6, \\ 12.6 & \text{otherwise,} \end{cases}$$

$$Score = 100.0 \times (\frac{49.8}{49.8 + 10.9} \times \frac{f_s}{12.6} + \frac{10.9}{49.8 + 10.9} \times \frac{f_s}{f_t}).$$

The full score is 100, and we give portioned weights to $\frac{f_s}{f_t}$ and f_s based on the precision discussed above.

Table 5. Results of Source Functions of Different Thresholds

Threshold	Number	Precision
60.0	2,232	21.5%
70.0	1,062	47.3%
80.0	589	69.6%
85.0	405	78.3%
90.0	242	80.2%
95.0	62	96.8%

5.3.2 Thresholds of Source Functions

Based on this strategy, we give all candidates a score to filter out the potential false positives. We also study the values of thresholds (Linux) in Table 5. As we can see from the table, when the threshold is set to 80.0, the precision of 69.6% is already higher than the precision of judging on only $\frac{f_s}{f_t}$ (10.9%) or f_s (48.9%). By setting the threshold to 85.0, we get 405 source functions with precision of approximately 80%. This configuration has a relatively low false positive rate, and it is used in Subsection 5.4.

Table 6. Results of Sink Functions of Different Thresholds

Threshold	Number	Precision
4.8 (Average)	226	83.2%
4	273	79.8%
3	368	76.1%
2	626	75.5%
1	2,408	65.0%

5.3.3 Thresholds of Sink Functions

We extract customized sink functions from the error paths as discussed previously. By recording the sink function frequency f, we list the results (Linux) in Table 6. We list the number of results and precision of the candidates whose frequencies are greater than or equal to the thresholds. We extract 2,408 sink functions in total, and the precision is 65.0%, which is even higher than source functions scoring more than 70.0. This result implies that identifying sink functions in the error paths can considerably reduce the false positives. When the threshold is four, we have 273 sink functions with reasonably high precision. This configuration is used in Subsection 5.4.

5.3.4 Source-sink Pairs

When we identify a sink function in the error path, we go back to the allocation site of this freed pointer and identify a source-sink pair. The precision of source function and sink function identification affects the precision of pair identification, and we use source functions whose scores are greater than 85.0 and sink functions whose scores are greater than four as our configuration. Then, we record the frequency of each pair and select the most frequent pair for a source function or a sink function. We get 362 source-sink pairs, containing 245 true positives, with precision of 67.7%.

Furthermore, we study the false positives of source function identification, sink function identification, and pair identification and propose some methods to reach much higher precision in Subsection 5.5

Table 7. Analysis Statistics of Source Function and Sink Function Identification

Kernel	SC	SK	Pair	MARV	Avg. DU
Linux	405	273	362	35,359	15.1
FreeBSD	33	19	25	4,196	10.3

Note: SC is the number of source functions. SK is the number of sink functions. Pair is the number of source-sink pairs. MARV is the number of MARVs. Avg. DU is the average size of define-use trees of all MARVs

In conclusion, we list our analysis statistics (Linux and FreeBSD) in Table 7 with the above-discussed configuration. Besides the number of source functions (Column SC), sink functions (Column SK), and pairs (Column Pair), we record the number of MARVs (Column MARV) and the average size of define-use trees (Column Avg. DU).

5.3.5 *Case Study*

We further display the detection results of common source functions and sink functions. For example, `kmalloc`() scores 86.9 in our strategy, `kcalloc`() scores 94.0, and `kzalloc`() scores 93.2. Additionally, `kfree`() is identified 3,155 times in the error paths, and `vfree`() is detected 174 times. These results demonstrate that our methods can identify these common functions.

For these identified source functions and sink functions, we manually study the proportion of the customized ones. We identify 405 source functions in Linux and exclude `kmalloc`() and the related functions, e.g., `kzalloc`(). The rest are considered as customized source functions. In total, we get 391 customized source functions out of 405 (96.5%) functions. For the identified 273 sink functions, we have 266 customized functions (97.4%).

Identifying these customized source functions and sink functions can help expose more MLC bugs than only using common functions. Identifying them can foster related research on memory life-cycle, including bug finding and other fields. For example, these source functions can be used in reverse engineering to precisely track heap data.

5.4 Exposed Bugs And CVEs

Table 8. Part of the Bugs Exposed by MEBS

ID	File	SC/SK	MARV	T	S	MC	ML	KM
1	awcd	kstrndup_nul	get_name	IAF	A	×	×	×
2	dat_faker	kfree_const	ce→xat_back	IAF	A	×	×	×
3	dfs_cache	kfree_const	ce→ce_path	IAF	A	×	√	×
4	dm1105	kstrndup	dev→tsbuf	IAF	A	×	×	×
5	libcxgbi	kstrndup	astr	IAF	A	×	×	√
6	in...m	kstrdup	str	UD	A	×	√	×
7	req	kmalloc	new_ra	UD	A	√	×	√
8	cxi	kstrndup	der→name	UD	A	×	×	×
9	mpt3sas_ctl	kmalloc	ioc_number	UD	A	√	√	×
10	rap	kstrdup	prop→name	UD	A	×	×	√
11	mc	kstrdup_const	cpumask	UD	A	×	×	√
12	dcache	cmnd	opcode	UD	N	×	×	×
13	lola_mixer	efi..prolog	save_pgd	UD	A	×	×	×
14	dm..hash	kmalloc	nreg	UD	N	√	√	×
15	grant_table	kstrdup	de→args	UD	N	×	×	√
16	s7_ago	km..array	save_pgd	UD	A	×	×	√
17	tda1997x	kmalloc	imp→table	UD	A	√	×	×
18	tegra-hsp	devm..const	db→name	UD	C	×	×	×
19	cdgt	kmem..zalloc	s	UD	N	√	×	√
20	vt	kzalloc	vc→vc..buf	LF	C	√	√	×
21	virtio_net	kzalloc	p	LF	C	×	×	×
22	mon_client	kmemdup_nul	old→cfg	LF	A	√	×	×
23	tty_io	tty..device	tty→dev	LF	N	×	√	√
24	vpfe_capture	kcalloc	payload	LF	N	√	×	√
25	hooks	kmemdup_nul	arg	LF	A	×	√	×
26	mixer_map	km..array	pool2	LF	A	×	×	×
27	pcie...host	devm_kzalloc	glue	LF	N	×	×	×
28	spi	kzalloc	glue	LF	N	√	√	×
29	pcm_fabric	plat..alloc	pd..device	LF	N	×	×	√
30	teg..m9712	plat..alloc	ma..codec	LF	N	√	×	×
31	uinput	in...device	dev→name	UAF	N	×	×	√
32	tg3	dev...any	new_skb	UAF	C	×	×	√
33	core	clear_bit	flags	UAF	C	×	×	×
34	inode	put_page	page	UAF	N	×	×	×
35	keyctl	kfree	name	UAF	N	√	×	√
36	kt...ts	dev...free	new_buf	DF	N	×	×	√
37	qlnxr_verbs	kfree	qp	IAF	N	×	√	×
38	cm_xm	kfree	page→ring	IAF	C	×	×	×
39	fir_os	kzalloc	dev→ret	UD	C	√	×	√
40	netdev	kzalloc	priv	UD	N	√	×	×
41	fs_dir	kstrdup_const	p→name	UD	C	×	×	×
42	cloudabi_vdso	kva_alloc	addr	UD	N	×	√	×
43	dpp	kzalloc	priv	UD	N	√	×	√
44	mlx5_qp	kzalloc	mbox_in	LF	N	√	√	×
45	hw_verbs	ql..alloc	q→tbl	LF	C	×	×	×
46	aw_cir	evdev_alloc	sc..area	LF	C	×	×	×
47	altera_band	ta..create	sc→pat	LF	N	×	×	√

Note: T indicates the types of MLC bugs. S demonstrates the status of the corresponding patch, where A is already applied, C is confirmed and accepted by Linux maintainers but not applied to the time of writing this paper, and N is submitted. MC indicates whether MCChecker can identify this bug. ML indicates whether MLCSan can identify this bug. KM indicates whether K-Miner can identify this bug

In Table 8, we select part of the detected bugs for observation [4]. Among these MLC bugs, more than half are applied or confirmed by maintainers, and others are submitted.

In the File column of Table 8, several MLC bugs

[4]The other part of the detected bugs is available at `https://figshare.com/articles/online_resource/Bugs_Exposed_by_MEBS/15187785`.

are discovered in some critical components of the Linux kernel, e.g., Bug 1 is in the net sub-system of Linux.

In addition, the respective source functions and sink functions are listed in the SC and SK column, respectively, such as `kmalloc`() and `kfree`(). Some are customized source functions or sink functions, e.g., `efi_call_phys_prolog`() in Bug 13.

As for the T column, there are seven incorrect allocation-freeing bugs, 19 unchecked dereferencing bugs, 15 lacking freeing bugs, five use-after-free bugs, and one double-free bug in the table.

Table 9. Bugs cannot be Detected by MCChecker and the Corresponding Reasons

Reason	Bug ID
Incomplete source function and sink function identification	3, 4, 6, 8, 10, 11, 18, 22, 26, 31, 32, 33, 34, 36, 41, 42
Error-paths-only free check	2, 14, 19, 23, 27, 29, 37, 39, 47
Allocation-free ignorance	7, 12, 13, 16, 17, 40, 43

Additionally, for the MC column in the table, we conduct comparison experiments with MCChecker [13]. It used meta-level compilation extensions to check kernel memory allocation, dereferencing, and freeing. We adjust MCChecker to check the already identified bugs by MEBS. As Table 8 shows, MCChecker succeeds in identifying 15 bugs and fails to detect the other 32 bugs. As discussed in our introduction, the most significant difference between MEBS and MCChecker is that the analysis of MEBS is under the memory life-cycle indivisibility. The difference can be found in three aspects when compared with MCChecker. 1) MEBS identifies all the source functions and sink functions in the kernel. 2) By completely examining the define-use tree, MEBS detects lacking freeing cases not only in the error paths but also in the non-error paths. 3) MEBS considers the allocation-freeing relationship. Because of the incomplete and absent examination of the relationship between allocation-dereferencing, dereferencing-freeing, and allocation-freeing, MCChecker fails in detecting the 32 bugs. Table 9 demonstrates the corresponding bugs that MCChecker cannot find.

In addition, we compare MEBS with MLCSan [5] in our experiments. The ML column of Table 8 shows whether MLCSan can identify the bugs. MLCSan can detect only 11 out of the 47 bugs. In contrast to MLCSan, MEBS improves the MLC bug definition. We add use-after-free and double-free to MLC bugs. They complete the original definition of MLCSan. In addition, we propose new techniques to identify the customized source functions and sink functions in OS kernels. The detection approach in MLCSan identified source functions and sink functions through function calling relationships, and we propose feature-based source function identification and error-path-based sink function identification.

Moreover, we compare K-Miner [20] with MEBS in our evaluation. The results of K-Miner are listed in the KM column of Table 8. In the 47 MLC bugs, K-Miner can identify 17 of them. K-Miner detected three types of bugs, including dangling pointer (DP), use-after-free, and double-free. Meanwhile, MEBS detects five types of MLC bugs. In these bugs, K-Miner cannot detect incorrect allocation-freeing, unchecked dereferencing, and lacking freeing. According to our evaluation, K-Miner fails to detect most of these three types of MLC bugs. The reason behind this is the incomplete identification of the source functions of K-Miner. As our evaluation indicates, there are 405 allocation functions identified by MEBS. Missing this large number of functions makes K-Miner fail to identify the MLC bugs.

Table 10. CVEs Detected by MEBS

Entry	Description	Type
19-12378	There is an unchecked `kmalloc()` of `new_ra`, which might allow an attacker to cause a denial-of-service (NULL pointer dereference and system crash).	UD
19-12379	There is a memory leaking in a certain case of an ENOMEM outcome of `kmalloc()`.	LF
19-12380	`phys_efi_set_virtual_address_map` mishandle memory allocation failures.	UD
19-12381	There is an unchecked `kmalloc()` of `new_ra`, which might allow an attacker to cause a denial-of-service (NULL pointer dereference and system crash).	UD
19-12382	There is an unchecked `kstrdup()` of `fwstr`, which might allow an attacker to cause a denial-of-service (NULL pointer dereference and system crash).	UD
19-12454	It uses `kstrndup()` instead of `kmemdup_nul()`, which allows attackers to have an unspecified impact via unknown vectors.	IAF
19-12455	There is an unchecked `kstrndup()` of `derived_name`, which might allow an attacker to cause a denial-of-service (NULL pointer dereference and system crash).	UD
19-12456	It allows local users to cause a denial-of service or possibly have unspecified other impact.	UD
19-12614	There is an unchecked `kstrdup` of `prop` → `name`, which might allow an attacker to cause a denial-of-service (NULL pointer dereference and system crash).	UD
19-12615	There is an unchecked `kstrdup_const()` of `node_info` → `vdev_port.name`, which might allow an attacker to cause a denial-of-service (NULL pointer dereference and system crash).	UD
20-18574	RESERVED	IAF
20-18575	RESERVED	UD

In Table 10, we have 12 CVEs assigned for the detected bugs. Two CVEs belong to incorrect allocation-freeing, nine in unchecked dereferencing, and one in lacking freeing. Besides, there are one memory leaking vulnerability and 11 denial-of-service vulnerabilities altogether. The security issues of these CVEs are clear. For example, unfreed memory space may cause memory leaking in OS kernels. Mal-users can compromise the kernel by using the leaking data. In addition, we manually investigate the results. The assigned CVEs cannot be detected by other tools. All the CVEs are from bugs which can only be detected by MEBS.

5.5 False Positives

5.5.1 False Positives in Source Function, Sink Function, and Pair Identification

As discussed above, the false positive rates of source functions whose scores are greater than 85.0, sink functions whose scores are greater than four, and pairs are 21.7%, 20.2%, and 32.3%, respectively. We manually investigate the causes of these false positives and propose some methods to eliminate them.

Table 11. Results of Source Functions, Sink Functions, and Pairs after Eliminating the Known False Positives

Item	Total	Correct	Precision
Source	309	280	90.6%
Sink	233	211	90.6%
Pair	297	245	82.5%

For source function identification, we have 88 false positives in total. There are mainly two kinds of false positives in them. 1) There are 68 search functions, e.g., `get_dev_by_name()`. These search functions have the same features as the source functions. 2) The other 20 functions also have the same features as the source functions. However, by checking the code of them and the source functions, we find out that the source functions often reset the allocated pointer inside the function. These false positives rarely have this procedure, and they return the allocated pointer directly to the caller. By using this observation, we successfully exclude 96 functions and achieve much higher precision of 90.6% (Linux). The result is listed in Table 11.

For sink function identification, we also study the cause of the false positives. 1) There are 36 debugging functions, e.g., `xfs_warn()`. They are located in the error paths and have similar behaviors to the sink functions. 2) There are 19 functions executed along with the sink functions. For example, `cpumask_next()` is executed in the error path and is along with a customized sink function. By excluding these functions, we finally get 233 sink functions with a false positive rate of 9.6%.

By using the above source functions and sink functions, we get 297 source-sink pairs in total, and 245 of them are true positives. Compared with 245 out of 363 (67.7%), we eliminate the potential false positives and reach higher precision.

5.5.2 False Positives of Bug Reporting

MEBS has an acceptable false positive rate in bug reporting. According to the evaluation, MEBS outputs 385 reports, and 169 are manually verified as true positives. We believe a false positive rate of 56.1% is acceptable in kernel analysis tools. For example, Wang et al. [18] claimed that 19 bugs were identified after checking over 2,000 lacking-recheck cases. Compared with it, MEBS has a much lower false positive rate.

We use several techniques to eliminate part of the false positives. 1) Several practical issues are considered. Some kernel functions have an `__init` attribute. If memory allocation of this kind fails, the whole system will reboot. Therefore, this kind of function is removed from our source function set. Other cases, e.g., `kmalloc(size, GFP_NOFAIL)`, are also handled. 2) For MARV identification, we utilize a termination condition to suspend the pointer propagation.

We summarize the causes of unsolved false positives: 1) Precise call graph construction and pointer analysis are always challenging for static analysis, e.g., indirect call analysis. More than half of the false positives are caused by them. For instance, there are three functions F1, F2, and F3. F1 calls F2 and then F3. Variable `ptr` is allocated in F2 and freed in F3. This case can be incorrectly identified as a lacking freeing bug since the relation between F2 and F3 is hard to catch.

2) There are some kernel issues related to implementation. It is difficult to identify these cases simply by a pattern match. For instance, the Linux kernel uses a null pointer dereferencing to check the status of the system. Therefore, we exclude them from identified bugs. Furthermore, `devm_kmalloc()` is a managed memory allocation function. On driver detach, memory allocated by it will be automatically freed. These complicated issues are out of our scope, and we simply treat them as false positives.

3) Some false positives result from the false positives in source function identification, sink function identification, and pair identification.

4) Others. There are other causes of false positives, such as programming complexity and imperfect analysis techniques. In total, they are about 15% of the false positives.

5.6 False Negatives

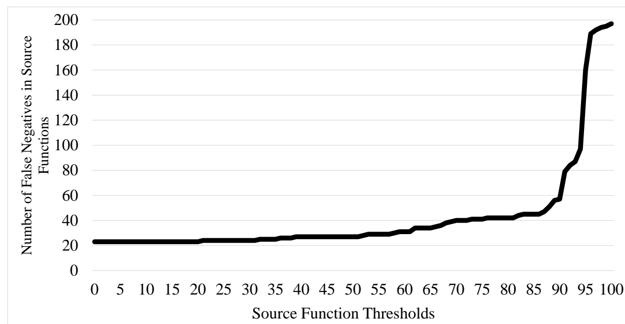


Fig. 7. Relationship between the number of false negatives in source functions and the source function thresholds.

For the false negatives of source function and sink function identification, we manually collect source functions and sink functions mentioned in the latest patches as the ground truth. We study the relationships between the number of false negatives and thresholds of source function and sink function identification. For source function identification, we collect 200 ground-truth customized source functions, and the result is in Fig. 7. This figure indicates that the number of false negatives increases as the threshold increases. The false negative rate is around 10% at the lowest point.

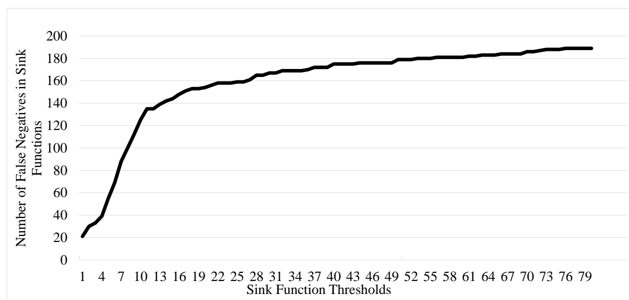


Fig.8. Relationship between the number of false negatives in sink functions and the sink function thresholds.

Similarly, we collect 200 ground-truth sink functions from the patches. Fig. 8 lists the relationship between the number of false negatives and the sink function thresholds. We eliminate thresholds more than 80 because the maximum threshold is too large (3,155), and the false negatives already reach the highest point when the threshold is 80. When the threshold is four as in our configuration, the false negative rate is around 15%, which is at an acceptable level.

Though well-designed static analysis approaches are used, false negatives cannot be entirely removed. First, not all source code is successfully compiled, and some sub-components may be missing. However, these incompatible files are rare. Besides, we cannot manually model all assemblies. Therefore, some memory allocation functions are not identified. Additionally, some source functions store the allocated pointer in the arguments, rather than in the return value. Our approach cannot cover these situations.

Moreover, there are challenges in pointer analysis. The identification of aliases and the termination of pointer propagation would cause potential false negatives. In addition, we extract source-sink pairs from the error paths. This approach is not able to identify all the source-sink match rules in the kernel and can bring about false negatives.

5.7 Portability and Scalability

The techniques in MEBS, e.g., feature-based source function identification, are not only applicable for Linux or FreeBSD. Memory life-cycle also exits in the user-space programs, e.g., pointers are allocated by `malloc()` and freed by `free()`. Source function and sink function identification can be adopted in these programs. With slight modification, the define-use tree can also be used in these programs.

Besides, we can compile programs into LLVM IR in other platforms as well, such as Windows and macOS, and conduct analysis on these platforms.

Previous work, such as Lu et al. [12] and Zhang et al. [19], claimed to use scalable techniques but failed to conduct scalability experiments. Their evaluations were only on the Linux platform.

6 Discussion

6.1 Significance of MLC Bugs

MLC bugs are common in kernels, and they can cause security issues in the system. We conduct a pioneer study in the CVE database. There are 2,667 vulnerabilities in the Linux kernel in total from 1999 to 2019. As discussed above, incorrect allocation-freeing and lacking freeing cases can cause potential memory leaking as the memory space is improperly freed or not freed. Similar memory leaking makes up 6.1% of all the vulnerabilities. Moreover, in an unchecked dereferencing case, a MARV is dereferenced without checking, resulting in a possible null pointer dereferencing, which occupies 8.7% of the vulnerabilities in the database. Use-after-free bugs cause memory corruption, making up 4.9% of the vulnerabilities. Therefore, MLC bugs are common in OS kernels with a 20% proportion of all the recorded vulnerabilities. On the other hand, from a theoretical perspective, all the allocated memory fits

the three steps in memory life-cycle, and any issue in them causes MLC bugs.

MLC bugs can cause severe consequences and security problems, such as denial-of-service and information leakage, and we have 12 CVEs assigned. Among them, we get high CVSS scores. We do not provide proof-of-concept (PoC) because this work focuses on detection, and exploitation is beyond the scope.

6.2 Findings

There are some findings from our evaluation results. Allocation, dereferencing, and freeing do tightly relate to each other. Our intuition to treat them as inseparable is reasonable. For example, in a lacking freeing case, unchecked dereferencing often coexists. This is solid proof that it is appropriate and necessary to consider the three steps as indivisible.

Furthermore, the size of the define-use tree varies from less than 10 to about 2,000. Too many operations on the same MARV can be error-prone since no one can guarantee the correctness in thousands of operations. Next, we discover that source-sink rules exist in kernels. Programmers should follow the rules when allocating or freeing memory. However, these rules are often kept in the comments that sometimes can be difficult to notice when programming.

Next, several applied patches have not been assigned CVE entries. For example, item 16 in Table 8 deals with an incorrect allocation-free case in the Linux kernel. However, the CVE assignment team does not accept it as a CVE. Because incorrect use of `kstrndup`() in this patch can hardly be exploitable, and it is not a vulnerability. It is always challenging to identify whether a bug is an exploitable vulnerability. The automation of this process requires analyzing the potential security impact of the bug. By automatically inferring the impact of MLC bugs, we are possibly able to detect bugs, submit patches, and require CVEs with little manual effort.

6.3 Unified CWE Names of MLC Bugs

CWE (common weakness enumeration) [5] is a community-developed list of software and hardware weakness types. Incorrect allocation-freeing is similar to CWE-665 (improper initialization) and CWE-404 (improper resource shutdown or release). Unchecked dereferencing is CWE-476 (null pointer dereference). Lacking freeing is CWE-772 (missing release of resource after effective lifetime). Use-after-free is CWE-416 (use-after-free). Double-free is CWE-415 (double-free).

6.4 Backward Analysis of Source-sink Pairs

The backward analysis does not miss many situations. For example, the heap block is allocated in function A, used in function B, and released in function C. First, such cases are not common in kernels. According to our evaluation, these cases are about 9% of all the cases. For most of the situations, the allocation, dereferencing, and freeing sites are in the same function. Second, even in such rare situations, MEBS can still find the source-sink pairs. The analysis process of MEBS is inter-procedural. With the help of an inter-procedural call graph, we can catch the relationships between A and B or A and C. We can traverse backward from a sink function in C to the memory pointer in B, then to the target source function in A. Third, we can see from the experiment results that MEBS detects 362 source-sink pairs in Linux. The numbers of the source functions and sink functions are 405 and 273, respectively. The results indicate that MEBS does NOT miss many source-sink pairs.

[5] The MITRE corporation. Common weakness enumeration. `https://cwe.mitre.org/index.html`, Aug. 2021

6.5 Future Work

First, considering the false positives in our paper, we can solve some problems with more precise call graph construction. In indirect call analysis, we design the two two-key dictionaries because most address-taken functions are initialized in a struct field, but not all of them. We can incorporate more information to deal with situations where the first dictionary fails. Second, in source function and sink function identification, we can use more precise data-flow analysis to determine whether one function is a source function or a sink function.

Moreover, by collecting more external information, we may be able to infer some kernel implementation issues and make some improvements, e.g., identifying some permitted allocation failures (often used to check whether the kernel is functioning correctly). Besides, assemblies can be manually modeled in kernels to address the assembly issue. As for the most challenging problem in pointer analysis, we can follow the latest achievements in this scope and adopt more precise pointer analysis in our future work.

7 Related Work

7.1 Memory Management Model Checking

Model checking is an automatic program verification method, and specifications need to be written to perform the checking. MEBS can be classified as a model checking tool for memory management, and some related work has been proposed by researchers [13–15]. We take the most related one to state our point. Engler et al. checked memory with metal-level compiler extensions [13] (MCChecker for short). MCChecker checked kernel memory allocation, dereferencing, and freeing, which resembled MEBS at first glance. However, MCChecker failed to follow the inseparability of memory life-cycle in the following aspects. 1) MCChecker did not identify all the source functions and sink functions in OS kernels. According to its description, it only checked the results of several frequently used allocation and freeing functions such as `kmalloc`() and `kfree`(). This insufficiency in source function and sink function identification hindered MCChecker from completely examining the relationship related to allocation and freeing. 2) MCChecker only checked the absent freeing operations in the error paths, which is an incomplete examination of the dereferencing-freeing relationship. A thorough inspection of the relationship should not only be in the error paths but also in the non-error paths. 3) Identifying errors in the allocation-freeing pairs is a significant component of revealing the relationships in the life-cycle, but MCChecker ignored doing so. Therefore, according to the above three defects, MCChecker did not maintain the indivisibility of memory life-cycle.

7.2 API Usage Verification

We check the kernel APIs to identify MLC bugs, and there are several related tools. SSLint [6] identified incorrect use of APIs in SSL/TLS protocols. Joern [7] used a code property graph to represent a program. Furthermore, APISan [9] adopted semantic cross-checking to verify APIs. MLC bugs are different from API misuse. Incorrect allocation-freeing is similar to API misuse. However, unchecked dereferencing, lacking freeing, use-after-free, and double-free are not related to API misuse. Moreover, none of the above API verification tools checked the allocation-free relationship and thus broke the indivisibility of memory life-cycle. MEBS is different from these tools because we use memory life-cycle and analyze the allocation-freeing relationship. We concentrate not only on a single API but also analyze the allocation-freeing pairs.

7.3 Missing Check Bugs

MEBS can detect unchecked dereferencing. Recently, some related work on missing check bugs was published, such as [10–12]. Moreover, Wang et al. [18] formally defined a lacking-recheck bug. In this paper, when a critical variable was modified, a security recheck should be enforced to prevent lacking-recheck bugs. In addition, CRIX [12] treated missing check bug as a new bug type. It identified over 2,000 bugs via semantic- and context-aware techniques. However, these missing check tools only inspected the allocation-dereferencing relationship, not treating the life-cycle as an inseparable entity. As for our work, MLC bugs cover missing check cases, including other four types of bugs. MEBS corresponds with the life-cycle inseparability with additional analysis of dereference-freeing and allocation-freeing.

7.4 MLC Bugs

Zhang proposed the first work which used the concept of MLC bugs [5]. Compared with MLCSan, MEBS has three aspects of novelties: 1) We extend the definition of MLC bugs. Use-after-free and double-free are added to MLC bugs. They complement the original definition of MLC bugs in MLCSan. In addition, we propose formal definitions of MLC bugs in Section 2. We explain the bugs from a higher level of vision than MLCSan. 2) We propose novel analysis techniques to identify MLC bugs. First, we propose a more detailed preprocessing stage. This stage contains call graph construction and pointer analysis. Specifically, we introduce dictionaries to recognize the targets of indirect calls. MLCSan failed to do so. Second, MLCSan identified the customized functions based on `kmalloc`() and `kfree`(). We propose feature-based source function identification and error-path-based sink function identification. Our methods are more precise than the identification approaches in MLCSan. Third, the bug reporting of MEBS is also superior to MLCSan. MLCSan did not describe the bug reporting in detail, and from our point of view, its bug reporting may require intensive manual work. MEBS automates this process and describes the details clearly. 3) We perform more extensive experiments compared with MLCSan. First, we list the results of indirect call analysis. MLCSan failed to do so. Second, we identify 127 more bugs and 12 more CVEs than MLCSan. Third, MEBS shows the precision of source function and sink function identification. We also discuss the selection of thresholds of the results. MLCSan failed to do so. Fourth, we perform analysis on the false positives and the false negatives. MLCSan did not contain this part. In conclusion, MEBS outperforms MLCSan in the above aspects.

7.5 Typestate Analysis

Tac [21] bridged the gap between typestate and pointer analyses by machine learning. This paper captured the correlations between program features and use-after-free-related aliases to help typestate analysis in finding true use-after-free bugs. UAFL [22] was a typestate-guided fuzzer. It performed typestate analysis to identify the operation sequences potentially violating the typestate properties and used operation sequence coverage to guide the fuzzing process. 2ndStrike [23] was a method to manifest hidden concurrency typestate bugs in software testing. It profiled runtime events related to the typestates and thread synchronization.

The differences between them and MEBS are as follows. First, the definitions of memory life-cycle of MEBS and the typestate automata in Tac, UAFL, and 2ndStrike are totally different. Our life-cycle consists of allocation, dereferencing, and freeing as three indivisible steps of a kernel memory pointer. The typestate of [21, 22] contained “live”, “dead”, and “error” in its

finite state automata. The definition of [23] was a little bit more complex but more or less the same. It had states for file descriptors, pointers, and locks. For example, the states of a pointer contained “valid”, “dangle”, and “null”. The states of the automata in [21–23] were artificial tags used to record the triggering of use-after-free and concurrency bugs. Second, the memory life-cycle is used to describe the behavior of a memory pointer. When the integrity of the life-cycle is compromised, MLC bugs happen. The automata in Tac, UAFL, and 2ndStrike showed only the results of use-after-free and concurrency bugs. Our memory life-cycle can reveal the root cause of MLC bugs, and [21–23] failed to. For example, for an unchecked dereferencing bug, the life-cycle can tell us it is triggered because of a dereferencing without a null check. Third, MEBS can detect five types of bugs. [21,22] detected only use-after-free bugs, and [23] detected only concurrency bugs.

7.6 Resource Usage Bugs

RUV [24] employed a mixture of compile-time analysis and run-time testing to verify that a program conformed to a resource usage policy specified by the deterministic finite state automata which detailed the allowed sequences of operations on resources. The resource usage policy of RUV seemed similar to the life-cycle of MEBS, but they are different. Resource usage policy was a regular language specified by the automata. The resource usage policy of a program was captured by its resource usage traces. Manually intensive work was required to generate the policies from the execution traces. However, memory pointers are born to have allocation, dereferencing, and freeing in the life-cycle. Compared with the resource usage policy, the life-cycle is not specified by a certain procedure. It is a natural rule for a memory pointer. The usage of memory pointers must obey the life-cycle.

State-taint analysis [25] proposed a static analysis called state-taint analysis to detect resource bugs. It dealt with the open-but-not-used problem of resources. The differences between state-taint analysis and MEBS are clear. First, state-taint analysis considered only the initialization, opening, and using of a resource to detect the open-but-not-used problem. Compared with the life-cycle of MEBS, the states of state-taint analysis were incomplete. MEBS contains allocation, dereferencing, and freeing of a pointer. Second, the states of [25] were also manually defined artifacts. On the one hand, the states were not related to the root cause of the bugs and vulnerabilities. In contrast, the life-cycle in MEBS reveals the root cause of MLC bugs. On the other hand, the paper [25] indicated that it also required manually intensive work to extract the rules to detect the bugs. On the contrary, MEBS requires no such work to construct the life-cycle.

8 Conclusion

In this paper, we propose the concept of memory life-cycle in kernels and conduct systematical study on MLC bugs. By using feature-based source function identification and error-path-based sink function identification, we detect 405 and 273 source functions and sink functions, respectively. Identifying these functions can foster related research on memory life-cycle, including bug finding and other fields in OS kernels. In addition, we have 169 new bugs identified and 12 CVEs assigned. We can identify 127 more bugs and 12 more CVEs than existing work. The results demonstrate that MEBS is effective in detecting MLC bugs. Moreover, MLC bugs are common in OS kernels and they can cause security issues. We need to focus on them to prevent security issues. Compared with previous work, we can identify more source functions, sink functions, and MLC bugs with higher precision. However, the pointer

analysis in our paper is still inaccurate in some scenarios. We will propose more precise pointer analysis in the future. Moreover, this paper does not contain the exploitation of MLC bugs. In the future, we will propose exploitation methods to automatically exploit MLC bugs.

References

[1] Akritidis P, Cadar C, Raiciu C, Costa M, Castro M. Preventing memory error exploits with WIT. In *Proc. the 2008 IEEE Symposium on Security and Privacy*, May 2008, pp.263-277. DOI: 10.1109/SP.2008.30.

[2] Lee B, Song C, Kim T, Lee W. Type casting verification: Stopping an emerging attack vector. In *Proc. the 24th USENIX Security Symposium*, Aug. 2015, pp.81-96. DOI: 10.5555/2831143.2831149.

[3] Szekeres L, Payer M, Wei T, Song D. Sok: Eternal war in memory. In *Proc. the 2013 IEEE Symposium on Security and Privacy*, May 2013, pp.48-62. DOI: 10.1109/SP.2013.13.

[4] Xu J, Mu D, Chen P, Xing X, Wang P, Liu P. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proc. the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2016, pp.529-540. DOI: 10.1145/2976749.2978340.

[5] Zhang G. Detecting memory life-cycle bugs with extended define-use chain analysis. *IEEE Access*, 2020, 8: 114968-114980. DOI: 10.1109/ACCESS.2020.2999351.

[6] He B, Rastogi V, Cao Y, Chen Y, Venkatakrishnan V N, Yang R, Zhang Z. Vetting SSL usage in applications with SSLint. In *Proc. the 2015 IEEE Symposium on Security and Privacy*, May 2015, pp.519-534. DOI: 10.1109/SP.2015.38.

[7] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In *Proc. the 2014 IEEE Symposium on Security and Privacy*, May 2014, pp.590-604. DOI: 10.1109/SP.2014.44.

[8] Chen H, Wagner D. MOPS: An infrastructure for examining security properties of software. In *Proc. the 9th ACM Conference on Computer and Communications Security*, Nov. 2002, pp.235-244. DOI: 10.1145/586110.586142.

[9] Yun I, Min C, Si X, Jang Y, Kim T, Naik M. Apisan: Sanitizing API usages through semantic cross-checking. In *Proc. the 25th USENIX Security Symposium*, Aug. 2016, pp.363-378. DOI: 10.5555/3241094.3241123.

[10] Son S, McKinley K S, Shmatikov V. Rolecast: Finding missing security checks when you do not know what checks are. In *Proc. the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Oct. 2011, pp.1069-1084. DOI: 10.1145/2048066.2048146.

[11] Yamaguchi F, Wressnegger C, Gascon H, Rieck K. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proc. the 2013 ACM SIGSAC Conference on Computer and Communications Security*, Nov. 2013, pp.499-510. DOI: 10.1145/2508859.2516665.

[12] Lu K, Pakki A, Wu Q. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proc. the 28th USENIX Security Symposium*, Aug. 2019, pp.1769-1786. DOI: 10.5555/3361338.3361461.

[13] Engler D, Chelf B, Chou A, Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. the 4th Conference on Symposium on Operating System Design and Implementation*, Oct. 2000, Article No.: 1. DOI: 10.5555/1251229.1251230.

[14] Engler D, Chen D Y, Hallem S, Chou A, Chelf B. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 2001, 35(5): 57-72. DOI: 10.1145/502059.502041.

[15] Brown F, Notzli A, Engler D. How to build static checking systems using orders of magnitude less code. In *Proc. the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2016, pp.143-157. DOI: 10.1145/2872362.2872364.

[16] Lu K., Pakki A, Wu Q. Automatically identifying security checks for detecting kernel semantic bugs. In *Proc. the 2019 European Symposium on Research in Computer Security*, Sep. 2019, pp.3-25. DOI: 10.1007/978-3-030-29962-0_1.

[17] Xu M, Qian C, Lu K, Backes M, Kim T. Precise and scalable detection of double-fetch bugs in OS kernels. In *Proc. the 2018 IEEE Symposium on Security and Privacy*, May 2018, pp.661-678. DOI: 10.1109/SP.2018.00017.

[18] Wang W, Lu K, Yew P C. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proc. the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2018, pp.1899-1913. DOI: 10.1145/3243734.3243844.

[19] Zhang T, Shen W, Lee D, Jung C, Azab A M, Wang R. Pex: A permission check analysis framework for linux kernel. In *Proc. the 28th USENIX Security Symposium*, Aug. 2019, pp.1205-1220. DOI: 10.5555/3361338.3361422.

[20] Gens D, Schmitt S, Davi L, Sadeghi A R. K-Miner: Uncovering memory corruption in Linux. In *Proc. the 2018 Network and Distributed System Security Symposium*, Feb. 2018. DOI: 10.14722/ndss.2018.23331.

[21] Yan H, Sui Y, Chen S, Xue J. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proc. the 33rd Annual Computer Security Applications Conference*, Dec. 2017, pp.42-54. DOI: 10.1145/3134600.3134620.

[22] Wang H, Xie X, Li Y, Wen C, Li Y, Liu Y, Sui Y. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proc. the 2020 IEEE/ACM 42nd International Conference on Software Engineering*, Oct. 2020, pp.999-1010. DOI: 10.1145/3377811.3380386.

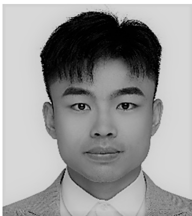
[23] Gao Q, Zhang W, Chen Z, Zheng M, Qin F. 2nd-Strike: Toward manifesting hidden concurrency typestate bugs. *ACM SIGPLAN Notices*, 2011, 46(3): 239-250. DOI: 10.1145/1961296.1950394.

[24] Marriott K, Stuckey P J, Sulzmann M. Resource usage verification. In *Proc. the 2003 Asian Symposium on Programming Languages and Systems*, Nov. 2003, pp.212-229. DOI: 10.1007/978-3-540-40018-9_15.

[25] Xu Z, Wen C, Qin S. State-taint analysis for detecting resource bugs. *Science of Computer Programming*, 2018, 162: 93-109. DOI: 10.1016/j.scico.2017.06.010.

Gen Zhang received his B.S. and M.S. degrees in computer science and technology, in 2016 and 2018, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now pursuing his Ph.D. degree in the College of Computer, National University of Defense Technology, Changsha. His research interests include fuzzing and software testing.

Peng-Fei Wang received his B.S., M.S., and Ph.D. degrees in computer science and technology, in 2011, 2013, and 2018, respectively, from the College of Computer, National University of Defense Technology, Changsha. His research interests include operating system and software testing.



Tai Yue received his B.S. and M.S. degrees in computer science and technology, in 2017 and 2019 from Nanjing University, Nanjing and the College of Computer, National University of Defense Technology, Changsha. He is now pursuing his Ph.D. degree in the College of Computer, National University of Defense Technology, Changsha. His research interests include fuzzing and software testing.

Xu Zhou received his B.S., M.S., and Ph.D. degrees in computer science and technology, in 2007, 2009, and 2014, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now an assistant professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems and parallel computing.

Kai Lu received his B.S. degree and Ph.D. degree in 1995 and 1999, respectively, both in computer science and technology, from the College of Computer, National University of Defense Technology, Changsha. He is now a professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems, parallel computing, and security.