IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received May 4, 2020, accepted May 29, 2020, date of publication June 2, 2020, date of current version July 1, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2999351

Detecting Memory Life-Cycle Bugs With Extended Define-Use Chain Analysis

GEN ZHANG
College of Computer, National University of Defense Technology, Changsha 410073, China
e-mail: zhang.gen@foxmail.com

This work was supported in part by the the National Key Research and Development Program of China under Grant 2016YFB0200401, in part by program for New Century Excellent Talents in University, in part by the National Science Foundation (NSF) China, under Grant 61402492, Grant 61402486, Grant 61379146, and in part by the Laboratory Pre-research fund under Grant 9140C810106150C81001.

ABSTRACT OS kernels leverage various memory allocation functions to carry out memory allocation, and memory data in kernel space of OS should be cautiously handled, e.g., allocating with *kmalloc()* and freeing with *kfree()*. However, real cases do exist where memory data is incorrectly allocated/freed, not checked before dereferenced, or left unfreed when out of use. We define these cases as Memory Life-cycle (**MLC**) bugs, and according to what we know, this new type of software bugs has not been deeply researched yet. In this paper, we go deep into the life-cycle of kernel memory space, including allocation, dereference and free, and propose the first systematical study of MLC bugs and build an automated and scalable detection framework, MLC bug sanitizer (**MLCSan**). MLCSan is capable of revealing memory allocation and free functions OS kernels. Besides, the occurrences of allocating, dereferencing and freeing sites can be automatically detected by MLCSan, leading to cases where MLC bugs may appear. Moreover, experiment result of analyzing the latest mainline OS kernels with MLCSan is a strong proof that MLCSan is effective in detecting MLC bugs and can scale to different platforms, in which 41 new bugs are identified in Linux and FreeBSD. And undoubtedly, we will open source MLCSan prototype to contribute to the security research in this area.

INDEX TERMS Systems security, software and application security.

I. INTRODUCTION

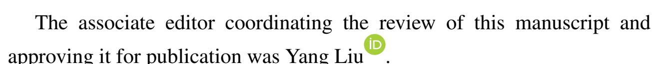
Vulnerabilities in OS can be intentionally exploited and thus the entire system may suffer from various attacks. Recent works [1]–[16] in security community also highlight the necessity to focus on OS kernel security. Memory in kernel space is allocated either by stack allocation or heap allocation. Stack memory is allocated and deallocated automatically by compilers, which can scarcely malfunction [17], while heap memory is dynamically allocated by programmers with allocation functions. Consequently, manually allocating heap memory tends to error-prone [17], [18], resulting in severe security issues such as memory leaks, privilege escalation and system crash, which should be attached more importance.

Memory allocation exits both in user and kernel space, and typically, Linux kernels carry out *kmalloc()*, *kmem_cache_alloc()*, etc. to conduct heap allocations. And a **MAR** is the pointer variable returned by a memory allocation function, pointing to the allocated memory space. Additionally, when the allocation fails, th function will return a null pointer as is often the case. Consequently, this MAR can no longer be dereferenced and some error handling code should be executed. Generally speaking, a MAR is initially allocated by an allocation **source** function, then dereferenced and finally freed by a **sink** function. We can summarize the life-cycle of kernel memory space as 3 steps, including (1) allocation, (2) dereference and (2) free finally. By thoroughly considering such 3 significant steps, we introduce a new bug classification, MLC bug.

```
1   static int wcd9335_codec_enable_dec(*w, ...)
2   {
3       u8 hpf_coff_freq;
4 -     widget_name = kstrndup(w->name, 15, GFP_KERNEL);
5 +     widget_name = kmemdup_nul(w->name, 15, GFP_KERNEL);
6       if (!widget_name)
7           return -ENOMEM;
8       dec_adc_mux_name = strsep(&widget_name, " ");
9   }
10  //sound/soc/codecs/wcd9335.c
```

FIGURE 1. **Part of an applied patch file. In this patch, MAR *widget_name* is allocated by a correct source *kmemdup_nul()*, instead of *kstrndup()*.**

Figure 1 is an applied patch detected by MLCSan, where MAR *widget_name* is improperly allocated by *kstrndup()*, which should be *kmemdup_nul()*. This is an issue in


The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

```
1    static int get_vdev_port_node_info(struct ...
2    {
3        node_info->vdev_port.name = kstrdup_const(name,
4                                         GFP_KERNEL);
5 +      if (!node_info->vdev_port.name)
6 +          return -1;
7        node_info->vdev_port.parent_cfg_hdl =
8                                    *parent_cfg_hdlp;
9        strncmp(node_info->vdev_port.name, ...);
10   }
11   //arch/sparc/kernel/mdesc.c
```

FIGURE 2. Part of an applied patch file, detected by MLCSan. In this patch, MAR ***node_info→vdev_port.name*** is checked when source ***kstrdup_const()*** fails.

```
1    static int selinux_add_mnt_opt(const char *option, ...
2    {
3        mnt_opts = kzalloc(sizeof(struct selinux_mnt_opts),
4                                         GFP_KERNEL);
5        if (token != Opt_seclabel) {
6            val = kmemdup_nul(val, len, GFP_KERNEL);
7 +          if (!val) {
8 +                  selinux_free_mnt_opts(*mnt_opts);
9 +                  return -ENOMEM;
10 +          }
11       }
12       rc = selinux_add_opt(token, val, mnt_opts);
13   }
14   //security/selinux/hooks.c
```

FIGURE 3. Part of an applied patch file, detected by MLCSan. In this patch, MAR ***val*** is checked and ***mnt_opts*** is freed when source ***kmemdup_nul()*** fails.

step (1), that an allocation function is incorrectly used. And in Figure 2P, a check for MAR *node_info→vdev_port.name* is enforced after allocated by source *kstrdup_const()*, which is related to step (2). Moreover, as shown in Figure 3, MAR *val* is allocated by source *kmemdup_nul()*, and it should be checked before dereferenced in *selinux_add_opt()*. Furthermore, the related MAR *mnt_opts* should be freed to prevent potential memory leaks in kernel. By considering the life-cycle of memory space, we can figure out that step (3) malfunctions in this case, where a free operation is absent.

In conclusion, we define bugs illustrated above as **MLC bugs**, which can be classified into three different scenarios: **incorrect source-sink** (ISS), **missing check** (MC) and **lacking sink** (LS). Generally speaking, a bug can fall into a MLC bug when any of the 3 steps is violated. If memory allocation and free functions are improperly used, an incorrect source-sink case is hit. Meanwhile, a missing check happens when a MAR is dereferenced without verification. And if certain memory space is left unfreed when out of use, we may come across a lacking sink case.

As we can figure out from the above examples, MLC bugs are distinct from common memory vulnerabilities, such as out-of-bound (OOB) and use-after-free (UAF). Generally speaking, MLC bugs are distinguishable from kernel memory errors [11], [19]–[23], API misuse bugs [24]–[27] and missing check bugs [15], [28], [29]. Exposing bugs such as OOB and UAF is focusing on the result of accessing memory by error. On the contrary, MLCSan targets the causes of memory errors and recognizes which step in the life-cycle of memory space is error-prone. Furthermore, previous works on API usage checking only consider single APIs, and contrarily we concentrate on coexisting source-sink pairs and incorporate detecting function absence in kernel memory to reason about potential memory leaks, which is different from API misuse bugs. And MLC bugs cannot be classified into missing check bugs for the reason the initial intention of MLCSan is detecting the violations to allocate, dereference, and free of memory space, which is far more different from concentrating on missing check of critical variables.

MLC bugs can be common in commodity OS kernels and they are capable of bringing out trouble in the system. Generally speaking, memory operations such as allocation or free widely exists in OS kernels. For instance, according to our evaluation, there are 13,422 allocation operations in Linux 4.12. Moreover, MLC bugs exist widely in kernels since all memory space accords with the 3 steps in a life-cycle as mentioned above, and any violation to the steps will result in potential MLC bugs. For example, the number of LLVM instructions operating on a certain MAR can reach 1,930 as demonstrated in Section V and programmers can hardly guarantee that there is no mis-operation in such a large scale.

Detecting MLC bugs in OS kernels can be laborious and challenging. Conducting static analysis on kernels can be non-trivial and demands customized techniques to efficiently analyze such a huge code base and handle exceptional circumstances. Second, a MLC bug is a new type of bugs, we need a clear and specific bug definition, but we can rarely find typically related works. Moreover, it is difficult to implement the MAR related rules, e.g., the identification of source functions, because there is no readily available approaches to determine whether a function is a source in numerous function definitions in kernels. At last, since MLCSan depends on data-flow and control-flow analysis traversing multiple functions, accurate inter-procedural analysis is necessary in the detection of MLC bugs.

In this paper, to overcome the above mentioned challenges to discover MLC bugs, we first summarize the life-cycle of allocated memory space, which includes allocation, dereference, and free. More importantly, we give a formal definition of source, sink, MAR, and MLC bugs. Moreover, by utilizing context-, flow- and field-sensitive program analysis techniques, we propose an automated and scalable detection framework, MLCSan, to expose MLC bugs. By leveraging classical call graph construction and pointer analysis in an inter-procedure manner, iterative and propagative detection techniques are adopted to locate all the sources, sinks and MARs in the huge code base of OS kernels. After the identification of them, MLCSan constructs define-use chains of MARs and performs bidirectional source-sink analysis, taking both the source and sink of a certain memory into consideration. And MLC bugs are detected by the above procedures in MLCSan.

Based on the new feature of LLVM [35], we implement MLCSan framework and analyze the latest mainline Linux

and FreeBSD kernels with it. The experiment results indicate that the entire analysis process of the kernels is finished in two hours and 41 new MLC bugs are identified. Specifically, we submitted 41 patches of all the bugs to the maintainers, most of which are confirmed or applied to the Linux and FreeBSD kernel. This is strong proof that MLC bugs do exist commonly in OS kernels, on which we need to focus to prevent potential security issues. Furthermore, the results also demonstrate that MLCSan can automatically and scalably detect MLC bugs in an effective way.

In conclusion, this paper makes the following contributions:

- **A new type of software bugs.** We come to a definition of a new type of software bugs, which is MLC bug. By examining the violations to allocate, dereference and free of memory space, we conclude incorrect source-sink, missing check and lacking sink as three sub-classes.
- **Automated and scalable analysis of MLC bugs.** In this paper, we propose MLCSan, an automated and scalable detection framework of MLC bugs. To start with, MLCSan performs inter-procedure global call graph and pointer analysis, followed by iteratively detecting both source functions and MARs. Moreover, we incorporate extended define-use chain analysis to detect the three cases in MLC bugs, and evaluation demonstrates that these techniques are effective and can be scaled to different platforms such as Linux and FreeBSD.
- **Identification of 41 MLC bugs.** We evaluated the latest mainline Linux kernels and FreeBSD, and submitted 41 patches for MLC bugs to maintainers, which are capable of compromising the entire system and causing security problems.
- **MLCSan static analysis prototype.** To make contributions to the security community and foster more work in this field, we will open source MLCSan.[1]

II. MLC BUGS

In this section we will discuss what is source function, sink function and MAR, and then give a formal definition of incorrect source-sink, missing check, lacking sink and MLC bugs.

A. A GLIMPSE AT MLC BUGS

As discussed above, memory space in kernel is mostly allocated by sources, dereferenced and freed by sinks. In summary, the life-cycle of a MAR can be concluded as following:

- **Allocation.** A MAR is allocated by a source function. At the very beginning, the usage of this source must be appropriate.
- **Dereference.** Then, before using this MAR, it should be verified to prevent potential null pointer dereference. After that, it is dereferenced.

[1] https://github.com/zhanggenex/MARC

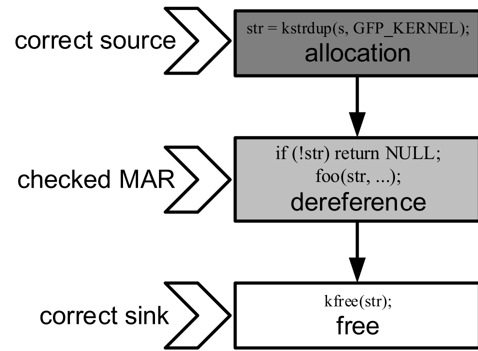


FIGURE 4. 3 steps of the life-cycle of memory space.

```
1  static void flush_cache_ent(struct dfs_cache_entry *ce)
2  {
3      ce->ce_path = kstrdup_const(path, GFP_KERNEL);
4  -   kfree(ce->ce_path);
5  +   kfree_const(ce->ce_path);
6      free_tgts(ce);
7  }
8  //fs/cifs/dfs_cache.c
```

FIGURE 5. Part of an applied patch file. In this patch, MAR *ce→ce_path* is freed correctly, rather than *kfree()*.

TABLE 1. Preliminaries.

Name	Definition
F	function
V	variable
SC	source
SK	sink

- **Free.** Finally, we need to free this memory space in a right way if it is out of use.

In brief, Figure 4 shows a simplified life-cycle of MAR *str*, which is allocated by a correct source *kstrdup()*, dereferenced after checked and freed properly by *kfree()*.

Generally speaking, a MLC bug may arise if any of the above three steps goes wrong. When a MAR is allocated or freed incorrectly, as shown in Figure 1 and 5 accordingly, an incorrect source-sink scenario will happen. Furthermore, Figure 2 indicates a missing check subclass of MLC bugs, where a MAR is not verified when dereferenced. Similarly, when the free operation of a certain MAR is not performed, a lacking sink case is hit in Figure 3.

B. DEFINITION DETAIL

By referring to the above simplified examples, we come to the detailed formal definition of source function, sink function, MAR and MLC bugs.

- **Definition 1.** Preliminaries in Table 1.
- **Definition 2-1.** $\{SC\}$ is the set of all source functions. At the beginning, it is equal to $\{SC_o\}$, which is the original source function set, containing memory

TABLE 2. Original source functions.

Name	Description
kmalloc()	the normal method of allocating memory for objects smaller than page size in the kernel
kzalloc()	allocate memory, the memory is set to zero
kcalloc()	allocate memory for an array, the memory is set to zero
krealloc()	reallocate memory, the contents will remain unchanged
vmalloc()	allocate virtually contiguous memory
vzalloc()	allocate virtually contiguous memory filled with zero
kmem_cache_alloc()	allocate an object from cache

TABLE 3. Original sink functions.

Name	Description
kfree()	free previously allocated memory
vfree()	release memory allocated by vmalloc()
kmem_cache_free()	deallocate an object

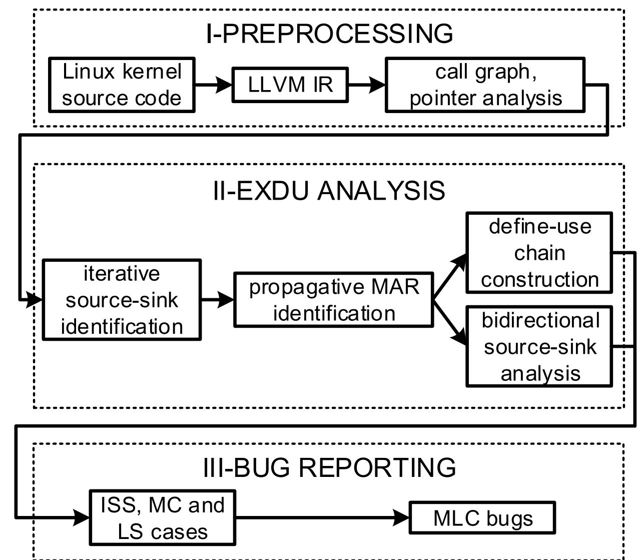


FIGURE 6. A demonstration of the primary work flow of MLCSan, containing 3 main procedures: preprocessing, extended define-use chain analysis and bug reporting in the end.

allocation functions of Linux and FreeBSD kernels,[2] such as *kmalloc()* and *vmalloc()*. Table 2 demonstrates original source functions leveraged in this paper.

- **Definition 2-2.** (1) When a function F calls a source F' ($F' \in \{SC\}$) in its function body, (2) and the return-value of F is a pointer variable, then we claim that $F \in \{SC\}$.
- **Definition 3-1.** Similar to Definition 2-1, $\{SK\}$ is the set of sink functions. $\{SK_o\}$ denotes the original sinks, which are described in Table 3.
- **Definition 3-2.** If a function F calls F' ($F' \in \{SK\}$) in its function body, then we claim that $F \in \{SK\}$.
- **Definition 4.** The set of MARs is $\{MAR\}$. If a pointer variable V is the return-value of a source function, then $V \in \{MAR\}$.
- **Definition 5.** If a MAR is incorrectly allocated or freed with an improper source or sink, then this case falls into incorrect source-sink. If a MAR is dereferenced without verification, then it is a missing check situation. Lacking sink happens if a MAR is not freed when out of use. These 3 subclasses are collectively called MLC bug.

III. DESIGN OF MLCSAN

As the definition detail of source function, sink function, MAR and MLC bugs is discussed above, we will propose the design of MLCSan in this section.

A. OVERVIEW OF MLCSAN

Figure 6 shows the basic work flow of MLCSan, consisting three key stages: preprocessing, extended define-use chain analysis and bug reporting.

[2]The implementation of these functions is different in Linux and FreeBSD.

MLCSan treats LLVM IR as its inputs, rather than kernel source code, which is an intermediate representation compiled by clang [36]. To perform inter-procedure analysis, MLCSan then builds a global call graph of functions in kernel. Besides, further pointer analysis is utilized to aid other memory related analysis in the following steps, e.g., alias analysis.

Stage II is the most significant component of the entire structure. This stage contains, extended define-use chain (ExDU) analysis: iterative source-sink identification, propagative MAR identification, define-use chain construction and bidirectional source-sink analysis. In this phase, sources, sinks and MARs will all be detected by ExDU, then it constructs a frequently used define-use chain and analyzes both sources and sinks.

In stage III, MLC bugs will be reported as output. By tracking the life-cycle of memory space, MLCSan can tell which part of the process is mistaken. Specifically, the content of these three stages will be proposed in the following subsections.

B. STAGE I - PREPROCESSING

LLVM compiler clang can compile kernel source code to LLVM IR, of which the file name always has a *.bc* suffix. After receiving LLVM IR as input, MLCSan then builds a global call graph of functions in source code to conduct inter-procedure analysis. As defined above, MLCSan iteratively tracks sources and sinks, which needs to reason about direct calls and indirect calls between functions, e.g., function *efi_call_phys_prolog()* in Figure 7 iteratively calls *kmalloc_array()* and *kmalloc()*, and it can be detected as a source function through the following stages.

```
1 static efi_status_t phys_efi_set_virtual_address_map(...
2 {
3     save_pgd = efi_call_phys_prolog();
4 +   if (!save_pgd)
5 +       return EFI_ABORTED;
6     efi_call_phys_epilog(save_pgd);
7 }
8 //arch/x86/platform/efi/efi.c
```

FIGURE 7. Part of an applied patch file. In this patch, MAR ***save_pgd*** is allocated by source ***efi_call_phys_prolog()***, and a check before dereferenced is enforced.

Likewise, pointer analysis is also conducted in this stage to help detect alias of MARs in the next stage. As shown in Figure 3, MAR *val* is allocated by *kmemdup_nul()*, and is used in function *selinux_add_opt()*. Further analysis into this function is required to verify whether there *val* is dereferenced, and pointer analysis can assist revealing this problem.

C. STAGE II - EXDU ANALYSIS

ExDU contains four basic parts: iterative source-sink identification, propagative MAR identification, define-use chain construction and bidirectional source-sink analysis.

To start with, iterative source-sink identification needs to traverse the global call graph to detect callers of functions in $\{SC_o\}$ and $\{SK_o\}$. And if a function F calls directly or indirectly another function in source set $\{SC\}$ and the return-value of F is a pointer variable, then we will add this function to $\{SC\}$ and iteratively handle all the functions in kernel in the same way.

Moreover, MARs are identified through a propagative MAR identification technique, which is accomplished mainly by alias analysis. For example, in Figure 3, MAR *val* is allocated by *kmemdup_nul()*. Furthermore, *val* is used later in function *selinux_add_opt()*, whose implementation is showed in Figure 8, and we need to figure out whether *val* is dereferenced. Apparently, *s* is the alias of *val* and it is dereferenced, thus *val* and *s* should both be checked before dereferenced. In this propagative manner, MLCSan can detect potential MARs hard to identify and eliminate false negatives as much as possible. Specifically, there should be a suspend condition to stop the alias propagation, otherwise tracking numerous alias will cause significant performance overhead and may report unnecessary false positives. In MLCSan, alias analysis is designed to suspend when a user-space variable, a global variable, etc. is met, which is difficult to perform deeper analysis.

```
1 static int selinux_add_opt(int token, const char *s, ...)
2 {
3     if (!s)
4         return -ENOMEM;
5     opts->context = s;
6 }
7 //security/selinux/hooks.c
```

FIGURE 8. Code of ***selinux_add_opt()***, in which ***s*** is the alias of ***val***.

Constructing define-use chain [37] of a variable is widely used in data-flow analysis. By combining it with a bidirectional source-sink analysis, MLCSan can reason about incorrect source-sink, missing check and lacking sink cases. Along the define-use chain, it will generate a series of marks related to the LLVM instructions operating on a MAR, e.g., there is a store instruction *store i32 %1, i32* %ptr, align 4*, a **Store** mark is recorded in MLCSan for the define-use chain of *ptr*. And other instructions, such as Load, GEP and Call, work in the same way. Figure 10 illustrates a C code scratch and the relative LLVM IR, in which *ptr* is allocated by *kmalloc()*, checked, dereferenced in *printf()* and finally freed, thus the define-use chain is *Alloc→Cmp→GEP→Free*.

Additionally, in oder to detect incorrect use of both sources and sinks or cases where sink functions are missing, we leverage a bidirectional source-sink analysis. When a pair of source and sink does not match previously defined rules, an incorrect source-sink case is hit, e.g., in Figure 5, *ce→ce_path* is allocated by *kstrdup_const()*, with an annotation "Strings allocated by kstrdup_const should be freed by kfree_const" in kernel source code. Therefore, *ce→ce_path* should be freed by *kfree_const()*, rather than *kfree()*. As for lacking sink scenarios, MLCSan traverses the global call graph of the function where an allocated MAR resides in, and if there is no sink for the MAR in all the direct or indirect callers of this function, a lacking sink case will be detected. As *mnt_opts* in Figure 3, a sink function *selinux_free_mnt_opts()* is enforced to handle this lacking sink case.

As illustrated in Algorithm 1, the above discussed iterative source-sink identification, propagative MAR identification, define-use chain construction and bidirectional source-sink analysis take initial set of source functions $\{SC_o\}$and sink functions $\{SK_o\}$ as inputs, and output incorrect source-sink, missing check and lacking sink cases respectively.

D. STAGE III - MLC BUG REPORTING

According to the life-cycle of memory space depicted in Figure 4, any violation of this life-cycle will be reported by MLCSan. Incorrect sources and sinks will be detected as incorrect source-sink, such as *kstrndup()* and *kfree()* in Figure 1 and 5, which should be *kmemdup_nul()* and *kfree_const()* instead to match their respective source and sink functions. A missing check case happens when a MAR is dereferenced without verification, and null pointer dereference will commonly cause system crash or denial-of-service. As shown in Figure 2 and 7, checks for *node_info→vdev_port.name* and *save_pgd* are enforced before dereferencing MARs to prevent missing check cases. Moreover, Figure 3 demonstrates that MLCSan is capable of discovering lacking sink cases, in which the sink for *mnt_opts*, *selinux_free_mnt_opts()*, is executed. And by freeing this allocated memory space, potential memory leaks are prevented.

Figure 11 in Appendix is the output of MLCSan on a missing check case. As illustrated, most of the significant information is given in the output, such as the LLVM

Algorithm 1 ExDU Analysis

```
function iterative_source_sink_identification
    {SC} = {SC_o}
    {SK} = {SK_o}
    for F in CallGraph do
        if F calls F'  &  F'∈{SC}  &  F returns pointer then
            {SC} = F∪{SC}
        end if
        if F calls F'  &  F'∈{SK} then
            {SK} = F∪{SK}
        end if
    end for
end function
function propagative_marv_identification
    {MARV} = Ø
    for F in {SC} do
        if F returns V then
            {MARV} = V∪Alias(V)∪{MARV}
        end if
    end for
end function
function constuct_define_use_tree
    Output: DUT
end function
function bidirectional_source_sink_analysis
    Output: P1, P2
end function
function bug_reporting(DUC, P1, P2)
    {ISS} = {MC} = {LS} = Ø
    if iss ∈ (P1, P2) then
        {ISS} = iss ∪ {ISS}
    end if
    if mc∈DUC then
        {MC} = mc ∪ {MC}
    end if
    if ls∈DUC then
        {LS} = ls ∪ {LS}
    end if
end function
Output: {ISS}, {MC}, {LS}
```

instruction, source, sink, MAR, define-use chain, etc. Moreover, source code around the bug site is presented to ease further debugging. We can identify from the output that the MAR is not checked before dereferencing in *strncmp()* and a check should be enforced to prevent potential null pointer dereference.

IV. IMPLEMENTATION DETAILS

This section will present implementation details of MLCSan, as well as several engineering issues related to static analysis with LLVM infrastructure and the solutions. MLCSan utilizes LLVM and clang version 7.0.0 as static analysis and compiler engine. OS kernel source code is firstly compiled to LLVM IR with clang, and then static analysis is performed upon it to expose MLC bugs. In total, MLCSan includes about 12K lines of code and 6 individual LLVM passes to conduct the 3-stage operation described in Section III.These passes construct global call graph and perform pointer analysis in advance, to aid the ExDU analysis in the following stage. And more detailed information will be given below.

A. STAGE I - PREPROCESSING

1) COMPILING LLVM IR

To start with, kernel source code needs to be compiled to LLVM IR, acting as inputs of the LLVM analysis passes. Older versions of Linux kernel can be directly compiled to LLVM IR using clang. However, asm-goto is enforced as a compiler prerequisite to guarantee the absence of dynamic branches in later Linux versions [38], and to the time of conducting experiment of this paper, clang does not support asm-goto yet [39]. In this paper, we leverage the compiling steps in [40] to successfully generate LLVM IR. In general, it bypasses asm-goto and produces LLVM bitcode files with clang, of which we will not further discuss since it is not related to our research topic. Besides, MLCSan accepts this input and continues to perform following analysis steps.

2) GLOBAL CALL GRAPH

The iterative source-sink identification in MLCSan requires a global call graph to conduct inter-procedure analysis between functions. Starting from $\{SC_o\}$ and $\{SK_o\}$, we can figure out all the sources and sinks in the kernel with such global call graph. For example, *efi_call_phys_prolog()* in Figure 7, as a matter of fact, it iteratively calls *kmalloc_array()* and *kmalloc()* in its implementation, which we need a global call graph to reveal the relationship between them, and indirect calls in some difficult cases. In MLCSan, we implement this part by following previous works, such as [12], [41], [42].

3) POINTER AND ALIAS ANALYSIS

In MLCSan, we need to identify MARs in a propagative manner, thus pointer and alias analysis is required. For example, in Figure 3, *val* is allocated by *kmemdup_nul()*, which is used later in function *selinux_add_opt()*, whose implementation is showed in Figure 8. *s* is the alias of *val* and it is dereferenced, therefore they should be checked before dereferenced. We leverage off-the-self alias analysis pass in LLVM and to balance between false negative rate and performance overhead, we study the usage in [12], [15] and finally consider "PartialAlias" and "MustAlias" as alias.

B. STAGE II - EXDU ANALYSIS

1) DEFINE-USE CHAIN AND GEP INSTRUCTION

After identifying MARs, we need to track all the operations on them by a define-use chain [37], e.g., the define-use chain of *ptr* in Figure 10 is *Alloc*→*Cmp*→*GEP*→*Free*. And since we discover that in our evaluation that there is always

a *Load* instruction coupling with significant operations on a MAR, for example, Line 10 in front of Line 11 in Figure 10, therefore we eliminate the *Load* in our define-use chain representation for simplicity.

GEP is a fundamental pointer calculation instruction in LLVM. In our implementation, we consider GEP instructions as memory dereferencing operations for the following reasons. GEP calculates the offset of the target memory and *Load* it, which is the same as dereferencing a pointer in C code, e.g., in Figure 10, *ptr* is dereferenced in *printf()* in Line 5 with C code and the same in Line 20 with LLVM GEP instruction.

2) BIDIRECTIONAL SOURCE-SINK ANALYSIS

We leverage bidirectional source-sink analysis to detect incorrect source-sink and lacking sink cases. We predefine several source-sink matches after examining the kernel, such as *kstrdup_const()* and *kfree_const()* in Figure 5. If any violation to the matches happens, MLCSan will report an incorrect source-sink case. Moreover, by traversing the define-use chain of a MAR, MLCSan can tell whether a sink function is operated. When a MAR is left unfreed after use, a lacking sink case is reported.

C. STAGE III - BUG REPORTING

1) REDUCING FALSE POSITIVES

In this step, we adopt several methods to eliminate false positives. Firstly, we add a terminate condition to stop the alias propagation, otherwise it will cause considerable performance overhead and may cause false positives. In MLCSan, the propagation is designed to suspend when a user-space variable, a global variable, etc. is hit, which is difficult to perform deeper analysis. Furthermore, we exclude some functions that iteratively call a source function but not sources themselves, e.g., function F_1 calls *kmalloc()*, allocates memory for p_1 and frees it finally. However, the return-value of F_1 is p_2, which is allocated by F_2 and F_2 is already added to the source set. In this case, we exclude F_1 to increase the detection accuracy. Moreover, we consider some practical problems. One of them is that some functions in kernels have a *__init* attribute, and according to the maintainers, if memory allocation in this kind of function fails, the whole system will reboot as a result, therefore we exclude them from our reported bugs at last. We also handle other cases such as *kmalloc()* with *GFP_NOFAIL*, etc.

V. EVALUATION

We will discuss the experiment results in the following aspects.

- **Analysis statistics,** including analysis time and statistics (Section V-A and V-B).
- **Effectiveness of bug finding.** MLCSan should be effective in exposing MLC bugs (Section V-C).
- **Portability and scalability.** Evaluation should confirm MLCSan can scale to different targets (Section V-E).

We conduct extensive experiments on MLCSan with three different version of Linux kernels and one version of FreeBSD, to evaluate both scalability and effectiveness of our framework. Moreover, our evaluation is performed on a laptop with 4 cores (Intel(R) Core(TM) i7-7500 CPU @ 2.70GHz), 8.0 GB RAM, and the distribution is ubuntu 16.04 LTS with kernel 4.4. Specifically, all the three versions are configured with default configuration *defconfig* and complied with LLVM/clang 7.0.0.

A. ANALYSIS TIME

Table 4 is the analysis time and related information on each version of kernels. Linux 5.1 and FreeBSD 11.3 are the newest versions at the time of our experiments. As we can see from the table, Linux kernel 3.19, 4.12 and 5.1 are released between at about a two-year interval, therefore we choose them as our targets. MLCSan succeeds in analyzing all of the four kernels, which is a solid proof that MLCSan is scalable on different targets.

TABLE 4. A list of analysis time on the four kernels, where Date is the release date of the kernel, LOC indicates lines of code, IR Size represents the size of the bitcode files compiled by clang, and Time implies the analysis time in seconds.

Version	Year	LOC	IR Size	Time
Linux 3.19	2015	17.7M	280MB	3.4K
Linux 4.12	2017	26.5M	364MB	5.0K
Linux 5.1	2019	27.1M	379MB	5.2K
FreeBSD 11.3	2019	20.1M	293MB	3.9K

More information is provided in the table, and the lines of code are proportional to the size of bitcode files compiled by clang. Besides, MLCSan accomplishes the bug detection task between 3,400 and 5,200 seconds for different kernels, which is approximately between 1 hour and 2 hours. This is also under the intuition that the more code in kernel, the longer it takes to thoroughly analyze it.

B. ANALYSIS STATISTICS

Analysis statistics of the three kernels are given in Table 5. In this table, the important statistics in our ExDU analysis such as source and sinks are presented. By examining the table, we can find that the number of sources, sinks and MARs is also generally in proportion to the code size of each kernel. Taking Linux kernel 4.12 as an example, the number of source functions is 2,880, which is the same order of magnitude with the number of sink functions.

The last column in Table 5 is quite interesting, and we will have further discussion on it. This column records the average length of define-use chain of MARs in each kernel, which indicates the number of LLVM instructions operating on a certain MAR from allocation to free. Additionally, as showed in Figure 9, we present the define-use chain occurrences distribution graph of MARs in kernel 4.12. The x axis is the length of define-use chain and the y axis is the number

TABLE 5. A list of analysis statistics analyzing the 4 kernels. SC, SK and MAR is the total number of them detected in kernels, and Avg. DU is the average length of define-use chain in each kernel.

Version	SC	SK	MAR	Avg. DU
Linux 3.19	2411	2017	10010	18.1
Linux 4.12	2880	2539	13422	19.3
Linux 5.1	2904	2596	13752	19.4
FreeBSD 11.3	2622	2519	11007	19.1

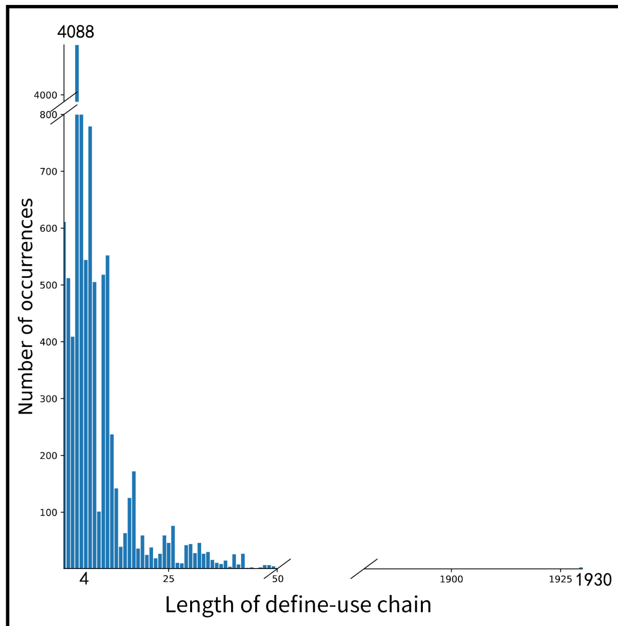


FIGURE 9. A graph of define-use chain length occurrences in Linux 4.12, where the x axis is the length of define-use chain, and y axis is the number of occurrence of the respective length.

of occurrences of this length, e.g., (10, 21) in this graph means there are 21 define-use chains of length 10 in kernel 4.12. More interestingly, the maximum define-use chain length reaches 1,930 in 4.12, which indicates that about two thousand LLVM instructions are operating on this MAR. And when we focusing on the maximum of y axis (4, 4088), which means that define-use chain of length 4 appears most frequently in this kernel. By considering the common define-use chain of a MAR, this would be more easy to understand. According to the life-cycle of memory space, allocation, checked dereference and free can be measured by define-use chain in MLCSan, which is $Alloc \rightarrow Cmp \rightarrow GEP \rightarrow Free$. So a define-use chain of length 4 is a typical operation on a MAR, therefore it makes the maximum occurrences in the graph.

C. EXPOSED BUGS

Table 6 illustrates an example of bugs discovered by MLCSan. In the detailed Table 7 and Table 8 in Appendix, among the 41 identified MLC bugs, we have 17 of them applied, 3 confirmed by maintainers and the rest bugs are submitted but not accepted. The reason for the rejection is mainly kernel-related, e.g., item 21 in Table 7 is a missing

TABLE 6. An example of bugs exposed by MLCSan. SC/SK represents the sources and sinks. And Type indicates types of MLC bugs. Status demonstrates the status of corresponding patch, where A is already applied.

File	Function	SC/SK	MAR	Type	Status
ipv6_sockglue.c	ip6_ra_control()	kmalloc()	new_ra	MC	A

check case discovered by MLCSan, in which s is allocated by *kmem_cache_zalloc()* and a check is absent before dereferencing it. We patched this bug with a check for s and an *ENOMEM* return-value. However, the maintainer argued that because the function where this allocation happens has a *__init* attribute, an allocation failure will result in a system reboot, so it cannot cause harmful results. Typically, this kind of consideration is closely related to the kernel low-level implementation, which is actually out of our research scope in this paper. MLCSan accomplishes detecting these MLC bugs as what is designed for.

In total, there are 5 incorrect source-sink, 14 missing check and 11 lacking sink cases of MLC bugs in the detailed table. It is worth noting that incorrect source-sink is less than missing check and lacking sink cases, because it rarely happens when programmers write a incorrect function in code, such as *kfree_const()*, in most cases kernel developers use it correctly to free a const variable.

When considering the second column of Table 7, we can find that 13 bugs are in the driver code, which is a widely accepted conclusion that codes in drivers tend to be more buggy [1], [3], [10], [13] since they are maintained by different vendors and the quality of code cannot be easily guaranteed. However, MLCSan still succeeds in discovering MLC bugs in some key components of Linux kernel, such as item 1 in *net* subsystem, item 10 in *arch* subsystem, item 14 in *security* subsystem, etc.

Crix [15] is a static analyzer targeting at missing-check bugs in OS kernels. The last column indicates whether it can detect the same bug by MLCSan. As we can see from the table, Crix can expose missing-check bugs, but fails to detect incorrect source-sink and lacking-sink bugs. The reason is that MLCSan takes allocation, dereference, free and their relations into consideration. Crix only accounts for one stage of the memory life-cycle and it can only detect missing-check in MLC bugs.

D. CASE STUDY

In total, among the exposed 41 bugs, there are 7 incorrect source-sink cases and 19 missing check with 15 lacking sink cases. For example, Figure 5 and 1 (item 12 and 7 in Table 7) belong to incorrect source-sink, where the source or sink function is improperly used for a MAR. Apparently, *kfree_const()* matches *kstrdup_const()* and *kmemdup_nul()* matches *kfree()*, and mis-matched sources and sinks will be detected by MLCSan. By applying bidirectional source-sink analysis, incorrect use of both source and sink functions is identified as breaking the predefined source-sink pairs.

TABLE 7. A list of bugs exposed by MLCSan in Linux. SC/SK represents the sources and sinks. And Type indicates types of MLC bugs, including ISS, MC and LS. Status demonstrates the status of 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. Crix indicates whether this bug can be detected by Crix.

ID	File	Function	SC/SK	MAR	Type	Status	Crix
1	net/.../ ipv6_sockglue.c	ip6_ra_control()	kmalloc()	new_ra	MC	A	√
2	drivers/.../ consolemap.c	con_insert_unipair()	kmalloc_array()	p2	LS	A	×
3	arch/.../efi/efi.c	phys_efi_set_virtual_ address_map()	efi_call_phys_ prolog()	save_pgd	MC	A	√
4	arch/.../efi_64.c	efi_call_phys_prolog()	kmalloc_array()	save_pgd	MC	A	√
5	net/.../ip_sockglue.c	ip_ra_control()	kmalloc()	new_ra	MC	A	√
6	drivers/../ drm_edid_load.c	drm_load_edid_ firmware()	kstrdup()	fwstr	MC	A	√
7	sound/.../wcd9335.c	wcd9335_codec_ enable_dec()	kstrndup_nul()	widget_name	ISS	A	×
8	drivers/.../clk-sunxi.c	sunxi_divs_clk_setup()	kstrndup()	derived_name	MC	A	√
9	drivers/.../ mpt3sas_ctl.c	_ctl_ioctl_main()	kmalloc()	ioc_number	MC	A	√
10	arch/.../dlpar.c	dlpar_parse_cc_ property()	kstrdup()	prop→name	MC	A	√
11	arch/.../mdesc.c	get_vdev_port_node_ info()	kstrdup_const()	node_info→ vdev_port.name	MC	A	√
12	fs/.../dfs_cache.c	flush_cache_ent()	kfree_const()	ce→ce_path	ISS	A	×
13	fs/.../dfs_cache.c	alloc_cache_entry()	kfree_const()	ce→ce_path	ISS	A	×
14	security/.../hooks.c	selinux_sb_eat_lsm_ opts()	kmemdup_nul()	arg	LS	A	×
15	security/.../hooks.c	selinux_add_mnt_opt()	kmemdup_nul()	val	LS	A	×
16	drivers/.../dm-init.c	dm_parse_table_entry()	kstrndup()	dev→ target_args_array[n]	ISS	A	×
17	drivers/.../dm-init.c	dm_init_init()	kstrndup()	str	ISS	A	×
18	drivers/.../tegra-hsp.c	tegra_hsp_doorbell_ create()	devm_kstrdup_ const()	db→name	MC	C	√
19	drivers/.../vt.c	con_init()	kzalloc()	vc	LS	C	√
20	drivers/.../vt.c	con_init()	kzalloc()	vc→vc_screenbuf	LS	C	×
21	mm/slub.c	bootst()	kmem_cache_ zalloc()	s	MC	N	√
22	init/initramfs.c	dir_add()	kstrdup()	de→name	MC	N	√
23	drivers/.../tty_io.c	alloc_tty_struct()	tty_get_device()	tty→dev	LS	N	×
24	sound/.../hdac_sysfs.c	kobject_create_and_add()	kcalloc()	tree→nodes	LS	N	×
25	drivers/.../ dm-region-hash.c	__rh_alloc()	kmalloc()	nreg	MC	N	√
26	drivers/.../sg.c	sg_write()	cmnd()	opcode	MC	N	√
27	drivers/.../ wlcore/sdio.c	wl1271_probe()	devm_kzalloc()	glue	LS	N	×
28	drivers/.../ wlcore/spi.c	wl1271_probe()	kzalloc()	glue	LS	N	×
29	sound/.../ pcm030-audio-fabric.c	pcm030_fabric_probe()	platform_device_ alloc()	pdata→ codec_device	LS	N	×
30	sound/.../ tegra_wm9712.c	tegra_wm9712_driver_ probe()	platform_device_ alloc()	machine→codec	LS	N	×

Moreover, as for missing check cases, Figure 2 and 7 (item 11 and 4 in Table 7) demonstrates that MLCSan is capable of discovering dereference of MARs without check for validity. As we construct define-use chain of a MAR and track all the operations on it, the life-cycle of the MAR is examined: where the allocation is, and when the check or dereference happens.

When considering lacking sink cases, Figure 3 and 12 (item 15 and 20 in Table 7) are among the detected lacking sink bugs by MLCSan. By examining the absence of sink functions, we can reason about what should be enforced. For example, *selinux_free_mnt_opts()* and *kfree()* are enforced and in addition, *console_unlock()* is added to the error handling code to unlock the console.

Besides, the SC/SK column demonstrates the respective source function and sink functions, among which 8 are in the original source set $\{SC_o\}$ or the original sink set $\{SK_o\}$, such as *kmalloc()* and *kzalloc()*. The iterative source-sink identification analysis we leverage in MLCSan can detect other functions outside the original sets as designed to, e.g., item 3 in the table has *efi_call_phys_prolog()* as the source, and the function call order is *efi_call_phys_prolog()*, *kmalloc_array()*, and finally *kmalloc()*.

E. PORTABILITY AND SCALABILITY

Generally speaking, the analysis techniques adopted in MLCSan, such as call graph construction, ExDU analysis, etc. are not only suitable to Linux or FreeBSD. The life-cycle

TABLE 8. A list of bugs exposed by MLCSan in FreeBSD. SC/SK represents the sources and sinks. And Type indicates types of MLC bugs, including ISS, MC and LS. Status demonstrates the status of corresponding patch, where A is already applied, C is confirmed and accepted by maintainers but not applied to the time of writing this paper and N is submitted. Crix indicates whether this bug can be detected by Crix.

ID	File	Function	SC/SK	MAR	Type	Status	Crix
1	sys/.../ mlx5_fs_tree.c	_fs_add_node()	kstrdup_const()	node→name	MC	A	√
2	sys/.../ altera_sdcard.c	altera_sdcard_ attach()	taskqueue_ create()	sc→as_taskqueue	LS	A	×
3	sys/.../ mlx4_en_netdev.c	mlx4_en_ init_netdev()	kzalloc()	priv	MC	A	√
4	sys/../ qlnxr_cm.c	qlnxr_create_ gsi_qp()	kfree()	qp	ISS	C	×
5	sys/.../ qlnxr_verbs.c	qlnxr_init_ user_queue()	qlnxr_alloc_ pbl_tbl()	q→ pbl_tbl	LS	C	×
6	sys/.../ qlnxr_os.c	qlnxr_add()	kzalloc()	dev→pdev	MC	C	√
7	sys/.../ qlnxr_verbs.c	qlnxr_create_ kernel_qp()	kfree()	qp	ISS	C	×
8	sys/.../ mlx5_qp.c	mbox_alloc()	kzalloc()	mbox→in	LS	N	×
9	sys/.../ mlx5_main.c	mlx5_enable_ msix()	kzalloc()	priv→msix_arr	MC	N	√
10	sys/.../ aw_cir.c	aw_ir_attach()	evdev_alloc()	sc→sc_evdev	LS	N	×
11	sys/.../ cloudabi_vdso.c	cloudabi_vdso_ init()	kva_alloc()	addr	MC	N	√

of memory space also exits in other programs, e.g., *malloc()* and *free()* in C programs. By replacing the original source function and sink function set, MLCSan can detect sources and sinks in other programs. And the define-use chain construction and bidirectional source-sink analysis can also be applied to other cases with slight modification.

Besides, owing to the well-designed modularization of LLVM and clang, we can compile programs into bitcode files in other mainstream platforms, such as Windows and MacOS. Since MLCSan only takes bitcode files of the target program as input, theoretically we are able to analyze other platforms as well.

VI. DISCUSSION

As demonstrated in evaluation, by leveraging global call graph, ExDU analysis, etc. MLCSan is effective in automatically and scalably detecting MLC bugs. However, in this section, we will present intrinsic limitations of MLCSan and the possible improving methods as well.

A. LIMITATIONS AND POSSIBLE IMPROVEMENTS

1) FALSE POSITIVES

Similar to most static analysis tools, MLCSan bears an acceptable percentage of false positives. According to the evaluation result on Linux and FreeBSD, MLCSan can detect 41 MLC bugs in total, which is the examined result of 73 reports. With a false positive rate of 56.1%, we believe that this is acceptable in static analysis tools, especially designed for analyzing OS kernels. For instance, LRSan [12] reports over two thousand lacking-recheck cases, and 19 bugs are identified after manual verification. Compared to it, false positive rate is much lower in MLCSan.

(1) Incomplete call graph and pointer analysis. Honestly speaking, the accuracy of call graph construction and pointer analysis is always a challenge for static analysis tools, such as indirect calls between functions and alias analysis of variables. MLCSan tries to handle this problem and more than half of the false positives are caused by this. For example, for a lacking sink case, we have 3 functions f_1, f_2 and f_3. f_1 calls f_2 and then f_3, where variable *ptr* is allocated in f_2 and freed in f_3. This situation is challenging to cope with since relation between f_2 and f_3 can hardly be measured. This problem may be handled by more accurate call graph construction and thorough reasoning about allocation and free sites of a certain variable.

(2) Kernel implementation related issues. Some cases are hard to identify simply by defining MLC bug patterns as discussed above, e.g., some null pointer dereferences are allowed in Linux kernel to check whether the system is functioning properly. Therefore similar patches are not submitted to Linux community and excluded from our identified bugs. Furthermore, there is a managed memory allocation function named *devm_kmalloc()* and the Linux documents introduce that, memory allocated by it is automatically freed on driver detach. Since these implementation and semantic related issues are out of our research scope, we also consider these reports as false positives in this paper. (However, there are over 50 cases where memory allocated by it is manually freed by the relative free function in Linux kernels, which is contrary to the documents.) By utilizing light weight semantic analysis, we may be able to avoid the above issues and make some improvements.

(3) Others. There are other causes of false positives as well, e.g., complicated programing logic, unreachable call graphs,

imprecise static analysis methods, etc. In total, these causes form about 10% of all the false positives.

2) FALSE NEGATIVES

Though MLCSan leverages several well-designed static analysis approaches, there are still potential false negatives. First, our kernel is configured with *defconfig*, in which some subcomponents may be absent compared to *allyesconfig*. Those missing parts are mainly codes of other architectures, and MLCSan only examines default configuration, which is the most widely used one in our desktops and laptops. What's more, there may be some memory allocation functions cannot be detected by our iterative source-sink identification for the reason that not all assemblies are manually modeled in MLCSan. And finally, it is still the challenging pointer analysis and alias analysis. The determining conditions of must-alias or may-alias and the terminating conditions of our propagative MAR identification would cause potential false negatives. The first problem can be easily solved by compiling the kernel with *allyesconfig* and the possible false negatives can be eliminated. And we could manually model assemblies in kernels to address the second issue. Moreover, more precise alias analysis approaches are required to handle the last but challenging problem.

VII. RELATED WORK

A. KERNEL MEMORY PROTECTION

In MLCSan, we focus on the life-cycle of kernel memory space and introduce 3 memory-related cases of MLC bugs. Previous works on kernel memory protection and sanitation are various. The well-known and widely-used kernel address sanitizer (KASan) [19] can expose OOB and UAF bugs by instrumenting the kernel in compile time. KASan dynamically verifies kernel memory accesses in run time and may cause performance overhead. Other memory protection techniques such as K-miner [11], Watchdoglite [20], SoftBound [21] and CETS [22] also target memory errors such as OOB, UAF, etc. They are distinguishable from MLCSan since we concentrate on the life-cycle of memory space and propose analysis more than detecting traditional memory errors. Generally speaking, exposing bugs such as OOB and UAF is focusing on the result of accessing memory by error. On the contrary, MLCSan targets the causes of memory errors and reasons about which step in the life-cycle of memory space is error-prone. By revealing incorrect use of source-sink, unchecked dereference and lacked sink, we can tell the causes of memory misuse and expose MLC bugs.

B. API USAGE VERIFICATION

We check the source and sink functions of a MAR in MLCSan, which are APIs in kernel, and there are several other works in this filed. SSLint [24] statically detects incorrect use of APIs in SSL and TLS protocols. By modeling the SSL/TLS libraries, SSLint analyzes the target programs with API signatures. Yamaguchi *et al.* introduce Joern [25] and code property graph to represent a program. Code property graph incorporates syntax tree, control flow graph, etc. to comprehensively traverse a target program. Furthermore, APISAN [27] sanitizes API usages by a semantic cross-checking, which extracts API usage patterns in source code automatically. It uses semantic attributes of code and successfully found numerous bugs in kernel and other programs. MLCSan differs from these API checking tools because we consider the life-cycle of memory space and adopt bidirectional source-sink analysis to figure out whether they are incorrectly used or absent. In other words, we concentrate not only on single APIs, but we analyze coexisting source-sink pairs to expose situations where source or sink functions are not matched and where sinks are absent, not accompanying previous sources.

C. MISSING CHECK BUGS

MLCSan verifies a MAR before dereferenced to prevent possible null pointer dereference in kernel memory. Recently some works on missing check bugs are presented, such as [15], [28], [29], [43]–[48]. Moreover, Wang *et al.* formally define a LRC bug [12], in which a security recheck is missing after modification to a critical variable happens. LRSan detects LRC bugs by static analysis of kernel source code and identifies several bugs in Linux. However, the bugs are result of manually investigating over two thousands reports, which is a relatively high false positive rate. A more recent work, Crix [15] claims missing check as a dependent bug type and identifies over two hundred bugs via semantic- and context-aware inferences. However, MLCSan concentrates on unchecked dereferencing of memory, and MLC bugs contain not only missing check cases, but also incorrect source-sink and lacking sink cases as discussed in our paper. Moreover, the initial intention of MLCSan is detecting the violations to allocate, dereference, and free of memory space, which is far more different from concentrating on missing check of critical variables.

D. MEMORY BUG DETECTION

Sigmund *et al.* proposes value-flow analysis to detect memory leak in general software [49]. Since memory life-cycle contains 3 three steps, allocation, dereference and free, and we check the relations between them. Memory leak is the error in dereference-free, where memory is not freed after dereferenced. Compared to value-flow analysis in [49], MLCSan has 2 new features. (1) We take the error in allocation-free into consideration, which will cause incorrect source-sink bugs. (2) We also account for the relation between allocation-dereference, and it results in missing-check bugs. In conclusion, we take the relations between allocation, dereference and free into consideration, which are inseparable steps in memory life-cycle. And [49] fails to do so, and it cannot detect incorrect source-sink and missing-check bugs.

```
1  char *ptr;
2  ptr = kmalloc(1, FLAGS);
3  if (!ptr)
4      return -1;
5  printf("%s\n", ptr);
6  kfree(ptr);
7  --------------------------------------------------------
8  %3 = call i8* @kmalloc(i32 1, i32 1)
9  store i8* %3, i8** %2, align 8
10 %4 = load i8*, i8** %2, align 8
11 %5 = icmp ne i8* %4, null
12 br i1 %5, label %7, label %6
13 ; <label>:6:                              ; preds = %0
14 store i32 -1, i32* %1, align 4
15 br label %10
16 ; <label>:7:                              ; preds = %0
17 %8 = load i8*, i8** %2, align 8
18 %9 = call i32 (i8*, ...)
19 @printf(i8* getelementptr inbounds([4xi8],
20     [4xi8]* @.str, i32 0, i32 0), i8* %8)
21 %10 = load i8*, i8** %2, align 8
22 call void @kfree(i8* %10)
```

FIGURE 10. Comparison between C code and LLVM IR.

```
---------------------------------Missing Check------------------------------
[Instruction] %13 = call i32 @strncmp(i8* getelementptr ...)
[Source] kstrdup_const
[Sink] kfree_const
[MAR] node_info->vdev_port.name
[DU Chain] Alloc->GEP->Free
[Src Code] source/arch/sparc/kernel/mdesc.c: 360
[Function] get_vdev_port_node_info
356
357     node_info->vdev_port.id = *idp;
358     node_info->vdev_port.name = kstrdup_const(name,
                                          GFP_KERNEL);
359     node_info->vdev_port.parent_cfg_hdl = *parent_cfg_hdlp;
360     strncmp(node_info->vdev_port.name, s, ax_len);
361
362     return 0;
-----------------------------------------------------------------------------
```

FIGURE 11. A demonstration of MLCSan output of an example missing check case. The relative LLVM instruction, source, sink, MAR, define-use chain and source code are presented.

```
1     static int __init con_init(void)
2     {
3         console_lock();
4         vc->vc_screenbuf = kzalloc(vc->vc_screenbuf_size,
5                                       GFP_NOWAIT);
6  +      if (!vc->vc_screenbuf) {
7  +          kfree(vc);
8  +          console_unlock();
9  +          return -ENOMEM;
10 +      }
11    }
12    //drivers/tty/vt/vt.c
```

FIGURE 12. Part of a confirmed patch file, detected by MLCSan. In this patch, MAR *vc→vc_screenbuf* is checked and *vc* is freed when source *kzalloc()* fails, and *console_unlock()* is enforced to unlock the console.

VIII. CONCLUSION

As we all know, OS kernels leverage various source functions to perform heap memory allocation. More significantly, the return-value from allocation can only be dereferenced when not null and we should free it when out of use to prevent potential memory leaks. However, there are cases where memory data is improperly allocated or freed, dereferenced without verification, or unfreed at last. As discussed above, we define these cases as MLC bugs, including incorrect source-sink, missing check and lacking sink.

This paper studies the life-cycle of memory space in kernel and proposes the first systematical study of MLC bugs. And we build an automated and scalable detection framework, MLCSan. Furthermore, the occurrences of source functions, sink functions and MARs can be automatically detected by MLCSan, where MLC bugs may exist. Moreover, evaluation of the Linux and FreeBSD kernels with MLCSan is strong evidence that MLCSan can effectively detect MLC bugs, and we identify 41 new bugs. Additionally, the prototype of MLCSan will be open-source to boost related research in this field.

APPENDIX

See Figs. 10–12.

REFERENCES

[1] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2123–2138.

[2] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2345–2358.

[3] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. Checker: A soundy analysis for Linux kernel drivers," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 1007–1024.

[4] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, 2018, pp. 729–743.

[5] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in OS kernels," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 661–678.

[6] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 167–182.

[7] J. Pan, G. Yan, and X. Fan, "Digtool: A virtualization-based framework for detecting kernel vulnerabilities," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 149–165.

[8] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 279–293.

[9] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features," in *Proc. Asia Conf. Comput. Commun. Secur. (ASIACCS)*, 2018, pp. 587–600.

[10] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 1–16.

[11] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in Linux," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2018, pp. 1–15.

[12] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in OS kernels," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Jan. 2018, pp. 1899–1913.

[13] Z. Tong, S. Wenbo, L. Dongyoon, J. Changhee, M. A. Ahmed, and W. Ruowen, "Pex: A permission check analysis framework for Linux kernel," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)* Santa Clara, CA, USA: USENIX Association, 2019, pp. 1205–1220.

[14] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[15] L. Kangjie, P. Aditya, and W. Qiushi, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, Santa Clara, CA, USA: USENIX Association, 2019, pp. 1769–1786.

[16] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.

[17] Wikipedia. (2019). *Stack (Abstract Data Type)*. [Online]. Available: https://en.wikipedia.org/wiki/Stack_abstract_data_type

[18] Wikipedia. (2019). *Stack-Based Memory Allocation*. [Online]. Available: https://en.wikipedia.org/wiki/Stack-based-memory-allocation

[19] The Kernel Development Community. (2019). *The Kernel Address Sanitizer*. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[20] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "WatchdogLite: hardware-accelerated compiler-based pointer checking," in *Proc. Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, Feb. 2014, p. 175.

[21] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 245–258, May 2009.

[22] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," *ACM SIGPLAN Notices*, vol. 45, no. 8, pp. 31–40, 2010.

[23] K. Lu, C. Song, T. Kim, and W. Lee, "UniSan: Proactive kernel memory initialization to eliminate data leakages," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 920–932.

[24] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting SSL usage in applications with SSLINT," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 519–534.

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 590–604.

[26] H. Chen and D. Wagner, "MOPS: An infrastructure for examining security properties of software," in *Proc. 9th ACM Conf. Comput. Commun. Secur. (CCS)*, 2002, pp. 235–244.

[27] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing api usages through semantic cross-checking," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 363–378.

[28] S. Son, K. S. McKinley, and V. Shmatikov, "RoleCast: Finding missing security checks when you do not know what checks are," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 1069–1084, Oct. 2011.

[29] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2013, pp. 499–510.

[30] Linux Foundation. (2018). *2017 Linux Kernel Report*. [Online]. Available: https://linuxfoundation.cn/2017-linux-kernel-report-landing-page

[31] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2008, pp. 263–277.

[32] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *Proc. 24th USENIX Secur. Symp. (USENIX Secur.)*, 2015, pp. 81–96.

[33] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 48–62.

[34] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "CREDAL: Towards locating a memory corruption vulnerability with your core dump," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 529–540.

[35] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Palo Alto, CA, USA, 2004, pp. 75–86.

[36] LLVM. (2019). *Clang: A C Language Family Frontend for LLVM*. [Online]. Available: http://clang.llvm.org/

[37] Wikipedia. (2019). *Use-Define Chain*. [Online]. Available: https://en.wikipedia.org/wiki/Use-define_chain

[38] T. Gleixner. (2018). *Force ASM-Goto*. [Online]. Available: https://lore.kernel.org/patchwork/patch/935085/

[39] Linux Kernel Mailing List. (2019). *Clang ASM-Goto Support*. [Online]. Available: https://lkml.org/lkml/2018/2/13/1049

[40] SSLAB Gatech. (2019). *Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels*. [Online]. Available: https://github.com/sslab-gatech/deadline

[41] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," presented at the 10th USENIX Symp. Oper. Syst. Design Implement. (OSDI), 2012, pp. 163–177.

[42] B. Niu and G. Tan, "Modular control-flow integrity," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, vol. 49, 2013, pp. 577–587.

[43] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proc. ACM SIGOPS Oper. Syst. Rev.*, vol. 35, 2001, pp. 57–72.

[44] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger, "JIGSAW: Protecting resource access by inferring programmer expectations," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur.)*, 2014, pp. 973–988.

[45] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proc. 25th Symp. Oper. Syst. Princ. (SOSP)*, 2015, pp. 361–377.

[46] M. Monshizadeh, P. Naldurg, and V. N. Venkatakrishnan, "MACE: Detecting privilege escalation vulnerabilities in Web applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2014, pp. 690–701.

[47] L. Situ, L. Wang, Y. Liu, B. Mao, and X. Li, "Vanguard: Detecting missing checks for prognosing potential vulnerabilities," in *Proc. 10th Asia–Pacific Symp. Internetware*, 2018, p. 5.

[48] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically inferring security specification and detecting violations," in *Proc. USENIX Secur. Symp.*, 2008, pp. 379–394.

[49] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, San Diego, CA, USA, Jun. 2007, pp. 480–491.

GEN ZHANG was born in China, in December 1993. He received the master's degree in computer science and software analysis, in 2018. He is currently pursuing the Ph.D. degree with the College of Computer, National University of Defense Technology, Changsha, Hunan, China.

He has published three articles. His current research interests are fuzzing, software analysis, and binary analysis. He received Extraordinary Student Awards in 2015 in College of Computer for his good performance.