Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis

Yunlong Lyu[1]✉, Yi Fang[2], Yiwei Zhang[7], Qibin Sun[1], Siqi Ma[4], Elisa Bertino[3], Kangjie Lu[5], and Juanru Li[2,6,7]✉
[1]*University of Science and Technology of China*, [2]*Feiyu Security*, [3]*Purdue University*,
[4]*The University of New South Wales*, [5]*University of Minnesota*, [6]*Shanghai Qi Zhi Institute*,
[7]*G.O.S.S.I.P, Shanghai Jiao Tong University*

***Abstract*—Existing tools for the automated detection of memory corruption bugs are not very effective in practice. They typically recognize only standard memory management (MM) APIs (e.g., `malloc` and `free`) and assume a naive paired-use model—an allocator is followed by a specific deallocator. However, we observe that programmers very often design their own MM functions and that these functions often manifest two major characteristics: (1) Custom allocator functions perform *multi-object or nested allocation* which then requires structure-aware deallocation functions. (2) Custom allocators and deallocators follow an *unpaired-use model*. A more effective detection thus needs to adapt those characteristics and capture memory bugs related to non-standard MM behaviors.**

In this paper, we present a MM function aware memory bug detection technique by introducing the concept of *structure-aware and object-centric Memory Operation Synopsis (MOS)*. A MOS abstractly describes the memory objects of a given MM function, how they are managed by the function, and their structural relations. By utilizing MOS, a bug detection could explore much less code but is still capable of handling multi-object or nested allocations and does not rely on the paired-use model. In addition, to extensively find MM functions and automatically generate MOS for them, we propose a new identification approach that combines natural language processing (NLP) and data flow analysis, which enables the efficient and comprehensive identification of MM functions, even in very large code bases.

We implement a MOS-enhanced memory bug detection system, GOSHAWK, to discover memory bugs caused by complex and custom MM behaviors. We applied GOSHAWK to well-tested and widely-used open source projects including OS kernels, server applications, and IoT SDKs. GOSHAWK outperforms the state-of-the-art data flow analysis driven bug detection tools by an order of magnitude in analysis speed and the number of accurately identified MM functions, reports the discovered bugs with a developer-friendly, MOS based description, and successfully detects 92 new double-free and use-after-free bugs.

I. INTRODUCTION

Replacing the painstaking and meticulous manual code review with automated bug detection is promising [1], but a major limitation of most current bug detection tools is that their underlying program analysis techniques are not aware of the high-level semantics of certain functions. Because of such a limitation, existing tools are either time-consuming to use and unable to scale for large code bases, such as the Linux kernel, or suffer from imprecise program analysis results when the code structure is very complex. An important case is the detection of memory bugs (e.g., *use-after-free* and *double-free*). Ideally, a simple static analysis could be applied to track the lifetime of dynamic memory objects allocated and deallocated by memory management (MM) functions (e.g., `malloc` and `free`) and detect those bugs. In practice, however, such a straightforward approach would not work due to scalability issues, especially when analyzing code bases with billions of code lines and complex structures. In addition, the bug reports, returned by static analysis tools, typically contain very complicated data flows, which are difficult for code reviewers to read and confirm [2]. In response, recent efforts have enhanced bug detection by identifying more non-standard MM functions and simplifying their related data flow. NLP-EYE [3] and Susi [4] leverage a MM function summary based data flow analysis to tackle scalability. They use either a manual or a machine learning based custom MM functions identification to find MM behaviors in source code, and abstract them to obtain a more concise data flow. SinkFinder [5] adopts an alternative bug detection model that only tracks the data flow between pairs of interested functions; K-MELD [6] and HERO [7] also capture the pair relationship between memory allocation and deallocation to avoid a heavy data flow analysis of the function implementation.

Unfortunately, those approaches are still unable to provide extensive and yet precise analyses of MM functions especially the custom ones, and thus miss many critical memory bugs. ❶ The techniques used by those approaches for detecting MM functions in the code are either not general enough to deal with different kinds of source code projects, as implementations of MM functions in different projects vary significantly, or not accurate and suffer from high false positives. ❷ The MM summarization strategies used by those approaches fall short of describing the behaviors of MM functions precisely. To support sophisticated management of dynamic memory objects with complex structures, many custom MM functions perform *multi-object or nested allocation*. A custom MM function may allocate/release multiple memory objects in one invocation that may relate to each other according to a specific structure, whereas a standard memory allocator/de-allocator only

handles one memory object (a consecutive memory buffer) at a time. Moreover, while most existing bug detection models assume that an allocation only matches one deallocation, in real-world source code such an assumption is often violated. A compound structure containing multiple dynamic memory objects allocated by a custom memory allocator could be released either by invoking the corresponding custom deallocator, or by invoking a standard deallocator to release each object separately.

To implement an effective memory bug detection, we need an approach not only scalable to handle millions of lines of code but also precise enough to capture the complicated structure of memory objects, and the semantics and unique usage of MM functions. Custom MM functions "conceal" the complicated structure of memory objects and the operations on them, which makes it difficult for developers to understand their correct use. In this paper, we propose a new concept—*structure-aware and object-centric Memory Operation Synopsis (MOS)*—to address the fundamental problems when encountering MM functions. In brief, a MOS is a tuple that comprises a primary function name, a primary property (allocation or deallocation) and a list of dynamically managed memory objects occurring in either a return value or parameters of the function. It summarizes the structural relations of memory objects in MM functions. Because how a memory object should be operated on by an MM function depends on its structure, a MOS focuses on the object (hence *object-centric*) and its structure (hence *structure-aware*). Moreover, a MOS not only captures the structure of memory objects, but also describes the function property (allocation or deallocation) against each object.

For each MM function, we can generate an associated MOS to describe its MM behaviors. By integrating MOS, a bug detection process does not need to explore the internal implementation of all MM functions but can still precisely model the dynamically managed memory objects. In addition, a MOS-based bug detection allows one to remove the assumption that the paired-use model is used and to instead focus on the actual memory objects summarized by MOS. Therefore, the bug detection can analyze unpaired uses of MM functions and thus find more bugs.

A significant requirement for our MOS-enhanced bug detection is, however, to first identify all MM functions, as these functions are the basis of MOS. To address such requirement despite MM functions following various implementation styles in different kinds of source code projects, we design an accurate and yet very efficient identification technique by combining natural language processing (NLP) and data flow analysis. Our identification technique is organized according to two main steps. The first step uses an NLP-assisted classification against function prototypes in source code to categorize functions as *MM-relevant* or *MM-irrelevant*. Our insight here is that a function prototype is often human readable, and the natural language semantics of the prototype usually reflects the functionality. By using this semantic information, we have been able to approximately classify MM functions in a short time even out of millions of functions. The second step applies a data flow analysis against the implementation of each MM-relevant function identified by the first step. The data flow analysis checks whether the function does indeed perform memory allocation/deallocation using known memory allocators/deallocators (which are previously defined by us manually). The combination of those two steps achieves both efficiency and accuracy. Efficiency is achieved because the NLP analysis prunes irrelevant functions (i.e., functions not related to MM), for which a detailed analysis is thus not required. Accuracy is achieved because the MM-relevant functions are analyzed in details by using a static analysis.

Based on our design, we develop a MOS-enhanced memory bug detection system, GOSHAWK, that annotates the source code with MOS and conducts bug detection by leveraging MOS. We have implemented a prototype of GOSHAWK based on the Clang Static Analyzer (CSA) [8], and applied it to well-tested open source projects including two OS Kernels (Linux and FreeBSD), two user programs (OpenSSL and Redis), and three IoT SDKs (provided by Microsoft and Tencent). The experimental results show that our approach is able to identify MM functions in projects of different styles, and the generated MOS information helps GOSHAWK find 92 new use-after-free and double-free bugs in less than 10 hours. Moreover, the use of MOS provides developer-friendly bug detection results. Compared to traditional bug reports that typically describe MM bugs with a very complex data flow, a bug report with MOS is much more concise since the used MM functions in data flow are simplified. Developers can hence more easily confirm the reported bugs. We have reported the discovered bugs to developers using our MOS description, and helped them to quickly confirm the root causes of detected bugs.

Contributions:

- **A novel abstraction structure to summarize MM functions in data flow analysis.** We introduce the MOS concept to summarize MM behaviors and enable an object-centric and structure-aware memory bug detection approach. The use of MOS enhances standard data flow analysis by abstracting used MM functions but still preserving their detailed behaviors, and it helps eliminate the paired-use model adopted in existing detection tools.
- **A new approach to identify MM functions.** We combine NLP and data flow analysis to comprehensively identify MM functions in source code. Both the analysis speed and the number of accurately identified MM functions outperform the results of the state of the art tools by an order of magnitude.
- **GOSHAWK, a MOS-enhanced tool able to detect non-trivial memory bugs.** With GOSHAWK, we discovered 92 new memory corruption bugs in OS kernels, server applications and libraries, and IoT SDKs. We have made available the source code of GOSHAWK and details of our detection results at https://goshawk.code-analysis.org.

```
1  /*File: Linux/drivers/video/fbdev/hyperv_fb.c*/
2  static int hvfb_probe(struct hv_device *hdev,
3                const struct hv_vmbus_device_id *dev_id)
4  {
5      struct fb_info *info;
6      struct hvfb_par *par;
7      int ret;
8      ...
9       /*ALLOCATE OBJECT info*/
10   ❶ info = framebuffer_alloc(sizeof(struct hvfb_par),
11                                          &hdev->device);
12     if (!info)
13            return -ENOMEM;
14     ...
15     ret = hvfb_getmem(hdev, info);
16
17     if (ret) {
18           pr_err("No memory for framebuffer\n");
19           goto error2;
20     }
21     ...
22 error2:
23     ...
24     /*DEALLOCATE OBJECT info*/
25   ❷ framebuffer_release(info);
26     ... //info->apertures double free!
27     return ret;
28 }
```

```
29 struct fb_info *framebuffer_alloc(size_t size, ...)
30 {
31     struct fb_info *info;
32     char *p;
33     ...
34     p = kzalloc(fb_info_size + size, GFP_KERNEL);
35     info = (struct fb_info *) p;
36     ...
37     return info;
38 }

39 static int hvfb_getmem(struct hv_device *hdev, ...)
40 {
41     ...
42   ❸ info->apertures = alloc_apertures(1);
43     ...              //info->apertures allocation
44     pdev = pci_get_device(...);
45     if (!pdev) {
46       ❹ kfree(info->apertures);//info->apertures free
47         return -ENODEV;
48     }
49 }

50 void framebuffer_release(struct fb_info *info)
51 {
52     if (!info) return;
53   ❺ kfree(info->apertures);
54     kfree(info);
55 }
```

Fig. 1. A double-free bug caused by improper use of MM functions. It would be missed by existing detection because it follows the paired-use model. An effective detection should be aware of the object structure and MM functions.

II. Problem and Solution

In this paper, we focus on detecting memory corruption issues related to the use of different MM functions. An *MM function* is either a memory allocator or a memory deallocator. In real-world projects, MM functions are implemented in a highly diverse way. In general, an MM function can be classified as either a *primitive* one or an *extended* one. A primitive MM function only handles a single object, while an extended MM function allocates/deallocates nested or multiple objects. For instance, the standard libc functions `malloc` and `free` are primitive MM functions, whereas an extended MM function is one that repeatedly invokes `malloc` to allocate multiple memory objects in one execution.

Both primitive and extended MM functions can be improperly used and lead to memory corruption. It is important to note that memory bugs caused by extended MM functions are often more difficult to detect than bugs resulting from the improper use of primitive MM functions. In what follows, we first introduce a real-world example of a custom MM code resulting in a bug and discuss the challenges in implementing a precise detection. We then introduce our solution.

A. Motivating Example

The example in Figure 1 demonstrates the complexity of using MM functions in Linux kernel. At a glance, a dynamic memory object `info` of `struct fb_info` type is allocated by a `framebuffer_alloc` function (❶), and is released using the corresponding `framebuffer_release` function (❷). However, the `info` object contains an `apertures` sub-object, which also involves a dynamic memory allocation by invoking the `alloc_apertures` function (❸). In this case, the two allocators (`framebuffer_alloc` and `alloc_apertures`) are primitive MM functions, but the deallocator (`framebuffer_release`) is an extended MM function. It indicates that the pairwise use of `framebuffer_alloc` (as the allocator) and `framebuffer_release` (as the deallocator) is not proper. Actually, due to a previous memory deallocation in the error handling of `hvfb_getmem` (❹), the memory deallocation using `framebuffer_release` suffers from a double-free bug when the pointer of `apertures` is freed twice (❺).

We have observed many cases where such a problem leads to critical memory bugs. Such bugs are often "stealthy" because the code pattern is seemingly "correct" — developers tried hard to properly pair allocation and deallocation for memory objects with MM functions; unfortunately, the standard models (i.e., assuming that they handle a single memory object at a time, and follow the *paired uses*) in these cases are no longer applicable.

B. Challenges

The core challenge in detecting memory issues caused by MM functions, like `framebuffer_alloc`, `framebuffer_release`, and `alloc_apertures`, is how to implement an MM function aware data flow analysis to drive the bug detection. That is, we need to identify all used MM functions, properly abstract their MM behaviors in data flow, but still keep the precision of the other parts of the data flow analysis. We observe that most existing memory bug detection tools fail to implement such an analysis due to the following reasons:

Cannot comprehensively identify MM functions in different source code projects. When detecting memory bugs with a data flow analysis, only considering a small set of standard MM functions (e.g., `malloc` and `free`) as the sources and sinks of memory objects would lead to flow explosion if the analyzed project is very complex. It is essential to identify as many MM functions as possible (especially those extended ones) to simplify the data flow and thus accelerate bug detection. By conducting simple static analysis, which starts from tracking the data flow of the memory objects dynamically managed by standard MM functions, it is inherently difficult to determine when to stop. To find the interface of an MM function effectively, existing approaches consider information such as the function name, signature, and description. They either adopt a heuristic rule based identification (e.g., MemBrush [9], K-MELD [6]) or utilize NLP (e.g., NLP-EYE[3]) to extract features directly from the source code. These approaches rely only on interface information to identify MM functions. Such information is not sufficient for dealing with implementation diversity. Hence, an additional validation is also required.

Cannot precisely describe the behaviors of MM functions. A particular limitation of existing analyses is that they seldom consider behavior diversity and specificity of MM functions. Developers might adopt different types of designs to handle specific structure of memory objects. It is thus necessary to precisely describe the multi-objects, structure-related behaviors of MM functions, especially those extended ones. Unfortunately, we found that many function summary based bug detection tools (e.g., K-MELD, PairMiner [10], PF-Miner [11]) often adopt a coarse-grained detection model and cannot well summarize the MM behaviors. Hence, they fail to establish accurate data flows and are unable to find relevant memory bugs.

Cannot execute effective bug detection with limited code coverage. To detect memory bugs in large projects, a cross-module, inter-procedural analysis is necessary to handle very complex control and data flows. Within a reasonable analysis time and specific hardware resources, a bug detection cannot simultaneously pursue in-depth code exploration and analysis precision. Since an imprecise analysis would easily render the bug detection results useless due to the overwhelming false positives (e.g., PeX [12] and LRSan [13] that sacrifice data flow analysis can only find dozens of real bugs from thousands of reports), a developer-friendly bug detection tends to restrict the scope of code exploration to obtain a low false positive result. However, the loss of code coverage would hamper the detection of stealthy bugs involving a long execution path.

C. Our Solution

To hunt memory corruption bugs caused by custom MM behaviors effectively, we propose the following techniques. First, we observed that MM functions can be distinguished from the non-MM functions by only considering function prototypes. Specifically, we execute an NLP-assisted classification to label most MM function candidates efficiently, while some non-MM functions may be mislabeled. Then we determine whether a candidate performs memory allocation/deallocation by applying a top-down data flow analysis from each MM function candidate to check its internal implementation.

```
1  /*File: Linux/mm/dmapool.c*/
2  static struct dma_page *pool_alloc_page(struct
3                          dma_pool *pool, gfp_t mem_flags)
4  {
5       struct dma_page *page;
6                          //primitive function
7       page = kmalloc(sizeof(*page), mem_flags);
8       if (!page)
9             return NULL;
10                         //primitive function
11      page->vaddr = dma_alloc_coherent(pool->dev,
12            pool->allocation,&page->dma, mem_flags);
13       ...
14      return page;
15 }
```

```
MOS Representation:
{     Function name          Property
      pool_alloc_page    : Allocator,
      Memory object list     Object type
      RetVal             : struct dma_page*,
      RetVal->vaddr      : void*
}
```

Fig. 2. An example of using MOS to abstract memory management behaviors.

Second, we propose the Memory Operation Synopsis (MOS), a structure-aware and object-centric abstract representation of memory management behaviors and related memory objects, to help model memory allocation/deallocation of custom MM functions.

Definition 1. *A Memory Operation Synopsis (MOS) is defined as a MM behaviors summary that consists of primary function name, primary property, and a list of correlated memory objects that are dynamically managed. It is represented as* $M = \{M_{name}, M_{type}, [O]\}$*, where* M_{name} *is the primary function name indicating the summary ancestor;* M_{type} *is the primary property indicating whether the ancestor is a memory allocator or deallocator;* $[O]$ *is a list of memory objects managed by the ancestor. In the list* $[O] = [o_1, o_2, ..., o_n]$*, each memory object, i.e.,* $o_i = (name, type)$*, is a 2-tuple where* $name$ *is a field-based variable name of current memory object to reveal its structure nested relationship, and* $type$ *is its pointer type.*

Figure 2 shows an example of how to model memory management behaviors with MOS. In this example, function `pool_alloc_page` first allocates a `dma_page` buffer using function `kmalloc`. Then, it invokes function `dma_alloc_coherent` to allocate a `vaddr` object as a member of the `dma_page` object. Finally, the allocated object is returned through the return value of the function. To summarize the memory management behavior of `pool_alloc_page`, the MOS representation first labels its name and its property as an allocator, then it stores a list to record the allocated memory objects and how they are returned (i.e., via RetVal and RetVal->vaddr).

Third, we utilize MOS to enhance data flow analysis based memory bug detection. By introducing MOS into the standard

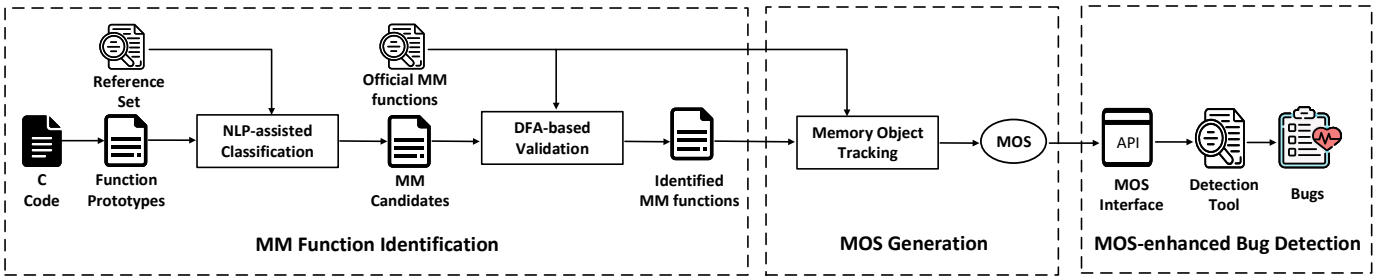

Fig. 3. Workflow of GOSHAWK

static analysis, the data flow of an MM function can be simplified by only retaining necessary details of all dynamically managed objects with compound structures. In comparison, many function summary based bug detection techniques assume that a MM function only handles one simple memory object, and cannot describe complex memory management cases. In addition, even if memory bug analysis tools report a potential problem, they need to output the data flow related to the improperly used memory object for developers to confirm the bug and its root cause. By utilizing MOS, we can cut the MM function part of a too-long-to-understand data flow, making it easier to understand.

In the following sections, we detail the design and implementation of GOSHAWK, our proposed MOS-enhanced bug detection tool, and show its effectiveness in finding non-trivial memory corruption bugs in various open-source projects.

III. GOSHAWK: A MOS-ENHANCED MEMORY BUG DETECTION SYSTEM

A. Overview

GOSHAWK, our detection system for memory corruption bugs in C source code, utilizes MOS to model memory operations of MM functions and the relevant dynamically allocated memory objects in a structure-aware way. It executes a three-phase workflow for memory bug detection (see Figure 3). In the MM function identification phase, GOSHAWK analyzes source code and pinpoints MM functions automatically. Then, through the MOS generation phase, GOSHAWK abstractly characterizes each identified MM function into a MOS representation. In the final phase of MOS-enhanced bug detection, GOSHAWK fulfils a MOS simplified memory object data flow tracking through the analyzed target and detects memory bugs correspondingly.

GOSHAWK combines an NLP-assisted classification and a data flow analysis based validation to identify MM functions. We observe that developers often declare the interface of a function to reflect its internal behavior, and we could classify functions according to the natural language semantics of their interfaces. Therefore, GOSHAWK utilizes NLP to extract specific features from function prototypes declared in source code, and computes the feature similarity for each function against a reference set, which consists of pre-collected prototypes of representative MM functions. This efficiently classifies MM functions and non-MM functions by only checking their prototypes. And even if the classification sometimes over-labels non-MM functions as candidates, it seldom mistakenly labels a MM function. Hence GOSHAWK can exclude a large portion of irrelevant functions via the classification. Next, GOSHAWK conducts a top-down data flow analysis starting from the interface of each candidate, checking whether the function uses any official MM functions (how we maintain a set of such allocators/deallocators is detailed in Section IV-B). And if the data flow analysis validates that the candidate does execute MM behavior, GOSHAWK identifies it as a MM function with a high accuracy.

To further summarize a MM function with a structure-aware granularity, **GOSHAWK utilizes the MOS representation to abstract how memory objects are dynamically allocated/deallocated**. Since for each MM function, the previously applied data flow analysis has tracked and labeled the dynamic memory status of all the relevant memory objects and analyzed the memory operations (e.g., allocating/releasing memory, returning a pointer through return value/parameters). GOSHAWK directly leverages this data flow information to translate implementation details into MOS.

With the help of MOS, **GOSHAWK implements a structure-aware, object-centric memory bug detection.** GOSHAWK leverages MOS information to update memory object status. When traversing again the source code to detect bugs, GOSHAWK skips the redundant data flow analysis when encountering an MM function; it instead reads information provided by MOS to understand the detailed MM behaviors, and directly updates the status of memory objects accordingly. MOS helps simplify the entire data flow by summarizing a MM function as a node, but still preserves the structure information of the memory objects. Since every memory object can be precisely described and tracked, more bugs are expected to be found. The simplified data flow also retrofits the bug report: the reported issues can be examined without inspecting the data flow inside the MM functions.

B. MM Function Identification

NLP-assisted classification. The NLP-assisted classification starts from parsing the source code of the tested project to extract all function prototypes, and chooses the prototype with at least one pointer (either a return value or a function parameter). The extracted prototypes are then sent to a ULM-based segmentation [14] process and each is divided

into a list of subwords (GOSHAWK utilizes Byte Pair Encode [15] algorithm to collect meaningful subwords and their occurrence frequency from the posts of StackOverflow [16]). Then, GOSHAWK converts the subword list into a numeric vector by utilizing a Siamese network [17] with Transformer encoders [18] trained with manually labeled MM and non-MM functions. Finally, GOSHAWK computes a similarity score between the numeric vector of a function and three reference vectors, which separately indicate *memory allocation function*, *memory deallocation function*, and *non-MM function* type. According to the similarity score, GOSHAWK classifies the tested functions as one of the above types. More details of model training are provided in Section A in Appendix.

The essential aspect of our proposed similarity comparison is how to build the three reference vectors. We first randomly collect 5,342 function prototypes as a *function prototype corpus* from real-world projects (e.g., Linux kernel) and manually classify them into three types. Next, we apply the same ULM-based segmentation to the classified functions to obtain three sets of subword lists. Using these sets as the training data, we drive a Siamese network to generate optimized Transformer encoders. After the training, we not only obtain a specific Siamese network with trained encoders, but also use this network to convert subword lists of our function prototype corpus into vector lists of three types. Finally, for vectors of each of the three types, we calculate the arithmetic mean and finally obtain three reference vectors.

Data Flow Analysis based Validation. Identifying MM functions only through an NLP-assisted classification is obviously inaccurate, and thus GOSHAWK double-checks the results by conducting a top-down data flow analysis against each MM function candidate to check whether the candidate or any of its sub-routine uses an *official MM function*. Official MM functions are those MM APIs (e.g., `malloc`, `kzalloc`, `free`) standardized by widely used libraries (e.g., libc) or systems (e.g., OS kernel). We collect such official MM functions from the official documents of operating systems [19] and libraries [20]. In detail, the validation traverses the code of a candidate and its sub-routines, finding whether there exist invocations of official MM functions. Once a specific invocation is found, GOSHAWK builds the data flow related to the official MM function inside the candidate, examines whether an allocated/deallocated memory object is connected to the parameter/return value of the candidate. If so, GOSHAWK identifies the candidate as a MM function with a high accuracy.

C. MOS Generation

GOSHAWK utilizes MOS to summarize all allocation/deallocation operations inside an MM function. To generate MOS for an MM function, GOSHAWK reuses the recursive data flow analysis of MM function validation to track the propagation of allocated/deallocated memory objects (pointers) from official MM functions. If the involved official MM functions are allocators, GOSHAWK adopts a forward data flow analysis to track whether the pointers of those allocated objects are copied to either a return value or any parameters of the function interface. Otherwise, it tracks data flow backwards to label the released parameters. Note that the adopted data-flow analysis is field-sensitive, and thus it maintains the nested relations of `struct` elements. Once all data flow and structure information of the involved memory objects is collected, GOSHAWK follows the MOS definition to generate the associated MOS for its MM function. In particular, if an MM function contains other MM functions with MOS already generated, GOSHAWK merges the MOS information of those sub-routines to that of the outermost MM function. For those official MM functions, GOSHAWK considers them as domain knowledge and prepares the MOS for them before any analysis.

```
1  static void tpci200_uninstall(struct tpci200_board
2                                                *tpci200)
3  {
4      tpci200_unregister(tpci200);
5      kfree(tpci200->slots);
6  }
7
8  static void tpci200_unregister(struct tpci200_board
9                                                *tpci200)
10 {
11     pci_iounmap(tpci200->info->pdev,
12                 tpci200->info->interface_regs);
13     pci_iounmap(tpci200->info->pdev,
14                         tpci200->info->cfg_regs);
15     ...
16 }
17
18 void pci_iounmap(struct pci_dev *dev, void *addr)
19 {
20     ...
21     iounmap(addr);
22 }
```

❹ ❺ ❷ ❸ ❶

```
MOS Representation:
{
      tpci200_uninstall: Deallocator,
      tpci200->info->interface_regs: void *,
      tpci200->info->cfg_regs      : void *,
      tpci200->slots               : struct tpci200_slot *
}
```

Fig. 4. An example of data flow based MOS generation.

Figure 4 shows a concrete example of MOS generation. In this case, GOSHAWK first identifies three deallocators, `tpci200_uninstall`, `tpci200_unregister`, and `pci_iounmap`. When GOSHAWK analyzes the outermost `tpci200_uninstall` function, it tracks the internal `pci_iounmap` function and further finds an official deallocator `iounmap`. GOSHAWK then generates the MOS for `pci_iounmap`, and since `tpci200_unregister` invokes `pci_iounmap`, GOSHAWK merges the MOS of `pci_iounmap` into that of `tpci200_unregister`. That is, the MOS of `tpci200_unregister` contains the (two objects) deallocation behaviors of `pci_iounmap`. Finally, GOSHAWK summarizes the MOS of `tpci200_uninstall` by combining the MOS of `tpci200_unregister` and that of the standard memory deallocation function `kfree`. As a result, when a program analysis traverses the code, it only needs to use the MOS of the outermost function `tpci200_uninstall` to precisely describe the three to-be-released sub-objects of the `tpci`

parameter without exploring the internal MM functions.

During our MOS generation stage, we do not consider the path constraints of conditional memory allocation/deallocation. Integrating such description into MOS is not difficult; it however significantly complicates the MOS definition. Our design choice is to consider the MM behaviors as unconditional in MOS (i.e., by executing a flow-insensitive analysis at this stage), and leave the task of checking constraints related with function invoking, global variables status, or logical arithmetic after the bug detection. That is, when a potential bug is reported, GOSHAWK then applies a fine-grained symbolic execution (e.g., by using the Z3 solver [21]) to only re-analyze the involved code paths of the reported issue. This can filter out the possible false positives caused by the conditional memory allocation/deallocation without incurring heavy performance overhead beforehand.

D. MOS-enhanced Bug Detection

When GOSHAWK traverses the source code to detect memory bugs, it achieves an efficient yet precise bug detection with the help of MOS. It modifies the way of how dynamically managed memory objects are modelled to reduce the scope of code exploration, and then detects complex memory corruption bugs in an object-centric and structure-aware way.

Traditionally, a memory corruption bug detection starts from recording memory object allocations of a few standard allocators (e.g., `malloc`), tracking the data flow of allocated objects until they are released by deallocators (e.g., `free`), and examining whether the released objects are used or released again. However, this strategy often needs to explore a long code path with very complex data flow involved, and easily leads to analysis state explosion. GOSHAWK, instead, explores memory objects propagation between custom allocators/deallocators and does not need to explore the internal code of MM functions, and thus avoids path explosion effectively.

In particular, when encountering a MM function, GOSHAWK first retrieves the function type from MOS. If the function is an allocator, GOSHAWK reads the dynamically managed memory objects list, and creates new symbols for all allocated objects accordingly; if the function is a deallocator, GOSHAWK updates the status of involved memory objects by labeling their symbols as "released". Consider the code in Figure 1 as an example. After the MM function identification and MOS generation, GOSHAWK creates MOS for `framebuffer_alloc` (❶) and `alloc_apertures` (❸). When the bug detection encounters these two MM functions, the MOS information helps GOSHAWK directly model the allocated memory objects (and avoids a redundant analysis against the MM functions). And when the bug detection moves forward to the deallocation function `framebuffer_release` (❷), GOSHAWK also reads its corresponding MOS and then updates the statuses of the allocated objects. In this case, GOSHAWK only needs to explore code paths between ❶ and ❷ to find the bug.

More importantly, MOS provides rich information to help GOSHAWK precisely model the memory management behaviors of multiple memory objects nested in structures. In the case in Figure 1, the MOS-based memory object modeling helps GOSHAWK understand the structure of object pointed by the `info` pointer. Therefore, GOSHAWK is able to accurately update the status changing of a compound struct: when the bug detection moves from ❶ to ❷, GOSHAWK not only knows the allocation/deallocation against the `info` pointer, but also that against the `info->apertures` pointer, and thus captures the double-free bug (❹ + ❺). In comparison, other approaches (e.g., the pair-based mining approaches assume that each memory allocator creates a single memory object, and there is a corresponding deallocator to release it) often fail to precisely track such compound objects even if they also generate summaries for custom MM functions.

IV. IMPLEMENTATION

The implemented GOSHAWK prototype consists of over 3.5K lines of C++ code and 4K lines of Python code. The source code is available at https://goshawk.code-analysis.org. We elaborate on implementation details below.

A. MOS-enhanced Bug Detection on CSA

The GOSHAWK prototype is built on top of the CSA code analysis engine. The original version of CSA detects memory corruption bugs by using path-sensitive and inter-procedural symbolic execution. To integrate MOS with CSA, we implement a MOS interface for CSA to interpret the MOS and model the related memory objects. We also design a use-after-free and double-free checker by reusing the detection logic of MallocChecker [22], which is the official memory corruption bug checker of CSA. Then, we use Z3 SMT solver to eliminate some false warnings due to infeasible paths.

MOS Interface Implementation. To implement the MOS interface, we overwrite the callback `evalCall` of CSA. The callback `evalCall` is used to model each invoked function. By overwriting the callback `evalCall`, the MOS interface skips analyzing the MM function implementation, and models the function memory management behaviors on the CSA engine according to the MOS information. In detail, when `evalcall` is invoked, the interface checks whether the callee function has a MOS attached to it. If no MOS is found, the interface returns back to CSA default analysis, steps into the callee function body and continues its inter-procedural analysis. Otherwise, the interface models the relevant memory objects in three steps: (1) The interface parses the MOS to obtain the detailed MM behavior information of the function and a list of dynamically managed memory objects. (2) By retrieving each memory object from the list, the interface utilizes the CSA engine to explore the corresponding symbolic expression that represents the memory object. If the memory object is a member nested in a structure, the interface traverses the structure definition and pinpoints the expression where the memory object is declared. (3) The interface separately creates a new symbol for the expression with allocator, retrieves the symbol of the deallocator expression, and updates its status to "released" for the expression with deallocator.

Checker Design. We implement a MOS-enhanced checker to detect use-after-free and double-free bug checker during the analysis phase of CSA. It starts a path-sensitive and inter-procedural analysis on each translation unit and selects the first node of a call graph as the entry point. It then analyzes each encountered callee function until all the call graphs of the current translation unit have been checked. When CSA encounters a callee function, the checker invokes the callback `evalCall` to see whether the callee function has an attached MOS; if this is the case, it interprets the MOS and creates or updates relevant symbols. If a symbol with "released" status is being updated to "released" status again, the checker reports a double-free violation. The checker also monitors all de-reference operations, and reports a use-after-free violation if a symbol with "released" status is being de-referenced.

Infeasible Paths Elimination. CSA uses a range based constraint solver [23], which maps constraint conditions to symbolic binary comparison expressions without performing logical or arithmetic calculations between unknown symbol values. Thus it has high performance but introduces false positives due to the imprecise constraint solving. Also, the conditional allocation/deallocation inside MM functions may introduce some false positives. To eliminate those false warnings, GOSHAWK leverages CodeChecker [24], a static analysis framework built on the LLVM/CSA toolchain and currently integrated with the Z3 SMT solver, to cross check the feasibility of each reported issue path. To be more specific, with the violations reported by the default range based constraint solver, GOSHAWK utilizes CodeChecker to collect the entire violation paths (from an official allocator to the violation point) and re-check the feasibility of those paths by using the Z3 solver.

Code Exploration Settings. To implement a cross-module inter-procedural analysis, CSA inlines invoked functions from different source files code with a Cross Translation Unit (CTU) [25] analysis. Intuitively, to obtain a complete code coverage, the analysis should explore as many functions/files as possible. This is, however, infeasible in most cases since it would inevitably encounter scalability issues [26]. To prevent a specific analysis from consuming too much time and hardware resources, CSA supports customized breadth and depth of its code exploration. In particular, the following analysis parameters are directly related to the bug detection of GOSHAWK:

1) Maximum analysis depth (*MAX-AD*): starting from an entry point function, the maximum number of explored (inlined) functions during an inter-procedural analysis;
2) Maximum analysis breadth (*MAX-AB*): starting from an entry point function, the maximum steps of execution on the symbolic state graph during a symbolic execution based path traversing.
3) CTU import threshold (*CTU-THR*): when analyzing a translation unit (i.e., a source file) the maximum number of external translation units to import.

B. MM Function Validation

As it is time consuming and inaccurate to conduct code analysis targeting an entire project, GOSHAWK only analyzes the sub-routines of the MM function candidates. Before that, we collected a list of generally and widely used MM functions from the official documents of the operating system kernel and libc. In total, 32 functions are added to the official MM function set (details are listed in the Appendix). Then GOSHAWK executes three steps to validate MM function candidates:

1) Starting from the program codes, GOSHAWK leverages the Clang [27] compiler and plugins to construct the whole program call graph and generate control flow graphs for each function.
2) Given a MM function candidate, GOSHAWK first records the call-chains to each official MM function invocation on the whole program call graph, and then records the corresponding paths on control flow graphs.
3) With the recorded call-chains and paths, GOSHAWK conducts backward program analysis to propagate the data flows of official MM functions by up-walking the call-chains and paths. Specifically, the data flow propagation is achieved by checking the variables involved in statements of `BinaryOperator` and `VarDecl` in Clang plugins, which assign values to variables. If there exists a data flow passed from outside of the MM function candidate, GOSHAWK regards this function as a valid MM function.

To avoid scalability issues (e.g., numerous call-chains and paths are recorded when starting from `main` function) and precision issues (i.e, indirect call target resolution and pointer address calculation on data flow propagation), we adopt a conservative policy: (1) Do not resolve indirect call target when constructing the whole program call graph. (2) Only record the MM-relative call-chains, such as MM candidate-> MM candidate -> official MM function. (3) Do not propagate the data flows involved in pointer address calculations.

C. Ambiguous MM Behaviors Modeling

For re-allocators which perform different memory allocation and deallocation behaviors based on the concrete values of their parameters at runtime, GOSHAWK simply considers them as common allocators, since in MOS we do not distinguish a re-allocator from an allocator. For functions that fetch or save memory objects in a global link list, GOSHAWK currently does not consider them as MM functions, although technically it is feasible to track their data flows.

V. EVALUATION

We assessed GOSHAWK by mainly considering its effectiveness in identifying MM functions and detecting memory corruption bugs (in particular, use-after-free and double-free bugs). The following research questions were answered:

- **RQ1:** Can GOSHAWK identify MM functions precisely and extensively for different kinds of source code projects?
- **RQ2:** Is MOS-based detection able to find memory bugs more effectively?
- **RQ3:** How does MOS help accelerate the data flow analysis and the bug detection?

TABLE I
RESULTS OF CUSTOMIZED MEMORY MANAGEMENT FUNCTION IDENTIFICATION AND MEMORY BUG DETECTION IN POPULAR OPEN-SOURCE PROJECTS

Program	Version	Lines of Code	Number of Functions	Identified Allocators		Identified Deallocators		Detected Bugs		
				Primitive	Extended	Primitive	Extended	UAF	DF	Confirmed
Linux	5.12-rc2	20.1M	549K	2,916	1,805	1,812	4,266	32	17	40
FreeBSD	13.0.0	5.9M	99K	482	273	393	853	9	9	17
OpenSSL	3.0.0	456K	19K	93	15	134	193	1	8	9
Redis	6.2.1	149K	3K	51	9	28	41	1	0	1
Azure	2021_Ref01	661K	4.8K	126	7	58	177	0	4	1
QcloudE	3.1.8	86K	759	21	1	26	21	2	5	7
QcloudH	3.2.3	79K	638	20	1	25	21	2	2	4
Total	-	-	-	**3,709**	**2,111**	**2,476**	**5,572**	**47**	**45**	**79**

QcloudE refers to Qcloud IoT Explorer Device SDK and QcloudH refers to Qcloud IoT Hub Device SDK.
UAF: use-after-free bug; **DF**: double-free bug

Environment. The evaluations were conducted on a server running Ubuntu 20.04 with an AMD Ryzen Threadripper 3990X, 192 GB RAM, and a GeForce GTX2080-Ti GPU card.

A. Overall Results

We applied GOSHAWK to seven well-known open source projects of different code scales. Table I summarizes the tested projects, and the identification and detection results for each tested project. GOSHAWK successfully analyzed all their source code and completed the use-after-free and double-free bug detection in less than one day (details of analysis costs are reported in Section V-F). In total, **GOSHAWK identified more than 10,000 MM functions, and detected 92 (47 use-after-free, 45 double-free) bugs**. All the found bugs have been reported to the corresponding communities[1]. In the following, we detail the results of our evaluation. We also provide the raw analysis and detection results as supplementary materials at https://goshawk.code-analysis.org.

B. Comparison with MallocChecker

To evaluate how identified MM functions and MOS enhance current bug detection tools, we select the official memory corruption bug checker of CSA, MallocChecker, and repeat our bug detection experiments. MallocChecker does not consider non-standard MM functions and straightforwardly tracks memory objects between standard MM functions (`malloc`, `free`, `kmalloc` and `kfree`). We first applied MallocChecker to the seven projects under the same code exploration settings (i.e., covering the same code range), and reviewed how many bugs detected by GOSHAWK were also detected by it. As Table II shows, MallocChecker only found 21 of the 92 bugs, and it did not find any new bug that GOSHAWK could not find. For small projects (OpenSSL, Redis and IoT SDKs) where the analysis scope is less limited ((i.e., could cover more code paths), MallocChecker still misses 14 out 25 bugs. For the larger projects (Linux and FreeBSD kernels) where MallocChecker could only explore a limited range of code, it detected only 7 out of 67 (16.4%) bugs.

[1]Before our submission (December 2021), 79 out of 92 bugs have been confirmed and the rest 13 bugs are under review.

TABLE II
COMPARISON WITH MALLOCCHECKER IN TERMS OF DETECTION EFFECTIVENESS AND EFFICIENCY (IN MINUTES)

	Goshawk		MalChk-S		MalChk-E	
	Bugs	Time	Bugs	Time	Bugs	Time
Linux	49	328.91	2	367.72	-	-
FreeBSD	18	20.84	5	19.73	6	664.76
OpenSSL	9	3.51	1	3.66	1	79.13
Redis	1	1.03	0	0.78	0	24.01
Azure	4	0.22	2	0.26	2	6.41
QcloudE	7	0.10	4	0.11	7	1.77
QcloudH	4	0.07	4	0.08	4	8.66
Presicion	**63.4%**		**20.6%**		**35.7%**	

MalChk-S : MallocChecker with AMD 3990x, 192G RAM.
MalChk-E : MallocChecker with Xeon 4126, 1T RAM.

Intuitively, it is the limitation of code exploration that hinders MallocChecker from discovering more bugs. To check whether the used hardware (and the corresponding code exploration scope) limits the effectiveness of MallocChecker, we repeated the experiments with a more powerful server (two Intel Xeon 4216 processors, 64 cores, 1TB RAM), which allowed the code exploration scope to extend from *MAX-AD*=5, *MAX-AB*=225,000, *CTU-THR*=100 to *MAX-AD*=10, *MAX-AB*=2,250,000, *CTU-THR*=20,000. As shown in Table II, the analysis time increased significantly, but MallocChecker only found four more bugs. More in detail, when analyzing the Linux kernel, MallocChecker cannot scale at all; even for a single source file, it consumed more than 15GB RAM and took four hours to finish the analysis. This is certainly infeasible for Linux kernel with more than 20,000 source files. In comparison, by covering the same (or less) amount of code, GOSHAWK could detect bugs since it does not need to explore code paths in MM functions and avoids the path explosion issue. For instance, a bug related to the `l2cap_sock_alloc_skb_cb`[2] custom allocator involves an 11-layer call chain to the final allocation. In this case only the MOS-enhanced bug detection could effectively handle it.

[2]This function is declared in `linux/net/buletooth/l2cap_sock.c`

Another interesting observation is that even though MallocChecker adopts a simpler bug detection model, it consumes more time than GOSHAWK when analyzing the Linux kernel (367.72 vs 328.91 minutes) with the same code exploration scope. This is because GOSHAWK avoids to redundantly analyze those 13,868 MM functions. To quantitatively measure how much analysis costs were saved, we checked all intermediate analysis records and found that by using MOS, GOSHAWK avoids 9,779,077 times of function analysis and does not need to repeatedly explore 253,324,662 paths.

C. Effectiveness of MM Function Identification

GOSHAWK analyzed more than 27.3 million lines of code for all tested projects, and extracted more than 676,000 function declarations. Then it conducted the NLP-assisted classification. Given the trained Siamese network and the corresponding reference vectors, 90,069 functions were first classified. By further considering whether the prototype of a classified function contains a data pointer, 53,977 MM function candidates were selected. Obviously, the candidate set contains many non-MM functions only with a similar prototype, and would introduce false positive to the subsequent bug detection. Therefore, GOSHAWK continuously applied data flow analysis based validation to filter irrelevant functions.

After applying data flow analysis based validation to cross-check the candidates, GOSHAWK identified 13,868 (5,820 allocators and 8,048 deallocators) custom MM functions out of 53,977 functions. For projects, OpenSSL, Redis, and IoT SDKs, we manually verified all the identified results and confirmed that they were correctly identified. For the Linux kernel and FreeBSD kernel, although we did not have ground truth data and were not able to manually verify all identified functions, we did check 300 functions randomly and found no misidentified case. We also randomly chose 50 filtered functions to check whether they were reasonably eliminated. Our manual inspection found that although the names of those functions were similar to names of allocation and deallocation functions (e.g., `percpu_alloc_setup`, `nr_free_buffer_pages`), they did not execute any memory allocation or deallocation.

To systematically evaluate how NLP-assisted classification and data flow analysis based validation affect the accuracy, we manually chose 200 allocators, 200 deallocators, and 600 non-MM functions to test the identification accuracy. As shown in Table III, GOSHAWK tended to mis-label non-MM functions when using only NLP-assisted classification (low precision). When using both NLP-assisted classification and data flow analysis based validation, the precision increased to 100% but the recall decreased, which indicates that it would miss more MM functions. To verify this at a larger scale, we referred to K-MELD [6] and SinkFinder [5], two recent tools which also labeled MM functions in Linux kernel. K-MELD and SinkFinder respectively labeled 1,267 and 438 MM functions in Linux kernel (version 5.2.13 and 4.19); thus we applied GOSHAWK to these two versions of Linux kernel and compared our results to them. GOSHAWK outperformed K-MELD and SinkFinder in finding more MM functions (Table IV), but it missed 594/259 functions labeled by K-MELD/SinkFinder. After manually reviewing all those functions, we found that 446/205 of them did not perform MM behaviors (they were thus false positives of K-MELD and SinkFinder). We then re-examined the rest 148/54 functions and found that our NLP-assisted classification did label 129/46 of them. However, the subsequent validation failed to return correct data flow information due to analysis issues such as indirect calls and sophisticated pointer arithmetic.

TABLE III
EVALUATING ACCURACY OF MM FUNCTION IDENTIFICATION WITH A GROUND TRUTH DATASET

	Allocator		Deallocator	
	Precision	**Recall**	**Precision**	**Recall**
w/o DFA	89.6%	91.0%	90.2%	97.0%
w/ DFA	100%	84.5%	100%	89.5%

Answer to RQ1: GOSHAWK guarantees the soundness of MM function identification but may miss a small portion of MM functions due to imperfect NLP and data flow analysis.

TABLE IV
COMPARISON OF MM FUNCTION IDENTIFICATION IN THE LINUX KERNEL

Version	**K-MELD**		**SinkFinder**		**Goshawk**			
	A	**D**	**A**	**D**	**A**	**D**	**MN**	**MD**
v5.2.13	806	461	-	-	4,571	4,847	19	129
v4.19	-	-	256	182	4,396	4,704	8	46

A : # of allocators; **D** : # of deallocators;
MN : # of MM functions missed by NLP-assisted classification;
MD : # of MM functions missed by DFA based validation.

D. Features of generated MOS

GOSHAWK successfully generated MOS for all 13,868 identified custom allocators/deallocators. Based on MOS information, we can observe the distribution (primitive vs extended) of custom MM functions in different projects. Table I lists the number of each kind of MM functions, and the overall distributions. We found that there are more custom deallocators (8,048) than custom allocators (5,820), and each project has this characteristic. This implies that allocators and deallocators are not always paired, so is their usage.

In all 5,820 custom allocators, GOSHAWK found that the number of primitive ones (3,709, 63.7%) is larger than that of the extended ones (2,111, 36.3%). In comparison, for the 8,048 custom deallocators the number of primitive ones (2,476, 30.7%) is smaller than that of extended ones (5,572, 69.3%). This distribution shows that developers may be more likely to allocate memory objects separately, but deallocate them in a centralized way (refer to the motivating example in Figure 1).

The generated MOS reflected the widely existence of complex memory object managements among the identified MM

functions. For the 5,820 memory allocation functions, 4,311 (74.1%) functions return allocated memory objects via a return value (548 functions return the allocated memory objects with at least one sub-object allocated), and 1,509 (25.9%) functions return the allocated memory objects via parameters. For the 8,048 memory deallocation functions, 2,504 (31.1%) functions directly release a pointer, and 3,938 (48.9%) functions only dereference the pointer to obtain the structure information to release its sub-objects, while 1,608 (20%) first dereference the pointer and then release both its sub-objects and the main object its points to. The experimental results demonstrate that modeling extended MM functions as primitive ones would lead to a imprecise bug detection. In response, the generated MOS preserved essential information to model MM functions and the related memory objects, thus are expected to help the following bug detection work precisely.

E. Bug Detection

1) Code Exploration Settings: To fully use of our hardware resources (64-core AMD 3990x and 192GB RAM), we empirically set the analysis parameters of CSA as *MAX-AD*=5, *MAX-AB*=225,000, and *CTU-THR*=100 (larger parameters would exhaust the CPU/RAM and stop the analysis).

2) Detected Bugs: With the help of the generated MOS information, GOSHAWK conducted use-after-free and double-free bug detection against recent versions of Linux kernel, FreeBSD kernel, OpenSSL, Redis and IoT SDKs. GOSHAWK reported 145 potential bugs. Our manual audit, only taking three days (8 hours per day) for one developer (who was not responsible for the buggy code), confirmed that **92 (63.4%) of the reported issues were real bugs and previously unknown** (a more detailed result is reported in Table VII in Appendix), even though those projects were tested by many code analyzers and checked by reviewers. Actually, we inspected these bugs in the commit history of the tested projects and found that many of them have been present for a long time. For instance, in the Linux kernel we found that 12 bugs have been present for more than 6 years[3], and another 3 bugs have even been present for 15 years[4]. Although the Linux kernel is periodically tested by different static tools, we argue that the lack of MM summarization still makes difficult for developers to confirm those bugs, and MOS based bug descriptions help them examine the issues more effectively. Except for few special cases, our reported bugs received the confirmation of kernel developers within 2 days on average.

3) Detection Accuracy: The bug detection of GOSHAWK against the seven tested projects initially labeled 308 issues in the source code. Among the 308 labeled issues, we first found 163 cases of infeasible path caused by insufficient contextual information, complex logical calculations or conditional allocation/deallocation. By using the Z3 SMT solver to re-analyze the issue relevant paths, we excluded 30 false warnings. For the other 133 cases, we found that even the Z3 solver could not practically explore those paths, and we directly excluded the infeasible path cases manually. After that, we then examined the false positive and false negative cases.

[3]ID 5, 11, 13, 15, 27, 32, 35, 36, 37, 38, 39, 48 in Table VII

[4]ID 41, 44, 45 in Table VII

TABLE V
ANALYSIS TIME FOR MM FUNCTION IDENTIFICATION, MOS GENERATION, AND BUG DETECTION (IN MINUTES)

	MMFI	MG	BD	Total
Linux	61	38	328	427
FreeBSD	11	3	20	34
OpenSSL	2	1	3	6.5
Redis	1.5	3	1	5.5
Azure	1.07	0.61	0.22	1.90
QcloudE	0.28	0.64	0.10	1.02
QcloudH	0.26	0.60	0.07	0.93

MMFI : MM function identification;
MG : MOS generation; **BD** : Bug detection

False positives. For the 145 issues left for a manual bug verification, we confirmed that GOSHAWK reported 33, 7, 0, 0 and 13 false warnings for Linux kernel, FreeBSD kernel, OpenSSL, Redis and IoT SDKs, respectively. We found that the root causes of those false positive cases are mainly due to the weakness of the memory model [28] of CSA engine; CSA sometimes incorrectly models two pointer variables (actually pointing to two different addresses) with the same symbolic values, thus causing false positives. We leave the enhancement of alias analysis as a future effort. Considering that GOSHAWK has analyzed millions of code and introduced more than 10,000 custom MM functions, such a precision is promising. Also, the number of the reported bugs is manageable for a manual audit.

False negatives. Since we did not have a ground truth to verify the false negative rate against the seven tested projects, we chose another bug list to evaluate GOSHAWK. The bug list contained 31 use-after-free bugs in Linux kernel 4.19 found by SinkFinder tool, which also utilized annotation-assisted detection (we list the details of those bugs in Table VI in Appendix[5]). GOSHAWK successfully detected 29 of those 31 bugs; also we found that GOSHAWK could not detect the left two bugs because it failed to identify the related deallocators `cfg80211_put_bss` and `nfc_put_device` due to the issues mentioned in Section V-C. Actually, we manually added `cfg80211_put_bss` and `nfc_put_device` into our custom MM function set and generated MOS for them. This time GOSHAWK detected the remaining two bugs successfully.

Answer to RQ2: GOSHAWK achieved a 63.4% of precision for reported bugs, and found new bugs that other annotation-based detection tools were not able to discover. Moreover, the MOS based bug description is more concise for analysts to understand.

F. Time and Performance Analysis

By utilizing MOS to simplify the code exploration, GOSHAWK avoids scalability issues in bug detection and

[5]Note that we excluded four reference count related bugs mentioned in SinkFinder paper since this is out of the scope of this paper.

can efficiently analyze complex projects. Table V shows the analysis time costs for GOSHAWK to check each of the seven tested projects. For the most time-consuming bug detection process against Linux kernel, GOSHAWK spent 427 minutes to execute the whole analysis. Specifically, it took 61 minutes to perform NLP-assisted classification and data flow analysis to identify MM functions from source codes. Then, 38 minutes were used to generate MOS for the identified MM functions. With the generated MOS, GOSHAWK took 328 minutes to detect bugs. In comparison, as we show in Section V-B, if we did not restrict the exploration scope, the analysis would have required unacceptable time even with a more powerful hardware configuration. For other projects, GOSHAWK could complete the bug detection in less than one hour. These results show that we can deploy GOSHAWK to implement a daily bug checking even for large projects such as the Linux kernel.

We further investigated the identified MM functions and found 988 out of 4,721 allocators contain a long (>5) call chain from the entry to the primitive allocators. For deallocation functions, the results were similar. Since GOSHAWK identified those MM functions and modeled them as MOS, it removed the internal data flow of all identified MM functions and thus reduced the length of data flow. On average, GOSHAWK reduces the data flow length by removing 2.99 functions for custom allocation and 2.54 functions for custom deallocation.

The use of NLP significantly boosts the entire MM function identification. Taking the identification against Linux Kernel as an example, we found that the NLP-assisted classification takes 22 minutes to classify 549,187 functions, and the subsequent data flow analysis based validation only takes 39 minutes to check all those functions. In fact, we ran a data flow analysis to check all functions in Linux Kernel without applying the NLP-assisted classification, and found that the experiment could not terminate and suffered from out-of-memory crash on our server with 192G RAM.

Answer to RQ3: By using MOS to facilitate bug detection, GOSHAWK avoids a large number of redundant explorations and addresses the path explosion issue. The NLP-assisted classification also boosts the MM function identification.

G. Comparison with Related Works

MM Function Identification. GOSHAWK significantly outperforms K-MELD and SinkFinder in terms of the number of identified MM functions (Table IV). Besides, the identification result of GOSHAWK is more extensive. K-MELD and SinkFinder were only applicable to Linux kernel. GOSHAWK is designed to broadly analyze source code with different scales and is able to accurately find MM functions in both large and small projects.

Bug Detection. We further compare GOSHAWK to K-MELD and SinkFinder in bug detection. Since K-MELD and SinkFinder were not specifically designed to detect use-after-free and double-free bugs, we adopted an indirect comparison between GOSHAWK and them. We applied the similar analysis process but only utilized the MM functions labeled by K-MELD and SinkFinder to help detect bugs in Linux kernel. We first added those MM functions into GOSHAWK but only as primitive allocators/deallocators (i.e., adopting a simple memory object model), and examined the results. We found that the MM function sets of both K-MELD and SinkFinder were inaccurate and caused false alarm: the number of bugs reported by separately using MM function sets of K-MELD and SinkFinder were 4,742 and 999, and GOSHAWK only re-detected 18 and 8 bugs among them. This shows that the bug detection models of K-MELD and SinkFinder are imprecise. They cannot handle nested allocation and unpaired uses of MM functions and thus are expected to incur high false positives and false negatives. Actually, even though SinkFinder has considered the custom function issues and annotates several function (pairs), its pair-based detection (against Linux kernel 4.19) still misses all 14 use-after-free bugs that are detected by GOSHAWK[6].

Hence we further applied the same data flow analysis validation and MOS generation procedures against their MM function sets to refine the memory model, and only preserved 663/160 MM functions. By utilizing their MOS information, the new detection results become much better: only 57/23 bugs are reported and 20/10 of them are valid after manual inspection. This demonstrates that our structure-aware and object-centric analysis improves the analysis precision effectively.

VI. DISCUSSION

Scalability Bottlenecks. Our evaluation has demonstrated that the depth and breadth of code exploration are the main factors impacting the analysis complexity: to enumerate more execution paths, the complexity of exploration increases exponentially, which is the well-known path-explosion problem. Hence our current analysis applies different thresholds to limit the range of code exploration to ensure that the analysis time and required hardware resources are practical (note that the settings guarantee that at least every function is explored).

We identify two potential strategies to improve the scalability and coverage. The first strategy is to reduce the number of paths for symbolic execution. Path-based symbolic execution [29], [30], concolic execution [31], [32], and selective symbolic execution [33] have shown to be highly scalable to complex programs. To adopt such a strategy, one can first apply static analysis or heuristics to select potentially buggy paths and focus the symbolic execution on only them to improve scalability. The second strategy is to improve the performance of the symbolic execution engine. SymCC [34] adopts a compilation-based approach (as opposed to the traditional interpretation-based approach [35]) to symbolic execution and improves the performance by up to three orders of magnitude. Path merging [36], [37], [38], on the other hand, reduces the number of paths by merging paths at the same program location that are similar. We believe that these two strategies are orthogonal to GOSHAWK and, when adopted, can further improve the scalability of GOSHAWK. Once the

[6]ID: 1, 5, 23, 32, 33, 35, 36, 37, 38, 39, 41, 45, 46, 48 in Table VII. These bugs existed in both Linux kernel 4.19 and 5.12

scalability is improved, we can then drop the thresholds to also improve coverage.

Sources of False Positives/Negatives. Many inherent challenges of both control-flow and data-flow analysis (e.g., indirect call resolution, alias analysis, loop analysis) are still not well addressed by state-of-the-art code analysis engines and would lead to imprecise results. Such imprecision naturally causes false positives and negatives. Since GOSHAWK leverages an existing code analysis engine as its underlying infrastructure, its bug detection has both false positives and false negatives. GOSHAWK focuses on memory corruption bug detection instead of addressing those well-known program analysis deficiencies. We argue that these aspects are orthogonal to GOSHAWK and we could benefit from their improvements.

Our experiments show that by only utilizing NLP-assisted MM function classification, although it could capture most MM functions, GOSHAWK would incorrectly identify many non-MM functions and thus incur false positives. When the follow-up data flow analysis was applied to eliminate non-MM functions, it unfortunately filtered out a small portion of real MM functions and caused false negatives. Since most bug detection systems tend to reduce false positives prior to false negatives, our design combined the NLP-assisted classification and data flow analysis based validation to prove all the identified MM functions are accurate. Also, the result in Section V-C demonstrates that GOSHAWK has acceptable false positive and false negative rates.

VII. Related Work

A. Memory Corruption Bug Detection

K-Miner [39] carries a scalable pointer analysis and interprocedural analysis to uncover memory corruption bugs. Hua *et al.* [40] used machine learning to mitigate the imprecision of pointer analysis when detecting use-after-free bugs in large programs. They learned the correlations between program features and pointer aliases to filter out ambiguous aliases and then detected use-after-free bugs. CRED [41] is a path-sensitive and pointer analysis based tool for detecting use-after-free bugs. It scales down exponential numbers of contexts by a spatio-temporal context reduction technique and achieves low false positive rates with a multi-stage analysis. DCUAF [42] statically detects concurrency use-after-free bugs in kernel drivers with a set of paired driver interface functions which can be executed concurrently. However, these approaches need a set of annotated allocation and deallocation functions from code base to perform source-to-sink analysis for detecting memory corruption bugs.

B. MM Function Identification Approaches

Pair-based Mining. Pair-based mining analyzes the relationships among functions and matches these functions in pairs. WYSIWIB [43] extracts data dependencies among functions and identifies pairs of MM functions. Similarly, SinkFinder uses data dependencies to find function pairs. It uses an analogical reasoning mechanism to infer function pairs similar to a given seed pair. PairMiner [10] also conducts similarity comparison by using multiple keywords matching. Giving the function pairs, it searches for the similar pairs of functions.

Some other approaches pinpoint function pairs relying on tracking paths of error handling. PF-Miner [11] and BP-Miner [44] track both normal execution paths and error handling paths to recognize function pairs. K-MELD [6] adopts the frequent pattern of <allocation call, pointer check, error handling release, return> to identify allocation functions and the matched deallocation functions from error handling paths. HERO [7] relies on the pattern of paired function invocation on normal execution paths and on reversely ordered error handling paths to mine the paired functions.

Routine-based mining. Routine-base mining identifies functions by abstracting the characteristics of a specific type of functions. MemBrush executes programs and tracks the custom memory allocators and deallocators. Based on the execution flows, it then searches for functions that comply with these flows in C/C++ binaries. DynPTA [45] relies on analyzing the patterns of memory-allocation wrapper functions such as invoking a libc memory-allocation function. Through the patterns, DynPTA recognizes function wrappers and locates the returned pointers that are related to memory allocation. However, DynPTA is unable to deal with memory allocation functions customized by developers.

Semantic-based mining. Targeting a specific type of functions, semantics-based mining depends on semantic features of specific functions to infer similar functions. Bai *et al.* [46] use keywords to extract potential allocation and deallocation functions by conducting a semantic analysis. Nonetheless, their approach requires manual efforts to filter out the irrelevant functions and insert the needed ones. SuSi [4] leverages a set of human-annotated functions to train a SVM classifier. The classifier is trained with a large number of human summarized semantics and data flow features. It is then uses to predict source and sink functions in Android framework. The effectiveness of the classifier highly relies on the annotated functions and features; thus such an approach requires a high human effort. Instead of summarizing semantic features of a group of specific functions, NLP-EYE[3] infers semantic meanings of functions by comparing function prototypes with a set of known functions. Although it could recognize function semantics by analyzing function prototypes, its result is imprecise because only few programming language and natural language are involved.

VIII. Conclusion

We introduce MOS, a novel concept to implement structure-aware and object-centric MM behavior summarization, and use it to help detect complex memory bugs with characteristics of nested allocation and unpaired uses of MM functions. Our implemented MOS-enhanced bug detection system GOSHAWK, combines NLP and data flow analysis to identify MM functions, and finally finds 92 new bugs in recent versions of Linux kernel, FreeBSD kernel, OpenSSL, Redis, and three IoT SDKs via a MOS-enhanced memory object tracking.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their valuable feedback. We specially thank Bodong Li from HiSilicon for helping improve GOSHAWK. This work was partially supported by the National Key Research and Development Program of China (No.2020AAA0107803) and SJTU-HiSilicon Research Grant (YBN2019125153). Yunlong Lyu was funded by the National Natural Science Foundation of China (U19B2023). Yunlong Lyu (*lyl2019@mail.ustc.edu.cn*) and Juanru Li (*mail@lijuanru.com*) are corresponding authors.

REFERENCES

[1] **Linux Kernel Security Done Right** . https://security.googleblog.com/2021/08/linux-kernel-security-done-right.html. Accessed 2021.

[2] **Lessons from Building Static Analysis Tools at Google.** https://cacm.acm.org/magazines/2018/4/226371-lessons-from-building-static-analysis-tools-at-google/fulltext. Accessed 2021.

[3] Jianqiang Wang and Siqi Ma and Yuanyuan Zhang and Juanru Li and Zheyu Ma and Long Mai and Tiancheng Chen and Dawu Gu, "**NLP-EYE: Detecting Memory Corruptions via Semantic-aware Memory Operation Function Identification**," in 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), 2019.

[4] Siegfried Rasthofer and Steven Arzt and Eric Bodden, "**A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks**," in Annual Network and Distributed System Security Symposium (NDSS), 2014.

[5] Bian, Pan and Liang, Bin and Huang, Jianjun and Shi, Wenchang and Wang, Xidong and Zhang, Jian, "**SinkFinder: Harvesting Hundreds of Unknown Interesting Function Pairs with Just One Seed**," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020.

[6] Emamdoost, Navid and Wu, Qiushi and Lu Kangjie and McCamant Stephen, "**Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning**," in Annual Network and Distributed System Security Symposium (NDSS), 2021.

[7] Qiushi Wu and Aditya Pakki and Navid Emamdoost and Stephen McCamant and Kangjie Lu, "**Understanding and Detecting Disordered Error Handling with Precise Function Pairing**," in 30th USENIX Security Symposium (USENIX Security 21), 2021.

[8] **Clang Static Analyzer**. https://clang-analyzer.llvm.org/. Accessed 2021.

[9] Chen, Xi and Slowinska, Asia and Bos, Herbert, "**Who allocated my memory? Detecting custom memory allocators in C binaries**," in 2013 20th Working Conference on Reverse Engineering (WCRE), 2013.

[10] Liu, Hu-Qiu and Bai, Jia-Ju and Wang, Yu-Ping and Bian, Zhe and Hu, Shi-Min, "**Pairminer: mining for paired functions in Kernel extensions**," in 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015.

[11] Liu, Huqiu and Wang, Yuping and Jiang, Lingbo and Hu, Shimin, "**PF-Miner: A New Paired Functions Mining Method for Android Kernel in Error Paths**," in 2014 IEEE 38th Annual Computer Software and Applications Conference, 2014.

[12] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "PeX: A permission check analysis framework for linux kernel," in 28th USENIX Security Symposium (USENIX Security 19), 2019.

[13] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018.

[14] Kudo, Taku, "**Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates**," in Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2018.

[15] Sennrich, Rico and Haddow, Barry and Birch, Alexandra, "**Neural machine translation of rare words with subword units**," in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016.

[16] **StackExchange archive site**. https://archive.org/download/stackexchange/stackoverflow.com-Posts.7z. Accessed 2021.

[17] Bromley, Jane and Guyon, Isabelle and LeCun, Yann and Säckinger, Eduard and Shah, Roopak, "**Signature verification using a "Siamese" time delay neural network**," in Proceedings of the 6th International Conference on Neural Information Processing Systems, 1993.

[18] Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N and Kaiser, Łukasz and Polosukhin, Illia, "**Attention is all you need**," in Advances in Neural Information Processing Systems 30, 2017.

[19] **Linux memory management APIs.** https://www.kernel.org/doc/html/latest/core-api/mm-api.html. Accessed 2021.

[20] **GNU libc.** https://www.gnu.org/software/libc/. Accessed 2021.

[21] de Moura, Leonardo and Bjørner, Nikolaj, "**Z3: An Efficient SMT Solver**," in Tools and Algorithms for the Construction and Analysis of Systems, 2008.

[22] **MallocChecker.** https://clang.llvm.org/doxygen/MallocChecker_8cpp_source.html. Accessed 2021.

[23] **Range based constraint manager.** https://clang.llvm.org/doxygen/RangeConstraintManager_8cpp_source.html. Accessed 2021.

[24] **CodeChecker: A static analysis infrastructure built on the LLVM/Clang Static Analyzer toolchain**. https://github.com/Ericsson/codechecker. Accessed 2021.

[25] **Cross Translation Unit (CTU) Analysis.** https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html. Accessed 2021.

[26] G. Horváth, P. Szécsi, Z. Gera, D. Krupp, and N. Pataki, "Poster: Implementation and evaluation of cross translation unit symbolic execution for c family languages," in 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), 2018.

[27] **Clang: a C language family frontend for LLVM.** https://clang.llvm.org/. Accessed 2021.

[28] Z. Xu, T. Kremenek, and J. Zhang, "A memory model for static analysis of c programs," in Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I, 2010.

[29] Xu, Meng and Qian, Chenxiong and Lu, Kangjie and Backes, Michael and Kim, Taesoo, "**Precise and scalable detection of double-fetch bugs in OS kernels**," in 2018 IEEE Symposium on Security and Privacy (SP), 2018.

[30] Fraser Brown and Deian Stefan and Dawson Engler, "**Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code**," in 29th USENIX Security Symposium (USENIX Security 20), 2020.

[31] Shoshitaishvili, Yan and Wang, Ruoyu and Salls, Christopher and Stephens, Nick and Polino, Mario and Dutcher, Audrey and Grosen, John and Feng, Siji and Hauser, Christophe and Kruegel, Christopher and Vigna, Giovanni, "**SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis**," in IEEE Symposium on Security and Privacy, 2016.

[32] Yun, Insu and Lee, Sangho and Xu, Meng and Jang, Yeongjin and Kim, Taesoo, "**{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing**," in 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018.

[33] Chipounov, Vitaly and Kuznetsov, Volodymyr and Candea, George, "**The S2E platform: Design, implementation, and applications**," ACM Transactions on Computer Systems (TOCS), 2012.

[34] Poeplau, Sebastian and Francillon, Aurélien, "**Symbolic execution with SymCC: Don't interpret, compile!**" in 29th {USENIX} Security Symposium ({USENIX} Security 20), 2020.

[35] Cadar, Cristian and Dunbar, Daniel and Engler, Dawson R and others, "**Klee: unassisted and automatic generation of high-coverage tests for complex systems programs**," in OSDI, 2008.

[36] Kuznetsov, Volodymyr and Kinder, Johannes and Bucur, Stefan and Candea, George, "**Efficient state merging in symbolic execution**," Acm Sigplan Notices, 2012.

[37] Babic, Domagoj and Hu, Alan J, "**Calysto: scalable and precise extended static checking**," in Proceedings of the 30th international conference on Software engineering, 2008.

[38] Avgerinos, Thanassis and Rebert, Alexandre and Cha, Sang Kil and Brumley, David, "**Enhancing symbolic execution with veritesting**," in Proceedings of the 36th International Conference on Software Engineering, 2014.

[39] David Gens and Simon Schmitt and Lucas Davi and Ahmad-Reza Sadeghi, "**K-Miner: Uncovering Memory Corruption in Linux**," in Annual Network and Distributed System Security Symposium, NDSS, 2018.

[40] Yan, Hua and Sui, Yulei and Chen, Shiping and Xue, Jingling, "**Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection**," in Proceedings of the 33rd Annual Computer Security Applications Conference, 2017.
[41] Yan, Hua and Sui, Yulei and Chen, Shiping and Xue, Jingling, "**Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-after-Free Vulnerabilities**," in Proceedings of the 40th International Conference on Software Engineering, 2018.
[42] Jia-Ju Bai and Julia Lawall and Qiu-Liang Chen and Shi-Min Hu, "**Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers**," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019.
[43] Lawall, Julia L. and Brunel, Julien and Palix, Nicolas and Hansen, Rene Rydhof and Stuart, Henrik and Muller, Gilles, "**WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code**," in 2009 IEEE/IFIP International Conference on Dependable Systems Networks, 2009.
[44] Liu, Hu-Qiu and Bai, Jia-Ju and Wang, Yu-Ping and Hu, Shi-Min, "**BP-Miner: Mining Paired Functions from the Binary Code of Drivers for Error Handling**," in 2014 21st Asia-Pacific Software Engineering Conference, 2014.
[45] T. Palit and J. Moon and F. Monrose and M. Polychronakis, "**DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection**," in 2021 2021 IEEE Symposium on Security and Privacy (SP), 2021.
[46] Bai, Jia-Ju and Liu, Hu-Qiu and Wang, Yu-Ping and Hu, Shi-Min, "**Runtime Checking for Paired Functions in Device Drivers**," in Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference - Volume 01, 2014.

APPENDIX

A. MM Function Classification Model Generating

ULM-based Segmentation. We construct a programming corpus to support function prototype segmentation based on the occurrence of informal terms. First, we collect 20G questions (including source code and description of questions) posted on StackOverflow [16]. By taking these questions as input, GOSHAWK removes all punctuation and generates meaningful *subwords* by utilizing BPE [15] algorithm. BPE algorithm initializes all the preprocessed contents as a sequence of characters and iteratively merges the characters into different units. By computing the occurrence frequency of each unit, GOSHAWK regards the unit with the highest frequency as a subword and adds the subword and its frequency to the programming corpus.

Based on subwords in the programming corpus, GOSHAWK segments each function prototype in different ways. To select the most suitable segmentation, it calculates an occurrence probability of each possible segmentation form by

$$P(res) = \prod_{w \in subwords}^{subwords} P(w) \tag{1}$$

where $P(res)$ denotes the occurrence probability of each segmentation form and $P(w)$ denotes the occurrence probability of each subword. Finally, GOSHAWK selects the segmentation form that has the highest probability.

Reference Set Creation. In order to recognize custom MM functions, we manually create a reference set for function comparison. To ensure the variety and representativeness of the reference set, we extend the *function prototype corpus* by iteratively labeling more MM functions (e.g., from Linux kernel) and training the Siamese network. Finally, 4,441 functions (1,807 memory allocation functions and 2,634 memory deallocation functions) are included in the reference set.

Siamese Network Training. GOSHAWK trains a Siamese network to convert each subword list into a numeric vector (containing natural language semantics) and classifies each function prototype to a reference vector. Given the created function prototype corpus where each function is classified as a memory allocation function, a memory deallocation function, or a non-MM function, GOSHAWK randomly pairs function (f_i, f_i') with ground truth pairing information $y_i \in \{+1, -1\}$, where $y_i = +1$ indicates that prototypes of function f_i and f_i' are in the same category, $y_i = -1$ otherwise. We denote the vector of function prototypes f_i and f_i' as $\vec{e_i}$ and $\vec{e_i'}$, respectively. The output of the Siamese network for each pair is the cosine similarity between $\vec{e_i}$ and $\vec{e_i'}$.

MM Candidate Selection. By using the reference set and the trained Siamese network, GOSHAWK classifies MM functions from the tested project. Since we only aim to identify memory allocation and deallocation functions, we only consider the vectors of the corresponding MM functions. Specifically, let $\vec{a}_1, ..., \vec{a}_{na}$ denote the reference vectors of the allocation functions in the training function set and $\vec{d}_1, ..., \vec{d}_{nd}$ denote the reference vectors of the deallocation functions in the training function set. Subscripts na and nd denote the number of allocation and deallocation functions in the training function set, respectively. To enhance comparison efficiency, we compress the semantics of the *reference vectors* into vectors $\vec{a}_m$ and $\vec{d}_m$ by averaging the weight of each function as follow:

$$\vec{a}_m = \frac{1}{na}\sum_{i}^{na}\frac{\vec{a}_i}{||\vec{a}_i||}, \quad \vec{d}_m = \frac{1}{nd}\sum_{i}^{nd}\frac{\vec{d}_i}{||\vec{d}_i||} \tag{2}$$

Then, for an unlabeled function f_t we generate its prototype vector $\vec{v}_t$ by using the trained Siamese network. After calculating the cosine similarity between $\vec{v}_t$, $\vec{a}_m$ and $\vec{v}_t$, $\vec{d}_m$, GOSHAWK can distinguish the type of f_t if any of the two similarity scores is higher than a threshold. And if so, GOSHAWK assigns f_t to its corresponding candidate set.

B. Official MM function set

The collected official memory allocation functions are: `malloc`, `kmalloc`, `kmalloc_array`, `krealloc_array`, `kcalloc`, `kzalloc`, `kzalloc_node`, `vmap`, `vmalloc`, `vzalloc`, `__vmalloc_node`, `vmalloc_no_huge`, `vmalloc_user`, `vmalloc_node`, `vzalloc_node`, `vmalloc_32`, `vmalloc_32_user`, `kmem_cache_alloc`, `kmem_cache_alloc_node`, `kvmalloc_node`, `ioremap` and `mmap`. The collected official memory deallocation functions are: `free`, `kfree`, `vfree`, `kmem_cache_free`, `kfree_sensitive`, `kfree_const`, `kvfree`, `iounmap`, `vunmap` and `munmap`.

TABLE VI
GROUND TRUTH BUGS DETECTION RESULT FOR LINUX-4.19

PATCH ID	Deallocator	Success/Failure
1016104	f2fs_put_page	Success
1016725	iput	Success
1016937	iput	Success
1018023	dev_kfree_skb_any	Success
1018816	usb_free_urb	Success
1151291	video_device_release	Success
1149734	kfree_skb	Success
1148926	kfree_skb	Success
1148789	dma_free_coherent	Success
1016402	dput	Success
1016714	dput	Success
1016907	dput	Success
1149388	dma_fence_put	Success
1149298	dma_fence_put	Success
1149083	kmem_cache_destroy	Success
1148731	devm_kfree	Success
1147894	mempool_free	Success
1017915	pci_dev_put	Success
1017916	pci_dev_put	Success
1017916	pci_dev_put	Success
1149409	pci_dev_put	Success
1149287	dma_pool_free	Success
1149266	i915_gem_object_put	Success
1016991	posix_acl_release	Success
1149186	dst_release	Success
1149184	dst_release	Success
1018030	free_netdev	Success
1016628	hfs_bnode_put	Success
1016665	hfsplus_bnode_put	Success
1018810	cfg80211_put_bss	Failure
1149816	nfc_put_device	Failure

TABLE VII. Previously unknown bugs found in Linux, FreeBSD, OpenSSL, Redis and IoT SDKs

ID	Program	File:line_number	Buggy Function	MM Function	Type	Confirmed	MallocChecker
1	Linux	drivers/gpu/drm/xen/xen_drm_front.c:555	xen_drm_drv_init	kfree	UAF	✓	✓
2	Linux	drivers/infiniband/sw/siw/siw_mem.c:116	siw_alloc_mr	siw_mem_put	UAF	✓	✗
3	Linux	drivers/scsi/myrs.c:2281	myrs_cleanup	iounmap	DF	✓	✗
4	Linux	drivers/ipack/carriers/tpci200.c:600	tpci200_pci_probe	tpci200_uninstall	DF	✓	✗
5	Linux	drivers/net/ethernet/qlogic/qlcnic/qlcnic_minidump.c:1424	qlcnic_83xx_get_minidump_template	vfree	UAF	✓	✗
6	Linux	lib/test_kmod.c:1148	register_test_dev_kmod	vfree	UAF		✗
7	Linux	drivers/dma/dmaengine.c:1088	dma_async_device_register	free_percpu	DF	✓	✗
8	Linux	drivers/net/ethernet/myricom/myri10ge/myri10ge.c:2897	myri10ge_sw_tso	dev_kfree_skb_any	UAF	✓	✗
9	Linux	drivers/net/ethernet/netronome/nfp/bpf/cmsg.c:456	nfp_bpf_ctrl_msg_rx	dev_kfree_skb_any	UAF	✓	✗
10	Linux	drivers/net/wireless/intersil/hostap/hostap_80211_rx.c:1019	hostap_80211_rx	dev_kfree_skb_any	UAF		✗
11	Linux	drivers/net/wireless/marvell/mwifiex/tdls.c:859	mwifiex_send_tdls_action_frame	dev_kfree_skb_any	DF	✓	✗
12	Linux	drivers/net/wireless/ath/ath10k/htc.c:656	ath10k_htc_send_bundle	dev_kfree_skb_any	UAF	✓	✗
13	Linux	drivers/crypto/qat/qat_common/adf_transport.c:173	adf_create_ring	dma_free_coherent	DF	✓	✗
14	Linux	drivers/net/ethernet/broadcom/bcm4908_enet.c:174	bcm4908_enet_dma_alloc	dma_free_coherent	DF	✓	✗
15	Linux	drivers/net/wireless/marvell/mwl8k.c:1474	mwl8k_probe_hw	dma_free_coherent	DF	✓	✗
16	Linux	drivers/net/wireless/intel/iwlwifi/queue/tx.c:1101	iwl_txq_dyn_alloc_dma	dma_free_coherent	DF		✗
17	Linux	drivers/mtd/nand/raw/gpmi-nand/gpmi-nand.c:2477	gpmi_nand_init	dma_free_coherent	DF	✓	✗
18	Linux	drivers/scsi/be2iscsi/be_mgmt.c:533	beiscsi_if_clr_ip	dma_free_coherent	UAF		✗
19	Linux	drivers/gpu/drm/i915/gt/gen8_ppgtt.c:631	gen8_preallocate_top_level_pdp	i915_gem_object_put	UAF	✓	✗
20	Linux	drivers/misc/ibmasm/remote.c:265	ibmasm_init_one	input_free_device	UAF	✓	✗
21	Linux	drivers/misc/ibmasm/remote.c:266	ibmasm_init_one	input_free_device	UAF	✓	✗
22	Linux	net/tipc/socket.c:1268	tipc_sk_mcast_rcv	kfree_skb	DF	✓	✗
23	Linux	drivers/net/ethernet/qualcomm/emac/emac-mac.c:1459	emac_mac_tx_buf_send	kfree_skb	UAF	✓	✗
24	Linux	drivers/net/ethernet/cisco/enic/enic_main.c:860	enic_hard_start_xmit	kfree_skb	UAF	✓	✗
25	Linux	net/ipv6/ip6_tunnel.c:1439	ip6_tnl_start_xmit	kfree_skb	DF		✗
26	Linux	drivers/scsi/bnx2fc/bnx2fc_fcoe.c:444	bnx2fc_rcv	kfree_skb	DF		✗
27	Linux	net/nfc/digital_dep.c:1287	digital_tg_recv_dep_req	kfree_skb	DF	✓	✗
28	Linux	drivers/video/fbdev/hyperv_fb.c:1273	hvfb_probe	framebuffer_release	DF	✓	✓
29	Linux	drivers/net/wan/hdlc_fr.c:417	pvc_xmit	__skb_pad	DF	✓	✗
30	Linux	drivers/firmware/efi/efi.c:937	efi_mem_reserve_persistent	memunmap	UAF	✓	✗
31	Linux	net/rds/message.c:350	rds_message_map_pages	rds_message_put	UAF	✓	✗
32	Linux	drivers/infiniband/ulp/isert/ib_isert.c:477	isert_connect_request	isert_device_put	UAF	✓	✗
33	Linux	drivers/scsi/st.c:1272	st_open	scsi_tape_put	UAF	✓	✗
34	Linux	drivers/nvme/target/rdma.c :803	nvmet_rdma_write_data_done	nvmet_rdma_release_rsp	UAF	✓	✗
35	Linux	drivers/block/drbd/drbd_nl.c:4917	get_initial_state	nlmsg_free	UAF	✓	✗
36	Linux	drivers/block/drbd/drbd_nl.c:4924	get_initial_state	nlmsg_free	UAF	✓	✗
37	Linux	drivers/block/drbd/drbd_nl.c:4930	get_initial_state	nlmsg_free	UAF	✓	✗
38	Linux	drivers/block/drbd/drbd_nl.c:4936	get_initial_state	nlmsg_free	UAF	✓	✗
39	Linux	drivers/block/drbd/drbd_nl.c:4942	get_initial_state	nlmsg_free	UAF	✓	✗
40	Linux	drivers/scsi/mpt3sas/mpt3sas_scsih.c:11438	pcie_device_make_active	pcie_device_put	UAF		✗
41	Linux	sound/isa/sb/emu8000.c:1029	snd_emu8000_create_mixer	snd_ctl_free_one	UAF	✓	✗
42	Linux	drivers/misc/habanalabs/gaudi/gaudi.c:5615	gaudi_memset_device_memory	hl_cb_put	UAF	✓	✗
43	Linux	drivers/infiniband/hw/bnxt_re/qplib_res.c:853	bnxt_qplib_alloc_res	pci_iounmap	DF	✓	✗
44	Linux	drivers/block/null_blk/main.c:1980	null_init	null_free_zoned_dev	DF	✓	✗
45	Linux	sound/isa/sb/sb16_csp.c:1048	snd_sb_qsound_build	snd_ctl_free_one	UAF	✓	✗
46	Linux	sound/isa/sb/sb16_csp.c:1050	snd_sb_qsound_build	snd_ctl_free_one	UAF	✓	✗
47	Linux	drivers/acpi/acpica/dbnames.c:561	acpi_db_walk_for_fields	acpi_os_free	UAF		✗
48	Linux	drivers/media/platform/exynos4-is/fimc-isp-video.c:314	isp_video_release	v4l2_fh_release	UAF	✓	✗
49	Linux	fs/orangefs/orangefs-cache.c:49	purge_waiting_ops	op_release	UAF		✗
50	FreeBSD	netsmb/smb_rq.c:742	smb_t2_request_int	smb_rq_done	UAF	✓	✗
51	FreeBSD	net/rtsock.c:979	update_rtm_from_rc	free	UAF	✓	✓
52	FreeBSD	netgraph/ng_checksum.c:687	ng_checksum_rcvdata	m_freem	DF	✓	✗
53	FreeBSD	netpfil/ipfw/dn_aqm_codel.c:273	aqm_codel_enqueue	m_freem	DF	✓	✗
54	FreeBSD	netpfil/ipfw/dn_aqm_pie.c:561	aqm_pie_enqueue	m_freem	DF	✓	✗
55	FreeBSD	netpfil/ipfw/dn_sched_fq_codel.c:205	codel_enqueue	m_freem	DF	✓	✗
56	FreeBSD	netpfil/ipfw/dn_sched_fq_pie.c:751	pie_enqueue	m_freem	DF	✓	✗
57	FreeBSD	dev/cxgb/cxgb_sge.c:2784	get_packet	m_freem	UAF	✓	✗
58	FreeBSD	dev/oce/oce_if.c:1229	oce_tx	m_freem	DF	✓	✗
59	FreeBSD	contrib/ipfilter/netinet/ip_nat.c:6253	ipf_nat_rule_deref	KFREE	UAF	✓	✓
60	FreeBSD	contrib/ngatm/netnatm/msg/uni_ie.c:7149	DEF_IE_ENCODE	uni_msg_destroy	UAF	✓	✗
61	FreeBSD	dev/acpica/acpi_pci_link.c:916	acpi_pci_link_route_irqs	AcpiOsFree	DF	✓	✓

62	FreeBSD	dev/ocs_fc/ocs_hw.c:11806	ocs_hw_async_call	ocs_free	DF	✓	✓
63	FreeBSD	dev/ocs_fc/ocs_sport.c:266	ocs_sport_free	ocs_free	UAF	✓	✓
64	FreeBSD	dev/qlnx/qlnxe/ecore_spq.c:1012	ecore_spq_post	OSAL_FREE	UAF	✓	✓
65	FreeBSD	kern/uipc_socket.c:485	sodealloc	crfree	UAF	✓	✗
66	FreeBSD	netgraph/bluetooth/hci/ng_hci_evnt.c:541	le_connection_complete	ng_hci_free_con	UAF	✓	✗
67	FreeBSD	rpc/rpcsec_gss/rpcsec_gss.c:595	rpc_gss_marshal	mem_free	DF		✗
68	OpenSSL	penssl/apps/cmp.c:1694	setup_request_ctx	X509_REQ_free	DF	✓	✗
69	OpenSSL	crypto/srp/srp_vfy.c:687	SRP_create_verifier_ex	BN_clear_free	DF	✓	✗
70	OpenSSL	crypto/ts/ts_rsp_verify.c:320	int_ts_RESP_verify_token	X509_ALGOR_free	DF	✓	✗
71	OpenSSL	engines/e_loader_attic.c:491	try_decode_PKCS8Encrypted	X509_SIG_free	DF	✓	✗
72	OpenSSL	test/evp_extra_test.c:577	test_EVP_PKEY_ffc_priv_pub	OSSL_PARAM_free	DF	✓	✗
73	OpenSSL	test/evp_extra_test.c:593	test_EVP_PKEY_ffc_priv_pub	OSSL_PARAM_free	DF	✓	✗
74	OpenSSL	test/evp_extra_test.c:609	test_EVP_PKEY_ffc_priv_pub	OSSL_PARAM_free	DF	✓	✗
75	OpenSSL	engines/e_loader_attic.c:709	try_decode_X509Certificate	X509_free	DF	✓	✗
76	OpenSSL	engines/e_loader_attic.c:210	new_EMBEDDED	OPENSSL_free	UAF	✓	✓
77	Redis	src/replication.c:3404	replicationCron	freeClient	UAF	✓	✗
78	Azure	iothub_service_client/src/iothub_messaging_ll.c:1575	IoTHubMessaging_LL_Send	message_destroy	DF	✓	✓
79	Azure	uamqp/src/amqp_definitions.c:10446	sasl_mechanisms_get_sasl_server_mechanisms	amqpvalue_destroy	DF		✗
80	Azure	src/uamqp_messaging.c:1184	readApplicationPropertiesFromuAMQPMessage	amqpvalue_destroy	DF		✓
81	Azure	uamqp/src/cbs.c:639	cbs_put_token_async	amqpvalue_destroy	DF		✗
82	QcloudE	samples/asr/asr_data_template_sample.c:416	main	HAL_Free	DF	✓	✓
83	QcloudE	samples/data_template/data_template_sample.c:520	main	HAL_Free	DF	✓	✓
84	QcloudE	samples/ota/ota_mqtt_sample.c:299	_get_local_fw_info	HAL_Free	UAF	✓	✓
85	QcloudE	samples/ota/ota_mqtt_sample.c:300	_get_local_fw_info	HAL_Free	DF	✓	✓
86	QcloudE	samples/scenarized/light_data_template_sample.c:837	main	HAL_Free	DF	✓	✓
87	QcloudE	sdk_src/services/asr/asr_client.c:674	IOT_Asr_Init	HAL_Free	UAF	✓	✓
88	QcloudE	sdk_src/services/data_template/data_template_client.c:870	IOT_Template_Construct	HAL_Free	DF	✓	✓
89	QcloudH	samples/mqtt/multi_thread_mqtt_sample.c:202	_mqtt_message_handler	HAL_Free	UAF	✓	✓
90	QcloudH	samples/multi_client/multi_client_shadow_sample.c:226	_shadow_client_thread_runner	HAL_Free	DF	✓	✓
91	QcloudH	samples/ota/ota_mqtt_sample.c:289	_get_local_fw_info	HAL_Free	UAF	✓	✓
92	QcloudH	samples/ota/ota_mqtt_sample.c:290	_get_local_fw_info	HAL_Free	DF	✓	✓