Making Memory Account Accountable: Analyzing and Detecting Memory Missing-account bugs for Container Platforms

Yutian Yang[*]
Zhejiang University
Hangzhou, China

Wenbo Shen[†]
Zhejiang University
Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province
Hangzhou, China

Xun Xie
Zhejiang University
Hangzhou, China

Kangjie Lu
University of Minnesota, Twin Cities
Minneapolis, United States

Mingsen Wang
Zhejiang University
Hangzhou, China

Tianyu Zhou
Zhejiang University
Hangzhou, China

Chenggang Qin
Ant Group
Hangzhou, China

Wang Yu
Ant Group
Hangzhou, China

Kui Ren
Zhejiang University
Hangzhou, China

ABSTRACT

Linux kernel introduces the memory control group (memcg) to account and confine memory usage at the process-level. Due to its flexibility and efficiency, memcg has been widely adopted by container platforms and has become a fundamental technique. While being critical, memory accounting is prone to missing-account bugs due to the diverse memory accounting interfaces and the massive amount of allocation/free paths. To our knowledge, there is still no systematic analysis against the memory missing-account problem, with respect to its security impacts, detection, etc.

In this paper, we present the first systematic study on the memory missing-account problem. We first perform an in-depth analysis of its exploitability and security impacts on container platforms. We then develop a tool named MANTA (short for Memory AccouNTing Analyzer), which combines both static and dynamic analysis techniques to detect and validate memory missing-account bugs automatically.

Our analysis shows that all container runtimes, including runC and Kata container, are vulnerable to memory missing-account-based attacks. Moreover, memory missing-account can be exploited to attack the Docker, the CaaS, and the FaaS platforms, leading to memory exhaustion, which crashes individual node or even the whole cluster. Our tool reports 53 exploitable memory missing-account bugs, 37 of which were confirmed by kernel developers with the corresponding patches submitted, and two new CVEs are assigned. Through the in-depth analysis, automated detection, the reported bugs and the submitted patches, we believe our research improves the correctness and security of memory accounting for container platforms.

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**; **Virtualization and security**.

KEYWORDS

Cloud infrastructure, Linux kernel, memory accounting, missing-account, DoS attack

ACM Reference Format:
Yutian Yang, Wenbo Shen, Xun Xie, Kangjie Lu, Mingsen Wang, Tianyu Zhou, Chenggang Qin, Wang Yu, and Kui Ren. 2022. Making Memory Account Accountable: Analyzing and Detecting Memory Missing-account bugs for Container Platforms. In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3564625.3564634

[*]Y. Yang, W. Shen, M. Wang, and K. Ren are also with the ZJU-Hangzhou Global Scientific and Technological Innovation Center and the Jiaxing Research Institute, Zhejiang University.
[†]Corresponding author.

1 INTRODUCTION

Accounting and limiting memory usage is a core functionality of every operating system kernel. In particular, Linux kernel introduces the memory control group (memcg), which can account and limit memory usage at the process-level. Therefore, compared with virtual-machine (VM) based memory partition techniques, memcg is more fine-grained and lightweight.

Due to its flexibility and efficiency, memcg has been widely adopted by container platforms. It is known that container platforms rely on resource sharing heavily to improve the utilization of their hardware resources. While memcg provides the only way to account and limit the memory usage of containers at the process-level. As a result, popular container platforms, such as Docker [17], the Container-as-a-Service platform (e.g., OpenShift) [23], and the Function-as-a-Service platform (e.g., OpenWhisk) [30], all adopt memcg to control the memory usage of their containers. Moreover, even the VM-based container runtime—Kata container [3] also uses memcg to limit the memory of container threads and other service threads on the host machine. Besides, Open Container Initiative (OCI) certification requires that container runtimes must use memcg in order to be certified [4]. Therefore, memcg has become a fundamental technique for container platforms and cloud computing.

Unfortunately, while being widely used, memcg is error-prone due to its complex nature. Specifically, Linux kernel chooses to insert memory accounting interfaces into the memory allocation/free paths to realize accounting, which may easily introduce memory missing-account (the allocated memory is not accounted) bugs due to the highly complicated memory usage and the massive number of memory usage-related interfaces and code paths. Moreover, even if the accounting interfaces are in the correct positions, their accounting behaviors are conditional and controlled by the accounting flags, which we find is often missing. Consequently, we find that memory missing-account bugs are common.

However, even memcg has been used widely, its memory missing-account bugs have never been analyzed systematically. The status quo of memcg resulted from two causes. First, it has never been clear what security impacts memory missing-account bugs would incur or whether they are exploitable at all. As a result, missing-account bugs were ignored by kernel developers, and no particular efforts have been put into eliminating these problems. Second, there is no automated detection of memory missing-account bugs. As previously discussed, memory accounting design and implementation are complex and deeply integrated into the complicated memory management subsystem of Linux kernel. As a result, it is hard to understand the accounting design and policy correctly, even for experienced users. For example, we find that the Docker official website mistakenly claims that "memory charge is split between the control groups" for the page cache, and that "when a cgroup is terminated, it could increase the memory usage of another cgroup" [17]. As a result, without the automated detection, it is virtually impossible to reason about the correctness of memory accounting manually. Even worse, whether memory missing-account bugs are detectable and how it can be detected are still open questions.

This paper thus conducts the first systematic analysis and detection against the Linux memory missing-account bugs. We first perform an in-depth analysis to understand the exploitability and security impacts of the memory missing-account bugs on container platforms. We then develop a tool named MANTA (short for Memory AccouNTing Analyzer), which combines both static and dynamic analysis techniques to automatically detect memory missing-account bugs. Our analysis shows that not only normal container runtimes (e.g., runC), but also secure container runtimes (e.g., Kata container) are vulnerable to memory exhaustion attacks caused by memory missing-account bugs. Even worse, memory missing-account bugs can be exploited to attack the Docker, the CaaS, and the FaaS platforms, which crashes container nodes or even the whole cluster.

Given the complexity of memcg, memory missing-account detection faces two challenges. First, there is no documentation nor existing study defining memory accounting interfaces used by memcg. Existing works [20, 37] adopt natural language processing or wrapper function analysis to identify memory allocation interfaces. However, these techniques cannot apply to accounting interface directly due to unmatched heuristics. To address the problem, MANTA proposes the counter-based interface identification to detect all memory accounting related interfaces automatically. The key observation is that all memory accounting interfaces finally increase/decrease corresponding page counters in their implementation. The identification starts from page counter modification and iterates on the whole call graph to identify accounting interfaces precisely.

Second, it is challenging to analyze the mappings between memory allocation and memory accounting. Such analysis is complicated by the deep and nested execution paths from memory allocation sites to memory accounting sites. It also needs to trace the data flow between the allocated pages and the accounted pages to get the correct alias-set. Moreover, the accounting could be conditional due to the control of conditional flags. Accordingly, MANTA proposes the alloc-charging mapping analysis and accounting flag analysis to address this challenge. Besides, to validate the detected memory missing-account bugs, MANTA further employs Linux Test Project to examine their triggerability and impacts.

With MANTA, we detect and report 53 exploitable bugs in memcg, 37 of which have been confirmed by kernel developers with patches that are either merged or pending for merging. The reported bugs got two new CVEs (one is pending). In summary, this work conducts an in-depth investigation that not only analyzes the exploitability and impacts but also systematically detects memory missing-account bugs. We believe the findings help kernel memcg developers improve memory accounting in the future. This paper makes the following contributions.

- **In-depth analysis of exploitability and impacts.** We design new attacks to exploit the memory missing-account problems on container platforms. Our attacks show that memory missing-account bugs can be easily exploited to attack both normal and secure container runtimes (i.e., runC and Kata container) and popular container platforms (i.e., Docker, OpenShift, and OpenWhisk), leading to container nodes or even the whole cluster crashes.
- **Automated detection with new techniques.** We propose multiple analysis techniques to effectively detect memory missing-account bugs, integrating both static analysis and dynamic validation. We implement the detection based on LLVM and evaluate it with the Linux memory accounting. We further use Linux Test Project (LTP) to validate the triggerability of the detected bugs.
- **Community impact.** We detect and report 53 exploitable memory missing-account bugs in Linux v5.10, 37 of which are confirmed by kernel developers. Two new CVEs are assigned (one is pending). Our findings raise awareness of missing-account impacts in the kernel community who also invited us to test their downstream kernels. We plan to open-source our detection tool to further help the community to improve the correctness and security of memory accounting.

Ethical considerations. All the experiments and attacks in this paper are conducted on a dedicated physical machine, which is used solely by us. We responsibly disclosed all detected bugs to the Linux

```
1  struct mem_cgroup {
2      ...
3      struct page_counter memory;/* Both v1 & v2 */
4      union {
5          struct page_counter swap;    /* v2 only */
6          struct page_counter memsw;    /* v1 only */
7      };
8      /* Legacy consumer-oriented counters */
9      struct page_counter kmem;    /* v1 only */
10     struct page_counter tcpmem;    /* v1 only */
11     ...
12 }
13 struct page_counter {
14     atomic_long_t usage;
15     ...
16     unsigned long max;
17     ...
18 }
19
20 bool page_counter_try_charge(struct page_counter *counter, unsigned
   ↪   long nr_pages, ...)
21 {
22     struct page_counter *c;
23     ...
24     new = atomic_long_add_return(nr_pages, &c->usage);
25     if (new > c->max) {
26         ...
27         goto failed;
28     }
29 }
```

Figure 1: Memcg structure and accounting code.

kernel developers and submitted patches for all confirmed ones. Moreover, we also reported false claims on the memory accounting to the Docker team.

2 BACKGROUND

In this section, we first give background knowledge on Linux memory accounting (i.e., memcg). Next, we introduce container platforms that use memcg.

2.1 Linux Memory Accounting

Memory accounting is a core functionality of every modern operating system kernel. Currently, Linux kernel employs the memory control group (memcg) to realize memory accounting. Memcg accounts for 4 types of memory: user, kernel, swap, and socket.

- **user** accounts all user space memory pages.
- **kernel** accounts kernel space memory pages and objects.
- **swap** accounts swap area pages.
- **socket** accounts the sock memory.

Memcg only accounts the memory for user space processes and skips the accounting on kernel daemons or internal memory usage. Note that Linux kernel v5.9 introduced *object cgroup* [7], which accounts for sub-page kernel memory usage, such as kernel objects. Object cgroup can charge objects to different cgroups, and thus eliminates the per-memcg slab and saves lots of slab memory [8].

Accounting interfaces. Linux kernel uses different accounting interfaces for different types of memory. Specifically, Linux kernel uses `mem_cgroup_charge` to account user and swap memory and `__memcg_kmem_charge` to account kernel memory. For kernel memory, Linux kernel also requires the `__GFP_ACCOUNT` flag to be set during the memory allocation to account for kernel pages. Users can interact with memcg through file interfaces. For example, a user can check the total memory usage from `memory.usage_in_bytes`.

A memcg instance may contain multiple processes whose memory usage is accounted in it. Memcg is organized in a tree hierarchy. As such, the memory limits on the parent node also affect all children nodes. There are two versions of memcg: v1 and v2. Their differences are mainly on hierarchical structures and user interfaces [34], with roughly the same implementations on accounting. Both of them are in use currently. This paper refers to cgroups v1, and most conclusions also hold on cgroups v2.

Data structures. The core data structure for memcg is `mem_cgroup`, as shown in Figure 1, which represents a memcg instance. `memcg` contains 4 page counters, corresponding to 4 types of accounted memory as previously discussed. Each page counter uses `usage` (Line 14) to count the number of allocated pages. While the total memory limit in pages is set in `max` (Line 16). At the memory charging, Linux kernel calls charging interfaces which in turn calls `page_counter_try_charge`, a function computing the total usage (Line 24) and checking it with the `max` limit (Line 25).

Accounting challenges. Accounting memory usage correctly in Linux kernel is very challenging due to numerous memory allocation interfaces and massive allocation paths in Linux kernel. It is difficult to mediate all memory allocation paths, and thus the current accounting mechanism is error-prone.

2.2 Container Runtimes and Platforms

Memcg can account and limit memory usage at per-process level. Compared with virtual-machine (VM) based memory control techniques, memcg is more fine-grained, lightweight, and flexible. Therefore, memcg has been widely adopted by Docker, the Container-as-a-Service (CaaS) and Function-as-a-Service (FaaS) platforms. Specifically, CaaS platforms provide users with provisioned container instances. Users on a CaaS platform can create/start/stop/delete containers with customized container images. On the other hand, instead of giving the users container instances, FaaS platforms allow users to input a function and triggering rules, and creates container instances automatically to execute the input function.

Docker and CaaS/FaaS platforms usually use the native container runtime (i.e., runC) for container instances. To improve the isolation among containers, people propose the secure container runtimes, such as gVisor and Kata container. gVisor is a sandboxed container runtime developed by Google. It runs each container on a user-space kernel called Sentry. Sentry intercepts and handles most syscalls from containers. Therefore, Sentry reduces the syscalls invoked on the host kernel from containers. Besides the sandboxed runtime, the container community also proposed to use virtualization to isolate container instances. One such virtualized container runtime is Kata container [24], in which each container instance runs in a micro-VM for strong isolation.

Our experiments show that both normal and secure container runtimes, including runC and Kata container, are vulnerable to memory exhaustion attacks caused by missing-account bugs. Moreover, memory missing-account bugs can be exploited to attack the Docker, the CaaS and the FaaS platforms, leading to memory exhaustion, which crashes individual nodes or even the whole cluster.

3 A STUDY OF EXPLOITABILITY AND IMPACT

The impact of memory missing-account bugs in production environments have never been systematically studied. They were generally considered as a less-harmful correctness issue, instead

```
1   static struct sem_array *sem_alloc(size_t nsems)
2   {
3       struct sem_array *sma;
4
5       if (nsems > (INT_MAX - sizeof(*sma)) / sizeof(sma->sems[0]))
6           return NULL;
7
8       sma = kvzalloc(struct_size(sma, sems, nsems), GFP_KERNEL);
9       if (unlikely(!sma))
10          return NULL;
11
12      return sma;
13  }
```

Figure 2: A missing-account bug in Linux kernel. This bug is detected and fixed by us and confirmed by Linux developers. Allocations at Line 8 are not charged. A new CVE is assigned.

of a security issue. In this section, we show that memory missing-account bugs can be exploited to cause memory exhaustion on container hosts and even the entire cloud platforms. In particular, we analyze the exploitability of memory accounting issues to understand its practical impacts on container platforms. We design new attacks to exploit missing-account bugs. These attacks show that these bugs can be used to attack both normal and secure container runtimes. Therefore, they can be exploited to cause DoS (host machine crashes) and financial charge problems on popular CaaS and FaaS container platforms. In the following, we first discuss the threat model and assumption. Next, we present details of exploiting memory missing-account problems.

3.1 Threat Model and Assumptions

In our experiments, we consider three container platforms—Docker, the Container-as-a-Service (CaaS) platform, and the Function-as-a-Service (FaaS) platform. We use the widely deployed CaaS platform—OpenShift as our CaaS platform and the popular OpenWhisk as our FaaS platform. These three platform use the native runtime (i.e., runC) as their container runtime. All containers on these three platforms are set up as non-privileged with default capabilities [2] and seccomp configurations [18]. For the Docker and CaaS platforms, users can create and start containers provisioned with self-defined images through cloud-provided interfaces. This is reasonable as all cloud vendors, including AWS, Google Cloud, and Azure, provide a console for users to manage the container instances [11]. For FaaS platforms, users can define, deploy, and trigger functions, where each function instance runs in a container.

The attacker is a malicious user using Docker, CaaS, and FaaS platforms. He/She can execute arbitrary code inside unprivileged containers. However, the capability of the container is restricted by the default capabilities. To further limit the attacker, we assume that attackers cannot escape from containers nor escalate their privileges. The attackers' goal is to break the limit of memory accounting and exhaust all memory on the host machine to crash other containers, functions or the host machine.

For the attacking environments, we set up the Docker on a local machine, while set up OpenShift and OpenWhisk clusters on Google Cloud. For ethical considerations, the cluster is built on a dedicated bare metal server provided by Google Cloud that is only used by us, and thus will not impact other users.

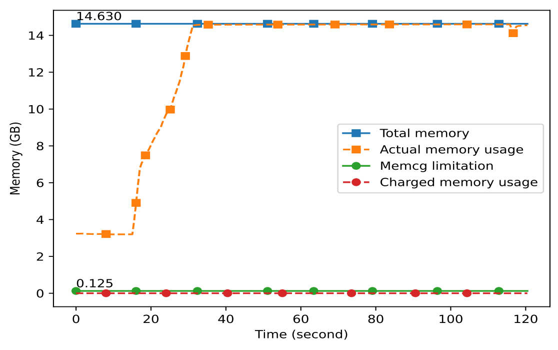

Figure 3: Memory exhaustion attacks on OpenShift.

3.2 Exploit Missing-Account Problem

In this section, we design new attacks to exploit the missing-account problem in memory accounting to understand its impact. As previously discussed, current memory accounting in the Linux kernel places memory accounting interfaces on the memory allocation paths and memory unaccounting interfaces on the memory free paths. However, Linux memory management is complex, containing thousands of memory allocation and free paths. It is hard to mediate all of them with the correct interfaces. Especially, it is very easy to miss one memory allocation paths, leading to the missing-account problem.

Figure 2 shows a missing-account bug on semaphore objects and semaphore arrays in the kernel, detected by MANTA and confirmed by the Linux developer. The function `sem_alloc` allocates a set of semaphores and uses `sem_array` to index them. The allocation site at Line 8 calls `kvzalloc` to allocate memory for `sem_array` and semaphores. The kernel only accounts for the allocated memory when `__GFP_ACCOUNT` flag is set in the third parameter of `kvzalloc`. However, the call to `kvzalloc` at Line 8 does not specify the `__GFP_ACCOUNT` flag, leading to missing-account on the allocated semaphores and `sem_array`. Moreover, `sem_alloc` is called by the `semget` system call to allocate semaphores. As a result, this missing-account bug can be easily triggered from the user space. We further design new attacks to show that the above bug can be exploited to break the container memory confinement and exhaust all host memory.

3.2.1 *Attacking Docker.*

We set up the Docker with default capabilities and seccomp configurations. The `semget` system call does not require any capabilities and is not blocked by the container seccomp profile either. Therefore, in our experiments, the attacking program in a non-privileged container can trigger `semget` system call repeatedly to break the memory accounting limit and exhaust all host memory. The host machine even crashes due to the out-of-memory error. We reported the above bug and submitted patches to Linux community. The patches have been merged to Linux mainline and a new CVE is assigned to us.

3.2.2 *Attacking the CaaS Platform.*

We set up a self-managed OpenShift cluster on Google Cloud, running all GCP VM instances. The cluster allows users to create containers on a node with 15GB memory and Linux kernel v5.14.14. We set the memory limit for the

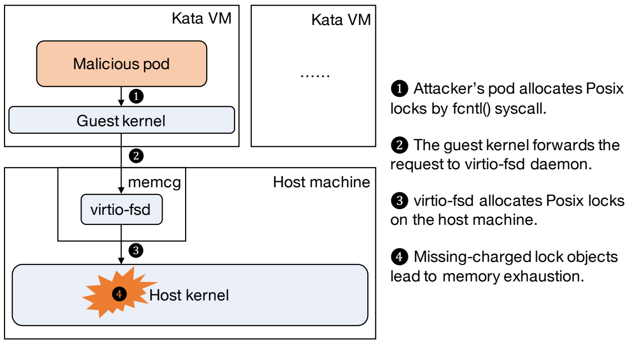
Kata VM
Malicious pod
Guest kernel
Kata VM
......
memcg
Host machine
virtio-fsd
Host kernel
❶ Attacker's pod allocates Posix locks by fcntl() syscall.
❷ The guest kernel forwards the request to virtio-fsd daemon.
❸ virtio-fsd allocates Posix locks on the host machine.
❹ Missing-charged lock objects lead to memory exhaustion.

Figure 4: Exploiting missing-account bugs to attack Kata containers.

container to 128MB, which is a popular memory size used in public cloud.

To launch the attack, the malicious user creates a container with normal user privilege and allocates a large number of semaphores inside the container. As shown in Figure 3, the memory usage of the malicious container increases rapidly and reaches the node's limit in around 20 seconds. In contrast, the charged memory usage for the attacker is extremely low, less than 1.4MB. As a result, the malicious container breaks the 128MB memory limit and consumes all 15GB memory on the host machine. The victim container on the same node cannot perform any operations due to the lack of memory.

3.2.3 Attacking the FaaS Platform. We evaluate the attack on OpenWhisk, a popular open-source FaaS platform served as the infrastructure of IBM Cloud Functions [15]. The cluster node contains 15GB memory and runs Linux kernel v5.11.0-1021-gcp. The memory limit for each function is set to 128MB.

To launch the attack, the malicious user creates a function which consume semaphores repeatedly. Next, the malicious user triggers the malicious functions repeatedly. As a result, the memory on the node is exhausted rapidly. Note that when a node's memory is exhausted, consecutive malicious functions will be dispatched to other nodes. As a result, the memory of all nodes in the cluster is exhausted by the malicious users, leading to a cluster-level DoS.

3.2.4 Attacking the Secure Runtime. We further design new attacks to evaluate memory missing-account impacts on the secure runtime—Kata container. Our experiment shows that memory missing-account bugs allow attackers to break the isolation of Kata container and attack the host machine and other container instances.

Kata container is a virtualized container runtime, in which each container instance runs in a micro-VM for strong isolation. However, Kata micro-VM forwards file operations to the host machine, which makes it vulnerable to the memory missing-account-based attack. More specifically, by exploiting memory missing-account bugs, attackers can exhaust host memory from the Kata container within the micro-VM. As shown in Figure 4, the malicious user is a normal user inside a container and issues `fcntl` syscall to allocate a huge number of POSIX locks on a set of files. The guest kernel, however, forwards the request to the virtio-fsd daemon running on the host. Therefore, the daemon allocates POSIX locks in the host

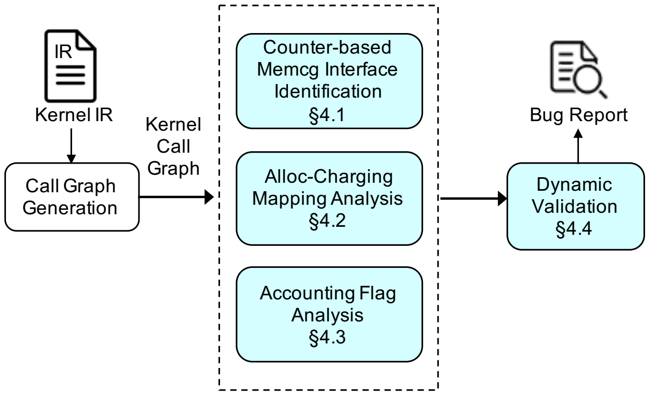
IR
Kernel IR
Call Graph Generation
Kernel Call Graph
Counter-based Memcg Interface Identification §4.1
Alloc-Charging Mapping Analysis §4.2
Accounting Flag Analysis §4.3
Bug Report
Dynamic Validation §4.4

Figure 5: Architecture of MANTA.

kernel. Although the memory usage of virtio-fsd daemon is limited by memcg, the memory used by lock objects is missing accounted. As a result, the malicious container can exhaust all physical memory on the host machine. We reported this attack to the Kata Container community. They confirmed this problem and are applying a new CVE for us.

3.3 Discussion

As demonstrated by the above attacks, the missing-account bugs can be exploited to attack both normal and secure container runtimes, Docker, and CaaS/FaaS platforms. The attacker can exploit these bugs to exhaust all memory and crash container nodes or even the whole cluster. Even worse, the attack requires only normal user privilege and thus are easy to launch. Though memory missing-account bugs can be exploited to attack container platforms, there is no systematic study to detect and eliminate the bugs. Therefore, this paper proposes the first memory accounting analysis tool named MANTA, to detect these problems systematically.

4 MISSING-ACCOUNT BUG DETECTION

Given the criticalness of memory missing-account bugs, it is important to detect these problems at an early stage before OS kernels actually run in production scenario. Unfortunately, there is no tool that can detect missing-account bugs with both high code coverage and high precision. Therefore, we propose the MANTA(short for Memory AccouNTing Analyzer) to automate the memory missing-account bug detection.

Design goals. MANTA aims to automatically and systematically detect memory missing-account bugs by analyzing the correctness of accounting interface placement. Specifically, MANTA needs to detect the memory missing-account bugs with high code coverage. To achieve this, MANTA leverages static analysis to traverse all memory allocation/free paths in Linux kernel. Moreover, MANTA needs to detect the bugs with high precision. Therefore, MANTA uses dynamic validation to test the dynamical triggerability of the detected bugs.

Challenges. To achieve both goals, MANTA needs to overcome the following challenges.

```
1  int __memcg_kmem_charge(struct mem_cgroup *memcg, gfp_t gfp, unsigned
↪   int nr_pages)
2  {
3      struct page_counter *counter;
4      ...
5      if (!cgroup_subsys_on_dfl(memory_cgrp_subsys) &&
6          !page_counter_try_charge(&memcg->kmem, nr_pages, &counter))
↪   {...}
7      ...
8  }
9
10 bool page_counter_try_charge(struct page_counter *counter,
11                     unsigned long nr_pages,
12                     struct page_counter **fail)
13 {
14     struct page_counter *c;
15     for (c = counter; c; c = c->parent) {
16         long new;
17         new = atomic_long_add_return(nr_pages, &c->usage);
18         ...
19     }
20     ...
21 }
22
23 int __memcg_kmem_charge_page(struct page *page, gfp_t gfp, int order)
24 {
25     struct mem_cgroup *memcg;
26     int ret = 0;
27     memcg = get_mem_cgroup_from_current();
28     ...
29     ret = __memcg_kmem_charge(memcg, gfp, 1 << order);
30     if (!ret) {
31         page->mem_cgroup = memcg;
32         ...
33     }
34     ...
35 }
```

Figure 6: Counter-based memcg interface identification. Use `page_counter` to detect charging/uncharging interfaces.

- **C1.** MANTA needs to identify the memory accounting interfaces, which are not described in kernel documents nor studied by previous work. Existing interface identification methods for memory allocation [20, 37] cannot be applied, as their heuristic does not work for memory accounting interfaces.
- **C2.** MANTA needs to decide for each memory allocation whether it is accounted and only accounted once. However, the execution paths from memory allocation sites to memory accounting sites tend to be deep, nested, and interleaved in Linux kernel. Moreover, even when such a path exists, the accounting can also be affected by conditional flags as the accounting requires both the accounting interface and the accounting flags §2.

Analysis techniques and workflow. The architecture of MANTA is shown in Figure 5. With the whole kernel IR as the input, MANTA first generates the kernel call graph. It then uses a page-counter-based method to identify all accounting interfaces to address C1 (§4.1). Based on accounting interfaces, MANTA builds the mapping between memory allocation/free and memory accounting (§4.2). After that, MANTA uses accounting flag analysis to further analyze the kernel memory accounting (§4.3). These two techniques address C2. With the detected memory accounting bugs, MANTA further evaluates each bug with our dynamic triggerability analysis based on thousands of test cases from Linux Test Project (§4.4). MANTA is currently implemented for Linux memory accounting. In the following, we present the details of each analysis technique.

4.1 Counter-based Interface Identification

MANTA first needs to identify the memory accounting interfaces. Such interfaces are diverse and can be custom. Existing techniques [20, 37] that use natural language processing (NLP) or wrapper function analysis would suffer from precision issues. We observe that memory accounting has to maintain memory usage *counters* (i.e., `page_counter`), which are operated through specific atomic functions. Memory usage counters can be easily located using type matching. Therefore, by identifying the primitive functions that are used for increasing/decreasing these counters, MANTA can automatically find out all functions that perform increasing/decreasing operations against accounting counters.

Specifically, MANTA walks through each kernel IR instruction and checks whether it increases/decreases `page_counter`. If so, MANTA marks the function that directly contains the instruction as a basic accounting function. Because in Linux kernel, `page_counter` is solely used by memcg, our approach can achieve high accuracy. MANTA then identifies accounting interfaces based on basic accounting functions. MANTA uses DFS to traverse backward from basic accounting functions and marks all visited functions on the kernel call graph. MANTA stops the traverse whenever it reaches a function outside the memcg subsystem because it cannot be a memcg interface. Among these marked functions that account memory usage, if a function is invoked from the outside of the memcg subsystem, i.e., source files defining memcg functionalities, MANTA marks it as an accounting interface. In this way, MANTA can identify all accounting interfaces.

Let's use the example in Figure 6 to illustrate the steps in counter-based interface identification. First, MANTA traverses all basic blocks of all kernel functions. In `__memcg_kmem_charge`, it identifies `page_counter` function (Line 6) and double confirms the first argument is from a memcg. After that, MANTA steps into the `page_counter_try_charge` and finds that it increases a field of `page_counter`. As `__memcg_kmem_charge` calls the `page_counter` function directly and increases the `page_counter` value, MANTA concludes that `__memcg_kmem_charge` is a basic charging interface. Similarly, for the interfaces that decrease the `page_counter` value, MANTA marks them as uncharging interfaces. Moreover, MANTA can distinguish interfaces for different memory types, as memcg uses different `page_counters` for each memory type (§2). At Line 6, `page_counter_try_charge` uses `&memcg->kmem` and MANTA thus knows `__memcg_kmem_charge` is for kernel memory accounting.

Starting from the basic charging interfaces, MANTA then walks up along the call chain to incrementally identify all charging interface wrappers. If a function calls the basic charging interfaces and the charging amount is from its arguments, MANTA will mark it as a charging interface wrapper. Use the same example in Figure 6, `__memcg_kmem_charge_page` calls the basic charging interface `__memcg_kmem_charge` (Line 29), while the charging amount `1 « order` is controlled by the arguments of `__memcg_kmem_charge_page`. Therefore, MANTA considers `__memcg_kmem_charge_page` as a charging wrapper because it merely passes the charging amount to the basic charging interface and has no control over the charging amount. MANTA uses the same approach to detect uncharging wrappers.

With the charging/uncharging interfaces, MANTA also needs to identify memory allocation/free interfaces. MANTA adopts existing techniques [20, 39] to produce the initial results. Moreover, on observing that memory allocation/free interfaces increase/decrease the `nr_free` counter of `struct free_area`, MANTA uses a similar counter-based approach to increase the precision of the result.

Algorithm 1: Alloc-Charging Mapping Analysis

```
   Data: $\mathcal{S} = \{s|s = (f, EscapedPages_f, ChargedPages_f)\}$
1  function MappingAnalysis($kernel$)
2      $\mathcal{S} \leftarrow$ GenSummary($kernel$)                         ❶
3      forall $f$ in $kernel$ do
4          forall $callinst$ in $f$ do
5              if IsAlloc($callinst.callee, \mathcal{S}$) then
6                  $page \leftarrow$
                     GetAllocatedPage($callinst, \mathcal{S}$)        ❷
7                  CheckPage($page, f, \mathcal{S}$)
8              end
9          end
10     end
11 end
12 function CheckPage($page, f, \mathcal{S}$)
13     if $\neg$IsCharged($page, f, \mathcal{S}$) then
14         if IsEscaped($page, f, \mathcal{S}$) then
15             // The $page$ is leaked to the callers.
16             // It is handled in the loop at Line 3-10.
17             return
18         else                                                      ❸
19             report missing-account
20         end
21     else if IsDupCharged($page, f, \mathcal{S}$) then
22         report redundant-account
23     end
24 end
```

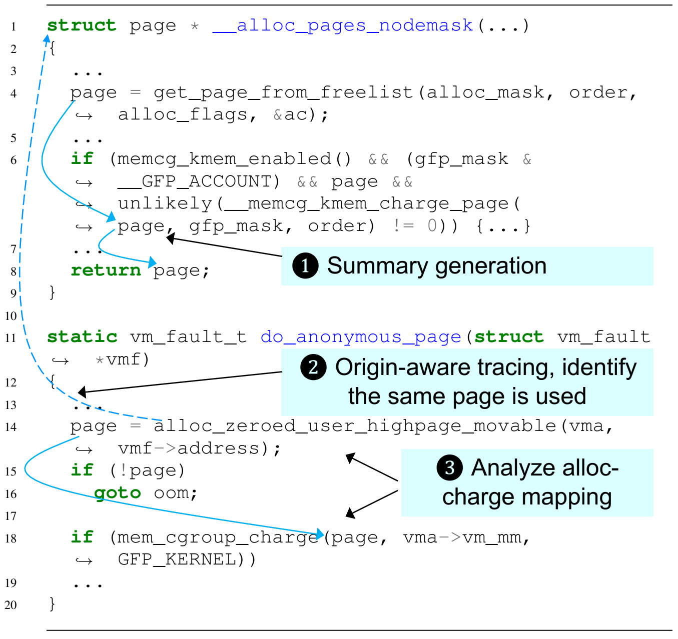
```
1  struct page * __alloc_pages_nodemask(...)
2  {
3    ...
4    page = get_page_from_freelist(alloc_mask, order,
     ↪  alloc_flags, &ac);
5    ...
6    if (memcg_kmem_enabled() && (gfp_mask &
     ↪  __GFP_ACCOUNT) && page &&
     ↪  unlikely(__memcg_kmem_charge_page(
     ↪  page, gfp_mask, order) != 0)) {...}
7    ...
8    return page;
9  }
10
11 static vm_fault_t do_anonymous_page(struct vm_fault
   ↪  *vmf)
12 {
13   ...
14   page = alloc_zeroed_user_highpage_movable(vma,
     ↪  vmf->address);
15   if (!page)
16     goto oom;
17
18   if (mem_cgroup_charge(page, vma->vm_mm,
     ↪  GFP_KERNEL))
19   ...
20 }
```
❶ Summary generation
❷ Origin-aware tracing, identify the same page is used
❸ Analyze alloc-charge mapping

Figure 7: MANTA's alloc-charging mapping analysis. The high-level algorithm of the analysis is on the left, while a concrete example is on the right.

4.2 Alloc-Charging Mapping Analysis

Intuitively, each allocated object should be accounted exactly once. Otherwise, the object can be missing-accounted. Therefore, the next step of MANTA is to build the relationship between memory allocation/free and charging/uncharging. Based on the relationship, MANTA can then identify cases that charge less or uncharge more than allocated/freed memory as missing-account bugs. For brevity, we focus on *allocation* and *charging*. The same technique can be applied to analyze free-uncharging mapping.

It is non-trivial to analyze the mapping among allocation and charging from source code because of deeply nested calling relationships and complex memory pointer propagation. To address these problems, we develop the alloc-charging mapping analysis. The basic idea is to first summarize the memory allocation and charging within a function as its function summary, and then build the memory alloc-charging mapping based on function summaries to detect missing-account bugs. More specifically, we use access-path-based analysis to generate per-function summary. The concept of access path is first used in SATURN framework [40] to express per-function summary for escaped objects (i.e., the object pointer is leaked out of the current function).

As shown by the left-side algorithm in Figure 7, MANTA first generates function summaries for all functions (Line 2) ❶. Next, for a function f, by gathering function summaries of its callees and analyzing the memory allocation sites (`IsAlloc`), MANTA can build the alloc-charging mapping for all non-escaped objects within f (Line 4-9). Specifically, MANTA substitutes the variables in callee summaries with arguments passed to the callee to calculate the allocated pages in f ❷, which is called origin-aware tracing. Then MANTA analyzes alloc-charge mapping inside f by calling `CheckPage` ❸.

In `CheckPage`, MANTA is able to report the missing-account bugs for non-escaped objects (Line 19). For the escaped objects from f (Line 14-18), they will be handled in a function (such as the caller of f) eventually as all functions in the kernel are traversed (Line 3-10). Moreover, both `IsCharged` (Line 13) and `IsDupCharged` functions (Line 21) conduct a data-flow analysis to connect the local objects to the return values or the parameters of the callee functions. In this way, they can trace a kernel object's charging state inter-procedurally using function summaries of callees.

We use the concrete example in the right side of Figure 7 to demonstrate the algorithm. MANTA first generates summary for the bottom memory allocation function `__alloc_pages_nodemask` ❶. MANTA identifies that `page` is returned by a lower function `get_page_from_freelist` (Line 4), charged (Line 6), and returned (Line 8). Therefore, `page` is both escaped and charged, and the function summary should be (`__alloc_pages_nodemask`, `retval`, `retval`). When MANTA analyzes `do_anonymous_page` at Step ❷, it fetches the function summary of `alloc_zeroed_user_highpage_movable`, which is a wrapper of `__alloc_pages_nodemask` and has the same summary. MANTA identifies that `page` in `do_anonymous_page` is already charged according to the function summary (Line 14). When analyzing alloc-charge mapping inside `do_anonymous_page` ❸, MANTA finds that the memory accounting at Line 18 is reachable from Line 14 and that `page` could be charged again at Line 18. As a result, MANTA reports a redundant-account warning in this case. On the other hand, if a page is not charged in the current function and does not escape, MANTA reports a missing-account bug.

4.3 Accounting Flag Analysis

We find that the mapping of allocation and charging sites alone is not sufficient for detecting missing-account bugs because memory

Table 1: Summary of recall evaluation. SA denotes `SLAB_ACCOUNT` flag. GKA denotes `GFP_KERNEL_ACCOUNT` flag.

No.	Affected Alloc.	Removed Flag	Detected
1	kernel/cred.c:687	SA	Y
2	kernel/cred.c:217	SA	Y
3	kernel/cred.c:258	SA	Y
4	kernel/delayacct.c:36	SA	Y
5	kernel/fork.c:170	SA	Y
6	kernel/fork.c:1508	SA	Y
7	fs/exec.c:1190	SA	Y
8	kernel/fork.c:1556	SA	Y
9	fs/file.c:293	SA	Y
10	fs/fs_struct.c:114	SA	Y
11	kernel/fork.c:1064	SA	Y
12	kernel/fork.c:1349	SA	Y
13	kernel/fork.c:348	SA	Y
14	kernel/fork.c:356	SA	Y
15	kernel/pid.c:180	SA	Y
16	kernel/utsname.c:34	SA	Y
17	security/security.c:533	GKA	Y
18	kernel/groups.c:21	GKA	Y
19	kernel/groups.c:23	GKA	Y

charging is often *conditional*. In particular, Linux kernel accounts kernel object memory only when a `__GFP_ACCOUNT` flag is passed to the accounting interfaces. Therefore if the `__GFP_ACCOUNT` flag is not provided, a memory accounting interface would still not account the memory. As a result, analyzing the values of accounting flags is critical to detecting missing-account bugs. MANTA thus conducts the accounting flag analysis to determine whether accounting flag `__GFP_ACCOUNT` is passed to accounting interfaces.

Bit-wise and inter-procedural analysis. MANTA adopts an inter-procedural bit-wise data-flow tracing approach to achieve accounting flag analysis. First, at the charging-site, MANTA checks whether the GFP flag itself is a constant. If yes, MANTA directly checks the bit of `__GFP_ACCOUNT`. Otherwise, MANTA recursively traces the use-def chain of the flag until it can confirm its accounting flag bit. Although LLVM provides a value tracking analysis pass [6] that can trace bit value propagation, the analysis is limited to intra-procedural. To track inter-procedural accounting bits, MANTA extends the pass with two analysis rules. Specifically, when MANTA traces the parameters of a function, it follows into all the caller functions to trace the values passed to the parameters; when MANTA traces the return value of a call site, it also follows into the callee functions and traces all possible returned values. If the `__GFP_ACCOUNT` is set, MANTA marks the kernel memory as charged at the allocation site. Otherwise, MANTA regards the memory as not-charged even if the charging interfaces are in place.

The analysis on allocation from `kmem_cache` is slightly different, as the allocation is also affected by the creation flag of `kmem_cache`. All allocation from a `kmem_cache` is accounted if the `kmem_cache` is created with the `SLAB_ACCOUNT` flag. Thus, for a `kmem_cache` related allocation, MANTA also finds the initialization site of the `kmem_cache` and adopts the same accounting flag analysis technique to compute the `SLAB_ACCOUNT` flag.

4.4 Dynamic Validation

Static analysis may have false positives. MANTA checks the static reachability to system calls for each missing-account allocation and filters out unreachable ones. However, statically reachable bugs are not necessarily triggerable at runtime. MANTA thus further analyzes the triggerability of statically-detected bugs through dynamic validation. Observing that Linux Test Project (LTP) [5] contains test cases for all 300 system calls, we leverage these test cases to examine the dynamical triggerability of the detected bugs. More specifically, we first instrument the code to intercept each missing-account allocation, so that we can confirm if it is actually reached. Next, we run the `syscalls` and `containers` test suites (containing 1,469 test cases) of LTP to check whether the bug is triggered.

Moreover, we manually run existing tools or develop new test cases to trigger the bugs that cannot be triggered by LTP. Some of these bugs are closely related to specific kernel features and cannot be triggered by general LTP test cases, e.g., kexec and selinux. Therefore, we manually run related user-space tools to trigger them. Others are due to specific syscall parameters uncovered by LTP and we manually develop test cases to pass these parameters.

For the missing-account bugs that can be triggered dynamically, we further develop the memory exhaustion proof-of-concept (PoC) to assess their security risks. We develop the PoCs based on test cases that trigger missing-account bugs. In particular, we repeatedly invoke syscalls that can trigger missing-account bugs in PoCs and record the amount of the missing-accounted memory. Moreover, some missing-account allocations are subjected to certain constraints, such as the ulimit and sysctl variables. Without loss of generality, we adopt the same constraints settings with cloud services like Amazon Fargate [9]. The details of these PoCs will be presented in §5.2.

4.5 Implementation

We implement MANTA as a pass of LLVM12 with 4K lines of C++ code. The analyzed Linux kernel is v5.10 with the default configuration for x86-64. MANTA uses wllvm [10] and Clang to generate IR bitcode for the whole kernel with default configurations. The bitcode size of the `vmlinux` is 481MB. MANTA uses the PeX [43, 46] approach to resolve indirect calls and generate the call graph. With the proposed techniques, MANTA is able to finish the analysis in about three minutes, making it scalable enough to be applied to every Linux major release.

5 MANTA RESULTS

In this section, we first present analysis results of MANTA. Next, we evaluate the precision and recall of MANTA. After that, we present the impact analysis and the bugs that are reported and fixed by Linux kernel community. Finally, we discuss the limitations of MANTA.

5.1 Result Overview

MANTA detects 242 missing-account bugs that are statically reachable from 273 syscalls via 60,590 different paths. Specifically, the number of userspace triggerable bugs is 162, among which 134 bugs are triggered by LTP, 23 bugs are triggered by existing tools, and 5 bugs are triggered by manually developed test cases. The remaining

Table 2: PoCs of the 47 bugs. Mis. Mem is short for missing-accounted memory. Bug IDs are consistent with Table 3.

No.	Related Syscalls	Bug IDs	Mis. Mem
1	semget, semop	30-32	>16GB
2	fcntl	20, 21	>16GB
3	io_uring_setup, io_uring_register	8-19	>16GB
4	prctl	33-35, 40, 41	>16GB
5	add_key	51-53	>16GB
6	fsopen	5-7	1167MB
7	io_uring_setup	8-13, 18, 19	835MB
8	perf_event_open	36, 37	801MB
9	timerfd_create, flock	20, 21, 28	441MB
10	timer_create	45	255MB
11	clone	23, 29, 38, 39, 43, 44, 46	231MB
12	epoll_create	3	165MB
13	bpf	33-35	116MB
14	timerfd_create	28	210MB
15	rt_sigqueueinfo	42	141MB
16	memfd_create	47, 48	192MB
17	eventfd	2	84MB

80 bugs cannot be triggered and the failure of triggering mainly results from deep call paths (72 out of 80 cases). The rest 8 allocations are either explicitly marked as exempted in kernel documentation or for kernel internal usage. Note that the 72 non-triggered bugs are not all false positives, as our test cases only cover a portion of execution paths. Conservatively speaking, the precision of MANTA is greater than 66.9% (162/242).

We also evaluate the recall of MANTA by manually removing the accounting flags. More specifically, we first generate an evaluation set by removing existing accounting flags under `kernel/` sub-directory. All affected allocation sites are collected in the evaluation set. In particular, we remove 11 `SLAB_ACCOUNT` flags and replace 5 `GFP_KERNEL_ACCOUNT` with `GFP_KERNEL` in total, affecting 19 memory allocations, as shown in Table 1. We then run MANTA on the modified kernel code and the evaluation shows that MANTA can detect all 19 missing-account sites. Although the evaluated sites are limited, it still indicates that MANTA has a high recall rate. Additionally, MANTA does not find any redundant-account bugs.

5.2 Impact Analysis

As previously mentioned, to understand the impacts of the detected bugs, we further develop proof-of-concepts (PoCs) based on LTP test cases. More specifically, we have developed 17 PoCs, which cover 47 bugs can be repeatedly triggered from the user space, as shown in Table 2. We develop and test the PoCs on a QEMU x86_64 machine with 16GB RAM. The Linux distribution is Debian Buster with Linux kernel v5.10. All PoCs run in memcg confined environments and with the default set of capabilities applied by Docker runtimes [2]. Besides the PoC covered bugs, we did not develop PoCs for the other triggerable bugs due to the limit of time and our domain knowledge. However, these bugs may also be triggered repeatedly and reliably by the attacker who has corresponding domain knowledge.

These PoCs show that all of these 47 bugs can break memory account limits. The missing-account memory in these PoCs ranges from 100MB to over 16GB, indicating that these bugs in memcg allow the attacker to use memory for free or even launch memory exhaustion attacks. More specifically, 25 bugs covered by the first five PoCs can be exploited to memory exhaustion attacks to the host system. They can consume all the 16GB memory while only being accounted for much less amount. For example, PoC-1 exploits the Bug-30, which is shown in Figure 2 and discussed in §3.2. The function `sem_alloc` allocates `sem_array` objects without accounting flags, leading to Bug-30. In addition, all `sem` objects contained in a `sem_array` are not charged either. As a result, users are not charged for any kernel semaphores. Moreover, memory allocation affected by Bug-31 and 32 are used to maintain `sem_undo` objects, so a user could also create such objects for free.

We use PoC-1 as an example to demonstrate that missing-account bugs can easily cause a system-wide DoS, which continues even after the attacking process is terminated. The PoC repeatedly calls `semget` to allocate 30,000 semaphore sets, with 30,000 semaphore instances in each set. For each set, PoC-1 further calls `semop` to create `sem_undo` objects for it. As a result, even though each `sem` object takes a small amount of memory, all of these objects consume much more than 16GB of memory and cause DoS. Even worse, the memory is not freed after the PoC program is terminated because `sem` objects continue to exist, leading to the persistent DoS attack. The above PoC is practical under the default system configurations. The semaphore number is mainly limited by the sysctl variable `kernel.sem`, which allows 32,000 semaphore sets and 32,000 objects in each set by default. We use 30,000 for PoC-1. In sum, the attacker can easily turn a missing-account bug into a DoS attack.

5.3 Reporting to Linux Community

We have reported 53 bugs to the Linux kernel community, including all 47 bugs that can be triggered by 17 user space PoCs and 6 bugs that can be repeatedly triggered by kernel operations. The full list of these bugs is listed in Table 3 of appendix A.

Among all reported bugs, 37 of them have been confirmed by kernel developers. The patches for all of these 37 bugs are submitted, with 18 are already merged and 19 are pending to be merged. The patching process is slow because memcg subsystem does not maintain its git sub-tree currently. Thus memcg maintainers do not track submitted patches and expect that the patch authors will push the patches upstream via other subsystem maintainers. This introduces lots of extra discussions and reviews that slow down the merge process significantly.

When communicating with the kernel developers, we find that the kernel community are highly concerned about these bugs. One kernel developer stressed that memcg is "not just rough accounting estimation" and "unbound allocation which can be triggered by userspace should be accounted". Besides, kernel developers are very interested in MANTA and invited us to test their kernels using our tools and experiments.

5.4 Limitations

MANTA currently has the following limitations.

Static analysis is not sound. MANTA's analysis can miss missing-account bugs due to path-insensitive function summaries. A function may not charge an allocated page in one of execution paths while it charges the page in another path. In this case, MANTA still summarizes the function as charged and misses missing-account bugs in later analysis. On the contrary, if MANTA summarizes the function as not-charged, it misses redundant-account bugs. In the future, we plan to adopt path-sensitive summaries like SATURN framework [40] to improve the soundness.

Dynamic triggerable test is not complete. Currently, MANTA uses LTP test cases to test whether a statically-detect bug can be triggered or not. The LTP test cases are limited. It is impossible for these test cases to cover all possible execution paths in the kernel. As a result, MANTA could miss some triggerable bugs due to the incomplete code coverage. Therefore, one of our future work is to develop more test cases and adopt fuzzing techniques to increase the code coverage.

6 RELATED WORK

In this section, we discuss existing works that are related to memory accounting and resource accounting.

6.1 Memory Accounting

JRes [16] implements memory accounting interfaces for JVM. The memory accounting interfaces charge memory usage during object allocation and uncharge during garbage collection. However, JRes does not implement a recharging mechanism for shared objects. Price et al. [33] and MzScheme [38] propose consumer-based memory accounting schemes for user space run-time environment to account subtasks' memory usage. They choose to charge memory usage to its actual users rather than to the memory allocators. They modify the runtime garbage collector so that it can uncharge an object before it is freed and recharge it to the entity that holds its references. However, they are hard to be directly adopted in kernel as implementing efficient enough garbage collecting in kernel remains a challenge. VM-based work provides better memory isolation and could resolve the per-process accounting problems in monolithic kernels. However, traditional VM-based schemes [1, 13, 25] are too heavy-weight and introduce high performance overhead. Unikernel-based methods [26, 29, 32, 44] achieve lower startup latency and higher through-puts but lack compatibility compared with container-based schemes [19, 35].

MANTA's alloc-charging mapping analysis is similar to alloc-free mapping analysis used in memory leak detection. Leak Checker [40] uses a context- and path-sensitive analysis method to detect memory leaks. Saber [36] and FaskCheck [14] use on-demand sparse value-flow analysis to detect memory leaks. LeakFix [21] first tries to locate memory free that leads to leaks and fix it. AutoFix [41] detects and fixes memory leaks by combining static analysis and runtime checks. PCA [27] uses selective flow-sensitive algorithm to further speeds up inter-procedural data-flow analysis for memory leak detection. However, these work relies on either manual input or specific heuristic to decide allocation/free interface, which cannot be applied to identify memory accounting interfaces.

6.2 Resource Accounting

Houdini's escape [22] leverages kernel bottom-half worker threads, service processes, and interrupts to bypass confinement of control groups. However, it didn't focus on memcg and didn't analyze the implementation of control groups either. Compared with Houdini's escape, our paper systematically defines possible problems in memcg and identify and report 53 bugs in memcg. Yang et al. proposes the concept of abstract resources and finds that exhausting these resources within containers lead to DoS attacks on host machines [42]. Resource container [12] proposes an OS-level abstraction to account for whole-system resource usage. It is the first work that describes the method to account general resources at the per-thread level. Compared to previous work, resource container works at system kernel and can account for resource consumption in kernel caused by user threads. Zhang et al. [45] proposed to charge CPU-time usage of kernel bottom-half processes to affected user processes to improve fairness.

Perez et al. [31] observes that EVM's *gas* metering is not consistent with actual resource consumption, especially CPU cycles, on working nodes. It shows the actual inconsistency further increases when page cache works poorly, while the payer is charged for the same amount of *gas*. The paper also proposes a method to automatically synthesize payloads that can fully exploit the inconsistency. Liu et at. [28] finds the CPU time metering vulnerable on various real-world cloud platforms. They propose 6 attacks that could exploit CPU metering at process launch time or run time.

7 CONCLUSION AND FUTURE WORK

In this paper, we conduct the first systematic analysis and detection against the Linux memory missing-account bugs. We perform an in-depth analysis to understand the exploitability and security impacts of the memory missing-account bugs on container platforms. Our analysis shows that all container runtimes, including both normal and secure runtimes, are vulnerable to memory exhaustion attacks resulting from missing-account bugs. Moreover, memory missing-account bugs can be exploited to attack the Docker, the CaaS, and the FaaS platforms, leading to the memory exhaustion, which crashes the node or even the whole cluster.

We then propose MANTA, which combines both static and dynamic analysis techniques to automatically detect memory missing-account bugs with both high code coverage and precision. Our detection tool reports 53 exploitable memory missing-account bugs, 37 of which were confirmed by kernel developers, two new CVEs are assigned. The result shows that MANTA effectively helps to mitigate missing-account problems in memory accounting systems. Our future work is to study how to account memory usage accurately without mediating all memory allocation/free paths.

ACKNOWLEDGMENTS

The authors would like to thank all reviewers for insightful comments. This work is partially supported by the National Natural Science Foundation of China (Grants No. 62002317), by the National Key R&D Program of China (Grant No. 2020AAA0107700), by the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (Grant No. 2018R01005), and by the Ant Group Funds for Security Research.

REFERENCES

[1] 2021. VMware ESXi: The Purpose-Built Bare Metal Hypervisor. https://www.vmware.com/products/esxi-and-esx.html.

[2] 2022. Docker run reference. https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities.

[3] 2022. Host cgroup management. https://github.com/kata-containers/kata-containers/blob/main/docs/design/host-cgroups.md.

[4] 2022. Linux Container Configuration. https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md#control-groups.

[5] 2022. Linux Test Project. https://linux-test-project.github.io/.

[6] 2022. LLVM Value Tracking Analysis. https://llvm.org/doxygen/ValueTracking_8cpp.html.

[7] 2022. The new cgroup slab memory controller. https://lwn.net/Articles/824216/.

[8] 2022. New Linux Memory Controller. https://thenewstack.io/a-new-linux-memory-controller-promises-to-save-lots-of-ram/.

[9] 2022. What is AWS Fargate? https://docs.aws.amazon.com/AmazonECS/latest/userguide/what-is-fargate.html.

[10] 2022. Whole Program LLVM. https://github.com/travitch/whole-program-llvm.

[11] AWS. 2022. AWS Management Console. https://aws.amazon.com/console/.

[12] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *OSDI*, Vol. 99. 45–58.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

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

[15] IBM Cloud. 2022. IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[16] Grzegorz Czajkowski and Thorsten Von Eicken. 1998. JRes: A resource accounting interface for Java. *ACM SIGPLAN Notices* 33, 10 (1998), 21–35.

[17] Docker. 2022. Runtime metrics. https://docs.docker.com/config/containers/runmetrics/.

[18] Docker. 2022. Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/.

[19] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[20] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning. In *In Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS'21)*.

[21] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for c programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 459–470.

[22] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 2019. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1073–1086.

[23] Red Hat. 2022. Red Hat OpenShift. https://www.redhat.com/en/technologies/cloud-computing/openshift.

[24] katacontainers. 2022. Kata Containers: The speed of containers, the security of VMs. https://katacontainers.io/.

[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. Dttawa, Dntorio, Canada, 225–230.

[26] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 61–72.

[27] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: Memory Leak Detection Using Partial Call-Path Analysis *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA.

[28] Mei Liu and Xuhua Ding. 2010. On trustworthiness of cpu usage metering and accounting. In *2010 IEEE 30th International Conference on Distributed Computing Systems Workshops*. IEEE, 82–91.

[29] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 218–233.

[30] Apache OpenWhisk. 2022. Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[31] Daniel Perez and Benjamin Livshits. 2020. Broken metre: Attacking resource metering in EVM. In *Network and Distributed Systems Security (NDSS) Symposium 2020*.

[32] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. 2011. Rethinking the library OS from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. 291–304.

[33] David W Price, Algis Rudys, and Dan S Wallach. 2003. Garbage collector memory accounting in language-based systems. In *2003 Symposium on Security and Privacy, 2003*. IEEE, 263–274.

[34] Rami Rosen. 2016. Namespaces and Cgroups – the basis of Linux Containers. https://netdevconf.info/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf.

[35] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 121–135.

[36] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 254–264.

[37] Jianqiang Wang, Siqi Ma, Yuanyuan Zhang, Juanru Li, Zheyu Ma, Long Mai, Tiancheng Chen, and Dawu Gu. 2019. 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)*. 309–321.

[38] Adam Wick and Matthew Flatt. 2004. Memory accounting without partitions. In *Proceedings of the 4th international symposium on Memory management*. 120–130.

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

[40] Yichen Xie and Alex Aiken. 2005. Context-and path-sensitive memory leak detection. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 115–125.

[41] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2016. Automated memory leak fixing on value-flow slices for c programs. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 1386–1393.

[42] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, et al. 2021. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 764–778.

[43] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. Pex: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*. 1205–1220.

[44] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: a dynamic library operating system for simplified and efficient cloud virtualization. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*. 173–186.

[45] Yuting Zhang and Richard West. 2006. Process-aware interrupt scheduling and accounting. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 191–201.

[46] Jinmeng Zhou, Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed Azab, Ruowen Wang, Peng Ning, and Kui Ren. 2022. Automatic Permission Check Analysis for Linux Kernel. *IEEE Transactions on Dependable and Secure Computing* (2022).

A REPORTED BUGS

As shown in Table 3, we have reported 53 missing-account bugs to Linux community. 37 of them are confirmed by Linux community and the corresponding patches are submitted. The *source location* column indicates where the bugs reside in Linux kernel and *function name* indicates the functions that contain the bugs. *Allocation interface* indicates the missing-accounted memory allocation function. *Triggered by* indicates the syscalls that can trigger the missing-account bugs.

Table 3: Summary of 53 exploitable missing-account bugs. 37 bugs are confirmed by Linux kernel developers and the corresponding patches are submitted. Out of these patches, 18 are merged and 19 are pending for merging. While the other 16 bugs are pending to be confirmed.

Bug ID	Source location	Function name	Allocation interface	Triggered by	Confirm Status	Patch Status
1	arch/x86/kernel/ldt.c:157	alloc_ldt_struct	kmalloc	clone	Confirmed	Merged
2	fs/eventfd.c:417	do_eventfd	kmalloc	eventfd	Pending	-
3	fs/eventpoll.c:1014	ep_alloc	kzalloc	epoll_create	Pending	-
4	fs/fcntl.c:899	fasync_alloc	kmem_cache_alloc	fcntl	Confirmed	Merged
5	fs/fs_context.c:234	alloc_fs_context	kzalloc	fsopen	Confirmed	Merged
6	fs/fs_context.c:634	legacy_init_fs_context	kzalloc	fsopen	Confirmed	Merged
7	fs/fsopen.c:100	fscontext_alloc_log	kzalloc	fsopen	Pending	-
8	fs/io-wq.c:1089	io_wq_create	kzalloc	io_uring_setup	Confirmed	Pending
9	fs/io-wq.c:1093	io_wq_create	kzalloc_node	io_uring_setup	Confirmed	Pending
10	fs/io-wq.c:1114	io_wq_create	kzalloc_node	io_uring_setup	Confirmed	Pending
11	fs/io_uring.c:1180	io_ring_ctx_alloc	kzalloc	io_uring_setup	Confirmed	Pending
12	fs/io_uring.c:1184	io_ring_ctx_alloc	kmem_cache_alloc	io_uring_setup	Confirmed	Pending
13	fs/io_uring.c:1197	io_ring_ctx_alloc	kmalloc	io_uring_setup	Confirmed	Pending
14	fs/io_uring.c:7254	__io_sqe_files_scm	kzalloc	io_uring_register	Pending	-
15	fs/io_uring.c:7507	alloc_fixed_file_ref_node	kzalloc	io_uring_register	Pending	-
16	fs/io_uring.c:7546	io_sqe_files_register	kzalloc	io_uring_register	Pending	-
17	fs/io_uring.c:7853	io_uring_alloc_task_context	kmalloc	io_uring_setup	Confirmed	Pending
18	fs/io_uring.c:8044	io_mem_alloc	__get_free_pages	io_uring_setup	Confirmed	Pending
19	fs/io_uring.c:9541	io_register_personality	kmalloc	io_uring_register	Pending	-
20	fs/locks.c:346	locks_alloc_lock	kmem_cache_zalloc	flock, fcntl	Confirmed	Merged
21	fs/locks.c:258	locks_get_lock_context	kmem_cache_alloc	flock	Confirmed	Merged
22	fs/namespace.c:177	alloc_vfsmnt	kmem_cache_zalloc	mount	Confirmed	Merged
23	fs/namespace.c:3267	alloc_mnt_ns	kzalloc	clone	Confirmed	Pending
24	fs/notify/group.c:121	fsnotify_alloc_group	kzalloc	inotify_init1	Confirmed	Merged
25	fs/notify/inotify/inotify_user.c:630	inotify_new_group	kmalloc	inotify_init1	Confirmed	Merged
26	fs/select.c:658	core_sys_select	kvmalloc	select	Confirmed	Merged
27	fs/signalfd.c:278	do_signalfd4	kmalloc	signalfd4	Pending	-
28	fs/timerfd.c:412	timerfd_create	kzalloc	timerfd_create	Pending	-
29	ipc/namespace.c:45	create_ipc_ns	kzalloc	clone	Confirmed	Pending
30	ipc/sem.c:514	sem_alloc	kvzalloc	semget	Confirmed	Merged
31	ipc/sem.c:1853	get_undo_list	kzalloc	semop	Confirmed	Merged
32	ipc/sem.c:1938	find_alloc_undo	kzalloc	semop	Confirmed	Merged
33	kernel/bpf/core.c:117	bpf_prog_alloc	alloc_percpu_gfp	bpf, prctl	Confirmed	Merged
34	kernel/bpf/core.c:85	bpf_prog_alloc_no_stats	__vmalloc	bpf, prctl	Confirmed	Merged
35	kernel/bpf/core.c:89	bpf_prog_alloc_no_stats	kzalloc	bpf, prctl	Confirmed	Merged
36	kernel/events/core.c:11142	perf_event_alloc	kzalloc	perf_event_open	Pending	-
37	kernel/events/core.c:4443	alloc_perf_context	kzalloc	perf	Pending	-
38	kernel/nsproxy.c:56	create_nsproxy	kmem_cache_alloc	setns	Confirmed	Pending
39	kernel/pid_namespace.c:90	create_pid_namespace	kmem_cache_zalloc	clone	Confirmed	Pending
40	kernel/seccomp.c:1481	init_listener	kzalloc	prctl	Pending	-
41	kernel/seccomp.c:565	seccomp_prepare_filter	kzalloc	prctl	Pending	-
42	kernel/signal.c:435	__sigqueue_alloc	kmem_cache_alloc	rt_sigqueueinfo	Confirmed	Merged
43	kernel/time/namespace.c:91	clone_time_ns	kmalloc	clone	Confirmed	Pending
44	kernel/time/namespace.c:97	clone_time_ns	alloc_page	clone	Confirmed	Pending
45	kernel/time/posix-timers.c:458	alloc_posix_timer	kmem_cache_zalloc	timer_create	Confirmed	Merged
46	kernel/user_namespace.c:105	create_user_ns	kmem_cache_zalloc	clone	Confirmed	Pending
47	mm/hugetlb.c:868	resv_map_alloc	kmalloc	memfd_create	Pending	-
48	mm/hugetlb.c:869	resv_map_alloc	kmalloc	memfd_create	Pending	-
49	mm/slab_common.c:245	create_cache	kmem_cache_zalloc	clone	Confirmed	Pending
50	net/core/net_namespace.c:423	net_alloc	kzalloc	clone	Pending	-
51	security/keys/key.c:277	key_alloc	kmem_cache_alloc	add_key	Confirmed	Pending
52	security/keys/key.c:282	key_alloc	kmemdup	add_key	Confirmed	Pending
53	security/keys/key.c:81	key_user_lookup	kzalloc	add_key	Confirmed	Pending