BoKASAN: Binary-only Kernel Address Sanitizer for Effective Kernel Fuzzing

Mingi Cho
Yonsei University

Dohyeon An
Yonsei University

Hoyong Jin
Yonsei University

Taekyoung Kwon
Yonsei University

Abstract

Kernel Address Sanitizer (KASAN), an invaluable tool for finding use-after-free and out-of-bounds bugs in the Linux kernel, needs the kernel source for compile-time instrumentation. To apply KASAN to closed-source systems, we should develop a binary-only KASAN, which is challenging. A technique that uses binary rewriting and processor support to run KASAN for binary modules needs a KASAN-applied kernel, thereby still the kernel source. Dynamic instrumentation offers an alternative way to it but greatly increases the performance overhead, rendering the kernel fuzzing impractical.

To address these problems, we present the first practical, binary-only KASAN named *BoKASAN*, which conducts address sanitization through dynamic instrumentation for the entire kernel binaries efficiently. Our key idea is *selective sanitization*, which identifies target processes to sanitize and hooks the page fault mechanism for significantly reducing the performance overhead of dynamic instrumentation. Our key insight is that the kernel bugs are most relevant to the processes created by a fuzzer. Thus, BoKASAN deliberately sanitizes the target memory regions related to these processes and leaves the remains unsanitized for effective kernel fuzzing.

Our evaluation results show that BoKASAN is practical on closed-source systems, achieving the compiler-level performance of KASAN even on binary-only kernels and modules. Compared to KASAN on the Linux kernel, BoKASAN detected slightly more bugs in the Janus dataset and slightly fewer bugs in the Syzkaller/SyzVegas dataset; and BoKASAN found the same number of unique bugs in the 5-day fuzzing and executed the similar number of basic blocks. For binary modules on the Windows kernel and the Linux kernel, resp., BoKASAN was effective in finding bugs. An ablation result shows that selective sanitization affected these outcomes.

1 Introduction

Modern operating systems are susceptible to use-after-free and out-of-bounds bugs that are hardly detectable by state-of-the-art fuzzers [51]. Kernel Address Sanitizer (KASAN) is a dynamic analysis tool specifically designed to find these bugs in the Linux kernel. KASAN uses compile-time instrumentation to insert code for calling hook functions before each memory access and validates the accessed memory bytes at run time. According to Syzbot [56], the coverage-guided kernel fuzzer Syzkaller detected a great number of Linux kernel bugs by incorporating KASAN, which indeed contributed to finding at least 27% of the bugs. However, this invaluable tool cannot be applied to closed-source systems. Not all kernels and modules are provided as open source, and even in the case of open source, the corresponding version is not always available. This crucial situation highlighted the need (§2.1) for developing binary-only KASAN [9, 34].

The development of binary-only KASAN is challenging though. It would be requiring dynamic or static binary instrumentation in the kernel mode as implementing binary-only ASAN for user-mode applications [8, 10, 12]. In practical senses, unfortunately, it is notorious to apply binary instrumentation to kernels because of the performance overhead owing to the complexity and large size of the kernel binaries (§2.2). For instance, Bochspwn Reloaded [21] used dynamic taint analysis with the Bochs emulator to find kernel memory disclosure bugs; but, the 13–18 times increase in the performance overhead was inevitable because of needing to decode all the instructions for emulation, which made it impractical for fuzzing. KRetroWrite [47] applied static binary instrumentation to support KASAN on the binary kernel modules. However, the binary rewriting of the modules requires the main kernel to which KASAN has already been applied, meaning that the correct kernel source is still necessary.

In this paper, we address these problems by introducing a novel hooking-based approach called *selective sanitization*. The basic insight behind this idea is that most of the bugs detected by fuzzing are actually discovered in the processes created by the fuzzer (§2.3). Thus, we selectively sanitize the memory region allocated by the fuzzer-created kernel processes and leave the remaining area unsanitized for execution (§2.4). This targeted sanitization method (§3.1) can significantly reduce the performance overhead of dynamic

instrumentation. We develop the first "practical" binary-only KASAN named *BoKASAN* using this idea.

BoKASAN dynamically performs 1) function instrumentation (§3.2) and 2) memory access instrumentation (§3.3), respectively. On function instrumentation, BoKASAN creates a red zone by hooking the memory allocation functions and then initializes the shadow memory. On memory access instrumentation, BoKASAN exploits the page fault mechanism of the OS. When the sanitized memory region is accessed, BoKASAN forcibly produces a page fault and checks the shadow memory. If the red zone was accessed, then BoKASAN alarms this to the fuzzer.

We implement BoKASAN on the Linux kernel (§4) and further on the Widows kernel (§A). We evaluate BoKASAN regarding its bug-finding capability (§5.1) and overhead reduction (§5.2), compared to KASAN on the Linux kernel. We then evaluate the effectiveness of selective sanitization in BoKASAN compared to its ablated version (§5.3). We integrate BoKASAN with Syzkaller for fuzzing and use Janus [64] and Syz (Syzkaller and SyzVegas [61]) datasets, respectively, and show that, unlike previous methods, BoKASAN practically uses dynamic instrumentation for address sanitization of the entire kernel binaries. Moreover, we evaluate BoKASAN with regard to binary-only fuzzing on the Windows kernel compared to the Linux kernel (§5.4). We integrate BoKASAN with kAFL for fuzzing and use the kAFL's buggy driver in our evaluation. We account for the random nature of fuzzing as guided in [25].

To sum up, this paper makes the following contributions:

- We present a novel method called a *selective sanitization* blended with a page-fault handling mechanism to significantly reduce the performance overhead in dynamic instrumentation for binary-only kernel address sanitization.
- We design and implement BoKASAN that realizes our new idea. To the best of our knowledge, BoKASAN is the first binary-only kernel address sanitizer practical for the "entire" kernel binaries, achieving compiler-level performance.
- We evaluate BoKASAN (with kernel binaries only) in comparison to KASAN (with kernel source code) on the Linux kernel and produce the following results.
  - BoKASAN detected slightly more bugs (21>20) in the Janus dataset and slightly fewer bugs (14<15) in the Syz dataset (§5.1); and the equal number of unique bugs (21) and more bugs (178>139) in 5d fuzzing.
  - BoKASAN executed almost the same number of basic blocks and 10.5% fewer syscalls in 24h fuzzing (§5.2). BoKASAN executed 5.1 times more syscalls than KASAN running with full emulation.
  - Selective sanitization affected the performance (§5.3), e.g., 4.3 times more syscalls than its ablated version.
- We further implement BoKASAN for binary-only fuzzing on the Windows kernel and confirm that BoKASAN can be applied to closed-source systems including COTS OSs.

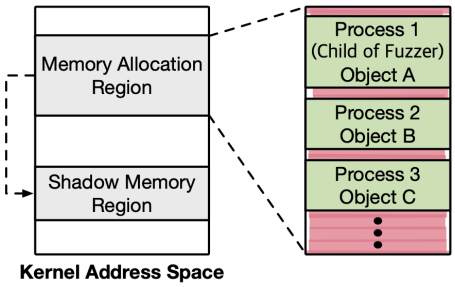


Figure 1: Overview of KASAN. Multiple processes share a single memory region and KASAN sanitizes all processes.

2 Background and Motivation

In this section, we describe the necessity for binary-only KASAN and explain why it is difficult to implement. We then present our approach and consider its feasibility.

2.1 Binary-only KASAN Is Necessary

Kernel address sanitizer (KASAN) detects dynamic memory errors, such as out-of-bounds (OOB) and use-after-free (UAF) bugs, in a kernel that employs an address space that differs from that used for user-mode applications. KASAN uses shadow memory-based red zones, similar to ASAN, to effectively check memory errors, as illustrated in Figure 1. For this purpose, it performs compile-time instrumentation in three phases of memory management: allocation, use, and free (deallocation). Memory allocation functions (e.g., *kmalloc*, *kfree*) are instrumented on memory allocation and free phases, and memory accesses are instrumented on memory use. The instrumentation operates as follows: First, the red zones are added around the allocated memory, and the shadow memory corresponding to this memory address is initialized. The shadow memory sets the allocated memory as accessible, but the red zones are inaccessible. When any process accesses the memory, the shadow memory is checked to determine whether the accessed memory address is valid (i.e., it is not located in the red zones). When the memory is freed after use, the shadow memory is updated to designate the corresponding memory address an inaccessible red zone. Any access to the red zones is reported as an error.

KASAN has been used with state-of-the-art kernel fuzzers, such as Syzkaller, to discover a large number of Linux kernel bugs. According to a report by Syzbot [60], among 3,173 Linux kernel bugs discovered by Syzkaller [59] and patched by April 2022, 933 (26%) bugs were detected by KASAN. Among the bugs found by KASAN, UAF and slab OOB access were the most common at 535 (57%) and 203 (21%), respectively. Based thereupon, we can conclude that KASAN is essential for discovering kernel memory errors, especially UAF and slab OOB access. However, existing technologies for KASAN commonly require a kernel source, for exam-

Table 1: LMBench Result. Time is in microseconds. Baseline and KASAN use KVM, and KASAN (FE) uses full emulation without using KVM. Numbers in parentheses indicate additional overhead compared to baseline.

Syscall	Baseline	KASAN (%)		KASAN (FE) (%)	
null	0.228	0.243	(6.9)	1.737	(633.7)
open/close	2.184	7.540	(245.2)	188.571	(8,534.2)
read	0.470	0.529	(12.5)	7.550	(1,506.8)
write	0.338	0.365	(8.1)	4.213	(1,147.8)
stat	1.312	4.297	(227.5)	82.920	(6,220.6)
fstat	0.361	0.457	(26.5)	4.141	(1,047.8)

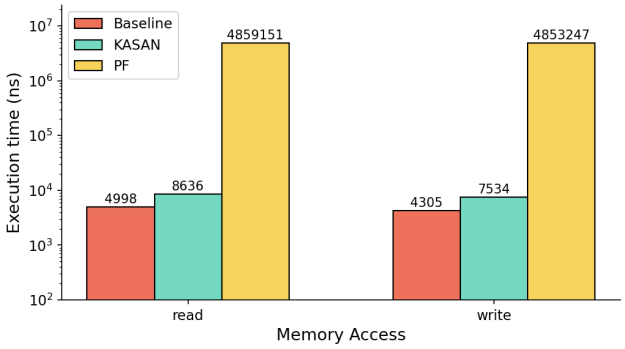


Figure 2: Time to access memory for 1,000 times. We used an average time of 30 trials. PF is 972 and 563 times slower than the baseline and KASAN for read, and 1,127 and 644 times slower for write, respectively.

ple, for compile-time instrumentation. Consequently, kernel fuzzers can only find bugs in COTS OSs when a kernel panic occurs. For example, Bochspwn Reloaded discovered more than 70 memory disclosure bugs in the Windows kernel, but existing kernel fuzzers cannot detect them. Note that both Bochspwn Reloaded and kernel fuzzers cannot detect UAF and slab OOB access bugs. Therefore, to discover memory error bugs more effectively in COTS OSs, KASAN would have to be applied to kernel binaries without requiring any kernel source.

2.2 Implementing Binary-only KASAN Is Challenging

To apply KASAN to kernel binaries only, we need a binary instrumentation method against target kernels: *static instrumentation* for directly modifying the binary or *dynamic instrumentation* for inserting code at runtime. For example, KRetroWrite [47] statically rewrites the binary kernel module to apply KASAN, whereas Bochspwn Reloaded performs dynamic taint tracking on the Bochs emulator to discover kernel memory disclosure bugs. However, our observation is that it would be highly challenging to use both of these approaches for implementing binary-only KASAN.

2.2.1 Static Instrumentation

For kernel binaries more complicated than user programs, static instrumentation is challenging, and in particular, sound instrumentation is infeasible [45], at least because of the following. If one part of the kernel does not work correctly, the operation of entire system may fail. To the best of our knowledge, a static instrumentation method does not exist for the entire kernel binary, but only for the binary kernel module. KRetroWrite [47] used static instrumentation to apply binary KASAN to a binary kernel module. However, KRetroWrite can only rewrite the binary kernel modules and directly load the modules to the kernel that KASAN has already employed, which means that it still requires the source code of the kernel. This technique cannot support binary-only kernels. Note that not all kernels are provided as open source, and even if the kernel were open source, the correct version is not always available [9]. In addition, the distinct binary format used by each vendor of COTS OSs makes static instrumentation more difficult [2]. For example, at Apple, much of the kernel is implemented in C++. This language has functions such as classes and exception handlers, and static analysis of C++ binaries is known to be more difficult than that of C binaries because much of the information necessary for these functions disappears during the compilation [42, 50]. RetroWrite fails in binary rewriting if a C++ exception handler exists.

2.2.2 Dynamic Instrumentation

Compared to static instrumentation, dynamic instrumentation causes higher instrumentation overhead although it is relatively easy to implement and effective in finding actual kernel bugs (e.g., coverage measurement) [35, 39]. Dynamic instrumentation methods for implementing binary-only KASAN could involve either 1) using an emulator or 2) utilizing the page fault mechanism of the kernel. We analyzed the overhead caused by these two methods to see why they are impractical.

Emulation is slow. According to the Bochspwn project, the Bochs [26] emulator takes approximately 13–18 times longer to boot the operating system compared to a virtual machine running on VirtualBox [21]. This means that at least 13 times the overhead is generated only for kernel emulation, which means that additional overhead is required for memory access instrumentation. When taint tracking was applied to discover memory disclosure in Bochspwn Reloaded, the total overhead, including emulation, increased by approximately 32 to 50 times. Additionally, the overhead was measured using LMbench [38] as a benchmark in Linux running on the QEMU emulator. Table 1 presents the results of running LMbench in Linux kernel v4.15 in microseconds. Baseline and KASAN are the results of using QEMU with KVM enabled, and KASAN (FE) results from full emulation without using

KVM. For memory access instrumentation, full emulation is required without KVM because it is trapped when a memory access instruction is executed on the emulator. In Table 1, when KASAN-applied Linux is operated with full emulation, an overhead of approximately 7–85 times is generated compared to the baseline. As a result, using the emulator introduces a very large amount of overhead of approximately 10 times or more compared to KASAN, which incurs an overhead of approximately three times more.

Hooking a page fault handler generates extremely high overhead. To analyze the overhead resulting from using the second method, the page fault mechanism, we hooked the default page fault handler of the kernel to instrument memory access. We then measured the memory read/write overhead. The memory access instrumentation is performed in the same way as Periscope [52]; however, unlike Periscope, which modifies the source code, we dynamically hooked the necessary functions (*kmalloc*, page fault handler, debug exception handler) using *ftrace*. Figure 2 shows the result of measuring the overhead in memory access of the baseline, KASAN, and page fault-based instrument (PF). The baseline is a kernel in which KASAN is not applied. Memory read/write was performed 1,000 times after allocating memory in the kernel space with the *kmalloc* function to measure the overhead. These measurements indicated that the PF had approximately 972–1,127 times and 563–644 times overhead compared to the baseline and KASAN, respectively. Because context switching is performed when an interrupt is triggered, the context of the current process is saved at the interrupt handler entry and restored at the exit. A large amount of overhead is incurred because all of the above processes must be performed for each memory access. The overhead is incurred only in the memory access instructions, meaning that the total overhead during the actual system call would be less; however, the amount is obviously large compared to the compile-time instrumentation, which generates approximately 2–3 times more overhead. The considerable overhead generated by such a dynamic instrumentation method makes kernel fuzzing impractical.

2.3 Interesting Difference in Kernel Fuzzing

Although binary-only KASAN is clearly required, the above-mentioned challenging problems have prevented a practical solution from being found. Before we attempt to solve this problem, we observe an interesting fact: the difference between the user space and kernel space when fuzzing. The intriguing difference between these two spaces is that in the user space, one process is generally tested, whereas in the kernel space, a specific process is tested in an environment where multiple processes are running. Because the fuzzer tests the entire code of the target program in the user space, applying the sanitizer to all the code of the user program is very reasonable. However, the kernel space is also used to run processes that are basic functions of the OS, and these

Table 2: Results of fuzzing with Syzkaller for 7 days with 60 VMs. 20 out of 22 bugs (91%) found by KASAN and 50 out of 57 total bugs (88%) were caused by the fuzzer process.

ID	Bug Description	Fuzzer Process
1	UAF Read in n_tty_receive_buf_common	✓
2	UAF Read in do_update_region	✗
3	UAF Read in vgacon_invert_region	✓
4	UAF Read in complement_pos	✓
5	UAF Read in disk_unblock_events	✓
6	UAF Read in screen_glyph	✓
7	Slab OOB Read in xattr_getsecurity	✓
8	Slab OOB Read in simple_xattr_alloc	✓
9	UAF Write in __ext4_expand_extra_isize	✓
10	UAF Read in vgacon_scroll	✓
11	Slab OOB Read in ext4_xattr_set_entry	✓
12	UAF Read in vc_do_resize	✓
13	Stack OOB Read in xfrm_state_find	✓
14	UAF Write in do_con_write	✓
15	Slab OOB Write in perf_callchain_user	✗
16	UAF Read in vcs_write	✓
17	UAF Read in iptunnel_handle_offloads	✓
18	UAF Read in copyout	✓
19	UAF Write in con_shutdown	✓
20	UAF Write in vgacon_scroll	✓
21	UAF Read in build_segment_manager	✓
22	Slab OOB Read in do_con_trol	✓
Total		20/22

are generally not related to the process of being fuzzed (i.e., it does not receive input from the user). Therefore, applying a sanitizer to processes other than the fuzzing target process that receives user input and detects bugs can incur unnecessary overhead. For example, in the kernel space, as shown in Figure 1, several processes allocate objects in one memory area, but a large number of allocated objects are not related to fuzzing.

Interestingly, most bugs found by kernel fuzzers are triggered by processes created by the fuzzer (i.e., processes that call random syscalls). Table 2 lists the memory corruption bugs detected by KASAN as a result of fuzzing Linux kernel v4.15. We performed 7d of fuzzing using 60 virtual machines. As a result of the experiment, a total of 57 bugs, of which KASAN detected 22 bugs, were found. The 22 OOB access and UAF type bugs detected by KASAN are listed in the second column of Table 2. The third column is the result of comparing process identifiers (PIDs) to check whether the crashed process is created by Syzkaller. As indicated in the Table, 20 of the 22 crashes (91%) found by KASAN occurred in the process generated by the fuzzer. In addition, 50 out of a total of 57 crashes (88%) occurred as a result of fuzzer-generated processes. Among the bugs detected by KASAN, bugs with different PIDs have occurred in objects allocated from system daemons such as *systemd* and *systemd-udevd*. If the process is selectively sanitized, more than 80% of crashes can be detected, and the performance overhead caused by dynamic instrumentation can be minimized.

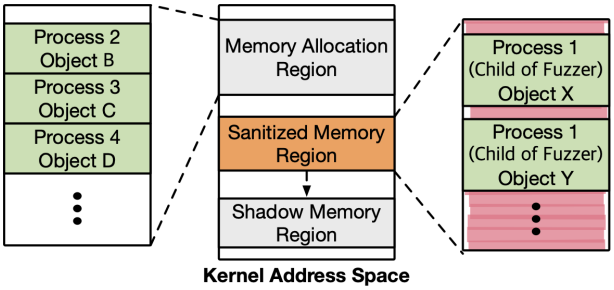


Figure 3: Overview of the kernel memory with selective sanitization applied. Unlike KASAN, selective sanitization is applied only to objects allocated in the sanitized memory region.

2.4 Our Approach: Selective Sanitization

The insight we gained by making these interesting observations led us to the idea of sanitizing the kernel by selecting the target process "well". Thus, our basic idea is to selectively sanitize only the memory objects allocated by the target process to minimize the sanitizing overhead. Careful selection of the target process means to select the process created by the fuzzer to call a random system call or a specific process running for kernel vulnerability analysis. Figure 3 depicts the concept of *selective sanitization*. In general, in the kernel, one memory allocation area is shared by multiple processes, and KASAN sanitizes all processes using this area (§2.3). On the other hand, selective sanitization does not entail applying the sanitizer to all the memory objects of a process; instead, it only applies the sanitizer to those of the target process. As shown in Figure 3, objects allocated by the target process, *process 1*, have a red zone, and objects allocated by other processes do not have a red zone. Selective sanitization divides the memory region into a *memory allocation region* where the existing kernel objects are allocated and a *sanitized memory region* where the sanitized objects are allocated.

3 Design

We designed and implemented a binary-only KASAN BoKASAN, which practically performs address sanitizing through dynamic instrumentation (without the source code) for the entire kernel binary. The key accomplishments of BoKASAN are as follows: (1) BoKASAN dramatically reduces the performance overhead of dynamic instrumentation by performing selective address sanitizing, which identifies the process targeted for vulnerability analysis among the processes running in the kernel. (2) Dynamically hooks the page fault mechanism to enable selective address sanitization on a KVM-enabled virtual machine.

Figure 4 presents an overview of BoKASAN. First, BoKASAN registers the target process for selective sanitization (❶). The target process can be created by a fuzzer or a specific process executed for vulnerability analysis. After registering the target process, the registered process calls the sanitized allocation instead of the default allocation mechanism when allocating a kernel object (❷). In sanitized allocation, objects are allocated to a sanitized memory region separate from the region generally allocated to the kernel memory, and the page present bit is cleared in this region to trap future memory access. Subsequently, when the memory in the sanitized memory region is accessed, the hooked page fault handler checks the validity of the memory address, and BoKASAN raises a kernel panic when an inaccessible address is accessed (❸). Finally, when the memory allocated to the sanitized memory region is deallocated, sanitized deallocations are performed (❹). The deallocated memory area is never used in the future, and when access to this area occurs, a use-after-free error is reported.

3.1 Target Selection

To apply selective sanitization, BoKASAN first registers the sanitization target (❶ in Figure 4). After the targets are registered, the memory allocator checks whether the requested process is registered when a request occurs. In the case of a registered process, sanitization is applied; otherwise, default memory allocation is performed. BoKASAN uses a PID to identify a sanitized target, which can be registered when a kernel fuzzer (e.g., Syzkaller) creates a new process to test a syscall. In addition, BoKASAN covers all descendant processes by hooking *fork()*. Because BoKASAN mostly incurs overhead when the page fault handler is executed, the overall overhead is minimized by generating a page fault only for objects to which the sanitizer had been applied through a separated memory area. One might be concerned that bugs that occur in non-children processes may be missed. We can alleviate this problem by randomly applying sanitization for the objects allocated by non-target processes. Note that the method of sanitizing random allocations is already used in memory error detectors applied to production-level software that requires minimal overhead. For example, GWP-ASan [29] is applied to user space heap allocation, and KFENCE [23] is used to heap allocation of the Linux kernel. As described in §2.3, our selective sanitization detected more than 90% of bugs without random sanitization, so we did not implement this feature in the current version of BoKASAN. Further research can apply random sanitization to minimize false negatives.

The use of selective sanitization enables BoKASAN to use a larger red zone and less shadow memory. For example, because BoKASAN sanitizes only approximately 12.5% of the total allocation (Table 11), only the shadow memory corresponding to the object needs to be allocated. In addition, owing to selective sanitization, BoKASAN uses a larger red zone, it was able to accurately classify the bugs misclassified by KASAN (§5.1). Since the address sanitizer cannot detect OOB access that exceeds the size of the red zone, it is possible to reduce such false negatives as the red zone grows.

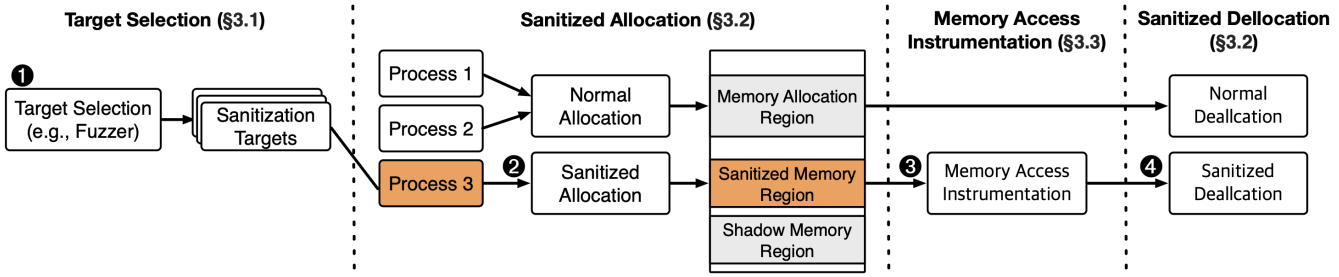


Figure 4: Overview of BoKASAN

3.2 Memory Allocation and Deallocation

BoKASAN dynamically hooks the memory allocation and deallocation functions of the Linux kernel to implement the sanitizer (the target functions are greater detail in §4). The process whereby BoKASAN allocates and deallocates memory is as follows.

Allocation. BoKASAN hooks the memory allocation functions (e.g., *kmalloc*) to allocate the object including the red zone (❷ in Figure 4). The memory allocation process consists of 1) checking whether the process is a target process; 2) allocating the requested memory including the red zone; 3) initializing the shadow memory; and 4) clearing the page present bit.

The use of selective sanitization requires a sanitization target to be allocated among the memory allocations requested by kernel processes. Because BoKASAN hooks the allocation functions of the kernel, the allocation handler is called whenever an allocation is requested. The handler checks whether the process that requested allocation is the target process. To do this, it checks the PID of the current process, and if the process is registered in BoKASAN, sanitized allocation is executed. However, if it is an unregistered process, the requested allocation is performed without generating a red zone.

Sanitized allocation allocates a larger object than the requested object, which includes a red zone when allocating an object, and sets the shadow value to the shadow memory. To initialize the shadow memory, we first obtain the shadow memory address corresponding to the address of the allocated object and check whether this address area is already allocated. If the obtained shadow memory address is an unmapped area, BoKASAN allocates shadow memory to page granularity. When shadow memory is allocated, the shadow memory of the memory page containing the allocated object is set to a shadow value, indicating that it is a sanitized page (e.g., BOKASAN_PAGE). To separate the sanitized memory region from the memory allocation region, only the target process can allocate memory objects in the area where the shadow value is BOKASAN_PAGE. After the initialization of the shadow memory, the allocated shadow memory of the allocated object is set to the shadow value indicating the sanitized object (e.g., BOKASAN_OBJECT), and the red zone of the allocated object is set to the shadow value indicating the address of the red zone (e.g., BOKASAN_REDZONE). For example, if an object of size 1024 bytes is allocated, BoKASAN allocates shadow memory of the size of a page (4096 bytes), and 512 bytes ($4096 \gg 3$) of shadow memory corresponding to the page containing the allocated object is initialized to BOKASAN_PAGE. Thereafter, 128 bytes ($1024 \gg 3$) of the shadow memory corresponding to the allocated object is set to BOKASAN_OBJECT, and the shadow memory corresponding to the red zone of the object is set to BOKASAN_REDZONE. Details of the implementation of the shadow memory are provided in §4.

After allocating memory, BoKASAN clears the present bit of the page containing the allocated object by manipulating the value of the page table entry. At this time, if the object is located over several pages because the size of the allocated object is larger than the page size, the present bits of all pages including the object are cleared. As a result, when the object is accessed in the future, the page fault handler is invoked, and memory access instrumentation can be performed. Finally, to indicate that the present bit of the page is cleared by BoKASAN, we set the PAGE_SPECIAL flag.

Deallocation. BoKASAN hooks memory deallocation functions (e.g., *kfree*) to perform sanitized deallocation (❹ in Figure 4). Because selective sanitization is also applied to memory deallocation, it selectively sanitizes memory objects that are freed in the kernel space. BoKASAN only sanitizes the objects in the sanitized memory region by checking the shadow memory of the object for which deallocation is requested (i.e., when the shadow value of the object is BOKASAN_OBJECT).

The process of sanitized deallocation is as follows: 1) Changing the shadow value of the freed object to a value representing the freed object (e.g., BOKASAN_FREE) and 2) the object is not actually freed but remains in memory (i.e., the address of the freed object will never be used). This enables BoKASAN to check the shadow memory when the freed object is accessed in the future. If a non-sanitized object is freed, the memory is freed by the deallocation function of the kernel.

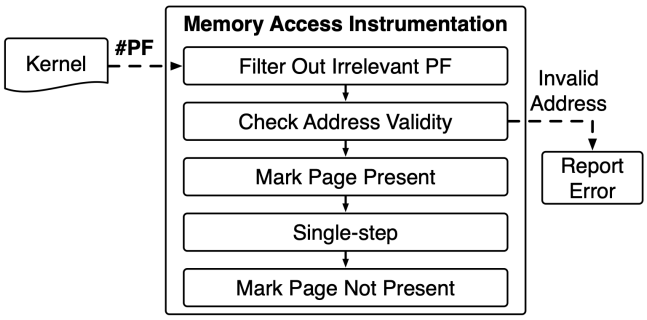


Figure 5: Memory access instrumentation overview.

3.3 Memory Access

BoKASAN hooks the kernel page fault handler and debug exception handler for the memory access instrument (❸ in Figure 4). When a kernel process accesses the sanitized memory region, a page fault occurs because the present bit of that region is not set. When a page fault occurs, the page fault handler is executed, and the faulted instruction address, accessed memory address, and other flags are delivered to the handler. BoKASAN uses this information to check whether the memory access is valid. If it is valid, the trapped instruction is single stepped, and the present bit of the accessed address is cleared in the debug exception handler. Otherwise, if the memory address is invalid, BoKASAN raises a kernel panic with an error report. Figure 5 shows the memory access instrumentation process.

Filter Out. BoKASAN only investigates page faults that occur in the sanitized region. If the accessed address is not a sanitized memory region (i.e., when a page fault occurs for reasons other than a fault caused by BoKASAN clearing the present bit), BoKASAN allows the default page fault handler to process the fault. Otherwise, BoKASAN checks the validity of the accessed address. If the memory allocation region and the sanitized memory region are completely separated and independent, as shown in Figure 3, the faulted address can be easily checked and filtered. However, because BoKASAN uses a page-level sanitized memory region (additional details are provided in §B), the address of the sanitized memory region is flexible, thus it cannot be filtered by simply comparing address ranges. Because the PAGE_SPECIAL flag of the sanitized page is always set, BoKASAN refers to the flag value of the accessed memory pages to determine the page to be filtered.

Check Address Validity. BoKASAN checks the shadow memory of the accessed address to validate the address. If the value of the shadow memory indicates an error, a kernel panic is raised, and an error report is generated. For example, if the shadow value is BOKASAN_REDZONE, BoKASAN reports an OOB access bug and raises a kernel panic. BoKASAN provides the accessed address, the instruction address where the corruption occurred, and the call stack obtained by *dump_stack()* in a format similar to KASAN.

Table 3: Memory allocation and deallocation functions.

Category	Function
Allocation	kmem_cache_alloc, kmem_cache_alloc_node, __kmalloc, kmem_cache_alloc_trace, __kmalloc_node, kmem_cache_alloc_node_trace, __kmalloc_track_caller, __kmalloc_node_track_caller, kmalloc_order, kmalloc_large_node
Deallocation	kmem_cache_free, kfree

Mark Page Present. If the shadow memory check determines the trapped address to be valid, the trapped instruction should be executed. However, the address the trapped process attempts to access is a page without a present bit, thus it cannot be executed until the present bit is set. Therefore, BoKASAN temporarily sets the present bit of the page to execute the trapped instruction.

Single-step. Now, the present bit of the page is set, the instruction is executed without a fault. BoKASAN regains control after executing a single instruction by utilizing the single-step supported by the processor. A single-step generates a debug exception after one instruction is executed, and before the next instruction is executed. BoKASAN hooks a debug exception handler to bring control back after a single step.

Mark Page Not Present. Finally, a trapped instruction is executed through a single step, and the present bit of the accessed page is cleared in the debug exception handler to trap future access to the memory page.

4 Implementation

We implemented BoKASAN with about 2,000 lines of C code as a Linux kernel module and modified */etc/rc.local* to load the module during kernel booting. Function instrumentation was implemented using ftrace-based *ftrace-hook* [17], and it hooks memory allocation/deallocation functions, page fault handler, and debug exception handler. To trace a function using *ftrace*, a kernel requires CONFIG_FUNCTION_TRACER, which is enabled by default in major Linux distributions such as Debian, Fedora, and Ubuntu. If CONFIG_FUNCTION_TRACER is disabled in the target kernel, function hooking can be implemented through Kprobes [14] or inline hooking (a.k.a splicing). Kernel function hooking is widely used not only in Linux but also in other operating systems such as Windows, macOS, and iOS [7, 20, 32]. Note that, the implementation details of BoKASAN of Windows version are described in §A. In this section, we provide implementation details of BoKASAN.

4.1 Function Instrumentation

BoKASAN instruments the memory allocation and deallocation functions of the Linux kernel. For the performance of the kernel, the allocation functions are usually implemented as an inline function. Because the inline function is inserted directly into the code of the calling function, a separate function is not created. Therefore, function-level instrumentation cannot be applied to inline functions. To address this problem, BoKASAN instruments the functions executed inside an inline allocation function. Table 3 lists the functions of the BoKASAN instrument.

When BoKASAN allocates an object, it additionally allocates a red zone area. Owing to selective sanitization, BoKASAN uses a larger red zone of 256–2048 bytes than KASAN, which uses a red zone of 16–2048 bytes. Because BoKASAN only sanitizes 12.5% of the allocated objects (§5.3), a larger red zone can be used compared to KASAN. Among the allocation functions, because *kmem_cache_alloc(_node)* allocates objects of a predetermined size when a slab cache is created, it is difficult to allocate additional sizes for the red zone. To change the size of the object to which *kmem_cache_alloc* is allocated, the *object_size* of the slab cache must be changed, but most slab caches are created by calling *kmem_cache_create* before the BoKASAN module is loaded. Therefore, it is impossible for BoKASAN to increase the *object_size* of the slab cache by hooking this function. BoKASAN addresses this problem by allocating two consecutive objects and using the object on the right as a red zone. Because objects are not always allocated consecutively, BoKASAN attempts a total of ten allocations. For example, if BoKASAN allocates the requested object by calling kmem_cache_alloc, it allocates one additional object for use as the red zone. If the object allocated to the red zone is not contiguous with the requested object, BoKASAN allocates a new red zone object and repeats it up to ten times. If consecutively allocated objects are not found, only the UAF bug is detected without creating a red zone for the object.

In addition to the allocation function, BoKASAN instruments page faults and debug exceptions by hooking *do_page_fault* and *do_debug*, respectively. The page fault handler checks the shadow memory when the page where the *PAGE_SPECIAL* flag is set is accessed. The debug exception handler clears the present bit of the page after executing the instructions trapped by the page fault handler. BoKASAN accomplishes this by checking the *DR_STEP* flag of the *DR6* register, and if set, clears the *DR6* register, *X86_EFLAGS_TF* flag, and page present bit in the debug exception handler.

4.2 Shadow Memory Management

BoKASAN is similar to KASAN in that it utilizes shadow memory. The address of the shadow memory is computed using one shift and one addition, as follows:

$$shadow\_addr = (addr \gg 3) + shadow\_offset$$

The problem that arises when implementing shadow memory is that KASAN uses the shadow address region reserved for the Linux kernel; however, when KASAN is not enabled, we cannot allocate memory in this region. BoKASAN overcomes this problem by allocating shadow memory in the vmalloc region rather than in the shadow address region. BoKASAN reserves an area of approximately 8TB for shadow memory (0xffffe0f000000000–0xffffe8ffffffffff) in the vmalloc area. The reserved area corresponds to the 64 TB direct mapping area (0xffff880000000000–0xffffc87fffffffff) where memory objects are allocated [22]. BoKASAN accomplishes this by setting *shadow_offset* to 0xdfffeff000000000. When the sanitized memory object is allocated, BoKASAN allocates the shadow memory of the object located in the shadow memory region for BoKASAN in page granularity (4 KB). In addition, BoKASAN sets the shadow values to BOKASAN_FREE for the pages ranging from 0xffff880000100000 to 0xffff880000200000 when loading the module. These address ranges are freed by the kernel before BoKASAN is loaded.

4.3 Fuzzing

BoKASAN can be integrated with existing kernel fuzzers, for example, Syzkaller, the fuzzer we selected. For selective sanitization, before Syzkaller executes a program consisting of a random sequence of system calls, the identifier of the worker process of the fuzzer is registered in BoKASAN. For example, Syzkaller creates several worker processes and executes a random program for each worker. At this time, when each worker process is executed, the PID is registered in BoKASAN to ensure that only the registered process is sanitized. To this end, we modified the *worker_thread()* of the executor of Syzkaller and added code to register the PID in the function. The PID registered through IOCTL communication, and the *thread group id* value, which is the identifier of the target process obtained by the *get_tgid()*, is transferred to the BoKASAN.

BoKASAN can also be used with non-Syzkaller-oriented kernel fuzzers such as kAFL. kAFL executes a fuzzer agent when fuzzing, and BoKASAN can be applied by calling IOCTL in the agent to register the target process.

5 Evaluation

We evaluated the performance of BoKASAN by attempting to answer the following questions:

- **RQ1:** Can BoKASAN successfully detect OOB and UAF bugs targeting only the kernel binary? (We tested this on a Linux kernel and compared the result with source-based KASAN.)
- **RQ2:** Is the amount of performance overhead incurred by BoKASAN acceptable?

- **RQ3:** To what extent is the selective sanitization of BoKASAN effective?
- **RQ4:** Can BoKASAN be applied to binary-only fuzzing?

Dataset. Known Linux kernel bugs were used to test the bug detecting capability of the sanitizers. We used bugs found by Syzkaller [59], SyzVegas [61], and Janus [64] as datasets. First, we selected 23 OOB and UAF bugs out of 62 Linux kernel bugs reported to Bugzilla by the author of Janus [64]. Since there is no ground-truth benchmark for evaluating kernel fuzzing yet, we used a Janus-based dataset consisting of the bugs discovered in the file system. To minimize the bias on the file system, we selected the 16 bugs in various kernel components found by Syzkaller and SyzVegas.

Sanitizers. We used three sanitizers, KASAN, KASAN (FE), and BoKASAN, in the experiments. KASAN and KASAN (FE) use a KASAN-applied kernel during compile time. KASAN (FE) was tested on a VM that was fully emulated without using KVM, thereby taking into consideration that KASAN was implemented on the emulator. BoKASAN loads a module into the kernel after the kernel is booted, and was tested on the KVM-enabled VM.

Platform and Configuration. §5.1, §5.2, and §5.3 were conducted on an Ubuntu 16.04 server with two Intel Xeon Gold 6148 CPUs (2.40 GHz * 80 cores) and 384 GB of RAM. Each fuzzing campaign was performed on the three VMs, each of which had one CPU and 4 GB of RAM. §5.4 was conducted on an Ubuntu 20.04 with an Intel i7-12700 CPU and 64GB of RAM. In consideration of the randomness of fuzzing, 20 repeated experiments were performed simultaneously according to published guidelines [25].

The main goal of evaluation in this paper was to see whether BoKASAN could find the bugs KASAN discovered. Therefore, we decided to use Syzkaller, which was easy to build environment, for our evaluation. Syzkaller was compiled from git commit *fdb2bb2c23ee7* using the default configuration. We disabled the reproducing process of Syzkaller when performing fuzzing. This is because the code coverage is affected by the number of bugs found when reproducing is enabled. For the experiment, Linux kernel v4.19 was built according to the guidelines of Syzkaller, and we set the *CONFIG_FUNCTION_TRACER* to use BoKASAN. Note that *CONFIG_FUNCTION_TRACER* is enabled by default in most Linux distributions. In addition, to operate KASAN (FE) in Syzkaller properly, we increased slowdown parameter to 10 for full emulation experiments. The slowdown is a scaling factor for configuring syscall timeout.

5.1 Bug Detection

To answer RQ1, we compared the number of bugs detected by KASAN and BoKASAN using the Janus and Syz dataset. Each bug was tested on the reported version of the kernel, and the code for applying the target selection of BoKASAN to the provided POC code was added. Additionally, to evaluate the bug detection capability of BoKASAN in fuzzing, the number of bugs detected by KASAN and BoKASAN was compared as a result of fuzzing for 5 days using Syzkaller. Because the kernel is larger than that in user-level software, we conducted long-term experiments in order to minimize the randomness of fuzzing when attempting to detect bugs. Owing to time constraints, only BoKASAN and KASAN were evaluated in the long-term experiment. Each fuzzing experiment was run on three VMs and was repeated 20 times (total of 360 CPU hours per fuzzing campaign).

Table 4: Overview of detected bugs by BoKASAN and KASAN, resp., on Janus benchmark.

#	FS	Report ID	Version	Type	KASAN	BoKASAN
1	EXT4	199181	4.15	OOB	✗	✗
2		199347	4.16-rc1	OOB[1]	✓	✓
3		199403	4.16-rc1	UAF	✓	✓[2]
4		199417	4.16-rc1	OOB	✓	✓
5		199865	4.17-rc4	OOB	✓[2]	✓
6		200001	4.17-rc4	OOB	✓[3]	✓[2]
7		200401	4.17-rc4	OOB[1]	✓	✓[2]
8		200931	4.18	UAF	✓	✓
9	XFS	199371	4.16-rc1	UAF	✓	✓
10		199373	4.16-rc1	UAF	✓	✓
11		199381	4.15.13	OOB[1]	✗	✓
12		199443	4.17-rc1	UAF	✓	✓[2]
13		200047	4.17-rc4	OOB	✓	✓
14		200053	4.17-rc4	OOB	✓	✓
15		200923	4.18	OOB	✓	✓
16	BTRFS	199837	4.17-rc5	OOB	✗	✗
17		199839	4.17-rc5	UAF	✓	✓[2]
18	F2FS	200167	4.18-rc1	OOB	✓[3]	✓[2]
19		200173	4.18-rc1	OOB	✓	✓
20		200179	4.18-rc1	UAF	✓	✓[2]
21		200219	4.18-rc1	OOB[1]	✓	✓
22		200419	4.16-rc1	OOB	✓	✓
23		200421	4.18-rc3	OOB	✓	✓[2]
Total					20	21

[1] A bug is triggered in the memcpy and memmove.
[2] Bugs can be detected depending on the allocated condition of the objects.
[3] A bug reported by KASAN as UAF but actually OOB.

Dataset Result. Table 4 lists the bugs detected by BoKASAN and KASAN. The third and fourth columns contain the bug id and bug type.

Based on the result in Table 4, BoKASAN and KASAN detected 21 and 20 bugs, respectively, among a total of 23 bugs. This shows that BoKASAN can detect slab OOB and UAF bugs that are detectable by KASAN even without source code being present in the Linux kernel; furthermore, it can detect a bug that KASAN missed. Some of the bugs detected by BoKASAN could be detected according to the location of the object, and most of their allocation functions were undetermined. This occurs when the bug is triggered by accessing the red zone of another object by greatly deviating from the address of the object where the OOB occurred. Considering that KASAN sanitizes most of the objects, even in this case,

Table 5: Bugs detected by BoKASAN and KASAN, respectively, on Syz (Syzkaller and SyzVegas) benchmark dataset.

#	Function	Version	Type	KA.	BoK.
1	vcs_write	4.19	OOB	✓	✓
2	ata_scsi_mode_select_xlat	4.19	UAF	✓	✓
3	clear_buffer_attributes	4.19	UAF	✓	✓
4	complement_pos	4.19	UAF	✓	✓
5	con_scroll	4.19	UAF	✓	✓
6	con_shutdown	4.19	UAF	✓	✓
7	do_con_write	4.19	UAF	✓	✓
8	do_update_region	4.19	UAF	✓	✓
9	get_work_pool_id	4.19	UAF	✓	✗
10	screen_glyph_unicode	4.19	UAF	✓	✓
11	vc_do_resize	4.19	UAF	✓	✓
12	vcs_write	4.19	UAF	✓	✓
13	vc_uniscr_check	4.19	UAF	✓	✓
14	vgacon_invert_region	4.19	UAF	✓	✓
15	vgacon_scroll	4.19	UAF	✓	✓
Total				15	14

the bug is highly likely to be detected. BoKASAN can alleviate this problem by increasing the total number of allocated red zones by allocating objects dedicated to the red zone at regular intervals.

Our observations indicated the existence of bugs that were detected but misclassified because KASAN uses a small red zone. The bugs with the identifiers 200001 and 200167 in Table 4 were actually OOB bugs, but KASAN detected them as UAF. Because OOB access occurred beyond the red zone, KASAN can detect the bug only when other objects are deallocated behind the object that actually triggers the bug. Because BoKASAN uses a larger red zone compared to KASAN, it accurately detected these bugs as OOB.

Table 5 shows the results of bug detection by BoKASAN and KASAN to the Syz dataset composed of bugs discovered by Syzkaller and SyzVegas. As a result of the experiment, among 15 bugs, KASAN and BoKASAN detected 15 and 14 bugs, respectively. A bug detected in *get_work_pool_id* that BoKASAN did not detect is triggered through *tty_release* function. The Syzkaller generated reproduce code of this bug calls *perf_event_open* syscall, but this syscall collides with BoKASAN, causing a kernel panic, as a result, BoKASAN fails to detect this bug. However, this does not mean that BoKASAN cannot detect this bug. After detecting the bug in *con_shutdown*, BoKASAN detected UAF in *release_tty* and *__cancel_work_timer* which are the caller of *get_work_pool_id*. After detecting the bug in *con_shutdown*, BoKASAN detected UAF in *release_tty* and *__cancel_work_timer* which are the caller of *get_work_pool_id*. From this, we can infer that these two bugs have some relationship, and if we can trigger UAF in *get_work_pool_id* without *perf_event_open*, we can detect this bug. Similarly, a bug that KASAN detected in *do_update_region* was detected by BoKASAN in *csi_J*. Note that, *do_update_region* is called in *csi_J*.

Fuzzing Result. Table 6 lists the bugs detected by KASAN and BoKASAN as a result of 5 days of fuzzing using Syzkaller. The third and fourth columns show the number of bugs found by KASAN and BoKASAN in 20 repetitive experiments. Among the 30 unique bugs, BoKASAN and KASAN discovered 21 (178 in total) and 21 (139 in total) bugs, respectively. Surprisingly, BoKASAN detected the same number of unique bugs as KASAN and a larger number of bugs in total. One might be concerned that bugs discovered only by BoKASAN may include false positives, thus we performed additional fuzzing using Linux kernel v4.19.219. If a bug is found in v4.19.219, the possibility of false positives exists, but as a result of the experiment, all the bugs that BoKASAN detected in v4.19 were not detected in v4.19.219. As a result, in this case of fuzzing, BoKASAN was able to detect a larger number of bugs than KASAN, which means that BoKASAN can be utilized for binary-only kernel fuzzing.

Table 6: Overview of bugs found as a result of 5d fuzzing.

Type	Function	KA.	BoK.
user-memory-access	n_tty_set_termios	1	0
use-after-free	vt_do_kdgkb_ioctl	0	1
use-after-free	vgacon_scroll	4	4
use-after-free	vgacon_invert_region	15	18
use-after-free	vgacon_blank	0	6
use-after-free	vcs_write	4	18
use-after-free	vc_uniscr_check	9	13
use-after-free	vc_do_resize	20	20
use-after-free	try_to_grab_pending	0	3
use-after-free	screen_glyph_unicode	8	13
use-after-free	screen_glyph	0	1
use-after-free	n_tty_receive_buf_common	1	4
use-after-free	iowrite32_rep	0	2
use-after-free	ioread32_rep	0	4
use-after-free	get_work_pool_id	8	0
use-after-free	do_update_region	7	0
use-after-free	do_con_write	15	17
use-after-free	csi_J	0	20
use-after-free	con_shutdown	5	0
use-after-free	con_scroll	3	11
use-after-free	complement_pos	5	2
use-after-free	clear_buffer_attributes	2	2
use-after-free	ata_scsi_mode_select_xlat	2	0
use-after-free	__xfrm_policy_unlink	2	0
stack-out-of-bounds	xfrm_state_find	6	0
slab-out-of-bounds	vcs_write	20	15
slab-out-of-bounds	sg_remove_sfp	0	1
slab-out-of-bounds	ioread32_rep	0	3
slab-out-of-bounds	corrupted	1	0
slab-out-of-bounds	ata_scsi_mode_select_xlat	1	0
Total		139	178
Total Unique		21	21

5.2 Performance Overhead

To answer RQ2, we compared the execution time for syscalls of BoKASAN, baseline, KASAN, and KASAN (FE) using LMbench. We tested six syscalls in our experiments, and each syscall was measured 20 times by using the *-N* option of *lat_syscall*. In addition, we performed fuzzing for 24h to analyze whether BoKASAN incurs acceptable overhead for fuzzing. Syscall was divided into three categories; all, file system, and network, and 100 and 18 file system and network

Table 7: LMBench Result. Time is in microseconds. Numbers in parentheses indicate overhead compared to baseline.

Syscall	Baseline	KASAN	BoKASAN	KASAN (FE)
null	0.228 (-)	0.243 (6.9%)	0.2401 (5.5%)	1.4015 (663.7%)
open/close	2.184 (-)	7.540 (245.2%)	2,076.0000 (94,954.9%)	188.571 (8,534.2%)
read	0.470 (-)	0.529 (12.5%)	0.456 (-3.0%)	7.550 (1,506.8%)
write	0.338 (-)	0.365 (8.1%)	0.349 (3.4%)	4.213 (1,147.8%)
stat	1.312 (-)	4.297 (227.5%)	149.600 (11,303.3%)	82.920 (6,220.6%)
fstat	0.361 (-)	0.457 (26.5%)	0.360 (-0.2%)	4.141 (1,047.8%)

syscalls were selected by referring to [1]. Finally, we present the results of the micro evaluation in §C.

LMbench Result. Table 7 lists the execution time for each syscall evaluated on LMbench. BoKASAN has significantly different execution times for different syscalls. In *null*, *read*, *write*, and *fstat*, the amount of overhead was almost similar to the baseline, and in *open/close* and *stat*, the amount of overhead exceeded that of KASAN (FE). According to our analysis, a very large number of *memset-like* functions were executed in *open/close* and *stat*. These data manipulation functions cause a large number of memory accesses, resulting in a large number of page faults in BoKASAN. Currently, in BoKASAN, the number of page faults is determined by the size requested to *memset*. In this case, as described in §2.2, the overhead may be approximately 1,000 times larger than the baseline. In spite of the overhead, BoKASAN was able to test a much larger number of syscalls than KASAN (FE) in fuzzing owing to selective sanitization and the low overhead incurred by syscalls such as *read*, *write*, and *fstat*. To alleviate this problem, we plan to optimize BoKASAN such that when a function such as *memset* is called, it can check the shadow memory of the requested memory region in a single page fault.

Fuzzing Result. Table 8 lists the number of executed syscalls as a result of 24h fuzzing. As a result of the experiment, when all syscalls were targeted, BoKASAN executed approximately 5.1 times more syscalls than KASAN (FE), and 10.5% fewer syscalls than KASAN. Kernel fuzzing is performed by executing a syscall, thus executing more syscalls can increase the code coverage, and consequently, the likelihood of finding bugs. Figure 9 in §D shows the number of executed syscalls as a result of 24h fuzzing and shows that BoKASAN executes a much larger number of syscalls than KASAN (FE) over time and almost the same number of syscalls as KASAN. The overhead of BoKASAN is incurred only when memory access to sanitized page occurs. Therefore, BoKASAN was able to execute many more syscalls compared to KASAN (FE),

Table 8: Number of syscall executions in 24h fuzzing.

Syscall	Number of syscall execution (median)				
	Baseline	KASAN	BoKASAN	Base.(FE)	KA.(FE)
All	2,598,575	2,206,090	1,973,865	567,713	388,696
File sys.	4,743,271	3,943,488	1,617,095	656,461	423,549
Network	4,688,673	4,435,589	2,559,070	714,866	422,210

Table 9: Number of discovered basic blocks in 24h fuzzing.

Syscall	Number of discovered basic blocks (median)				
	Baseline	KASAN	BoKASAN	Base.(FE)	KA.(FE)
All	71,921	69,807	69,425	61,107	56,014
File sys.	30,496	30,585	29,984	29,650	29,209
Network	22,412	22,661	23,314	18,739	17,296

which had to emulate all instructions not related to sanitization. In addition, owing to selective sanitization, BoKASAN was able to execute a similar number of syscalls to KASAN by checking memory access only for objects that need sanitizing.

The experiment on syscalls related to the file system revealed that BoKASAN executed fewer syscalls than when all syscalls were targeted. As indicated by the LMbench results, a large number of memory copies occur in file system-related syscalls, and as a result, BoKASAN is responsible for a large number of page faults. Nevertheless, BoKASAN was still able to execute approximately 3.8 times more syscalls compared to KASAN (FE). BoKASAN can optimize this problem by enabling memory access functions such as *memcpy* to check the red zone with a single page fault.

Table 9 lists the number of basic blocks executed as a result of 24h fuzzing. BoKASAN executed approximately 23.9% more basic blocks than KASAN (FE) and executed almost the same number of basic blocks as KASAN when all syscalls were targeted. BoKASAN executed a larger number of syscalls than KASAN (FE), and as a result, its code coverage was higher than that of KASAN (FE).

5.3 Effectiveness of Selective Sanitization

To answer RQ3, we conducted 24h fuzzing with BoKASAN by ablating selective sanitization and compared the results to the original BoKASAN with selective sanitization. We accomplish this by sanitizing all the objects allocated by the allocation functions except for those including “node" in the allocation function name. When sanitizing all objects allocated by these functions, such as *__kmalloc_node*, the kernel does not run normally. In addition, we compared the number of sanitized allocations among all allocations executed while fuzzing. This shows how many allocations are minimized by BoKASAN. Finally, we compared the number of sanitized syscalls among all executed syscalls during fuzzing.

Result. Figures 6 and 7 show the results of 24h fuzzing with and without selective sanitization. When selective sanitization was applied to BoKASAN, 4.3 times more syscalls were

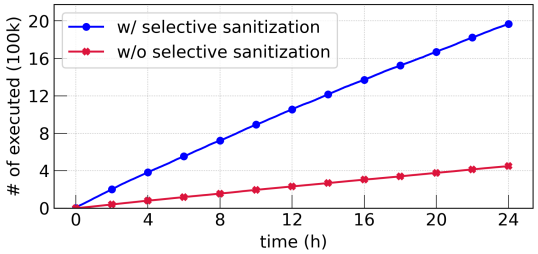


Figure 6: Number of syscalls executions during 24h fuzzing using BoKASAN w/ and w/o selective sanitization, respectively.

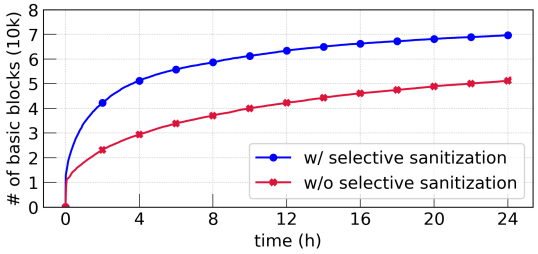


Figure 7: Number of discovered basic blocks during 24h fuzzing using BoKASAN w/ and w/o selective sanitization, respectively.

executed and 32.8% higher code coverage was achieved than when selective sanitization was not applied. Without selective sanitization, lower code coverage was achieved than KASAN implemented based on full emulation. In addition, BoKASAN detected 14 and 6 when selective sanitization was applied and when not, respectively. These results support that selective sanitization helps BoKASAN to validate the address of the accessed memory effectively.

Table 10 lists the total number of the allocation function executed during fuzzing and the number of sanitized among them. Notably, selective sanitization selected only 12.5% of allocations to be sanitized. The three functions at the bottom of the table are sanitized at a high rate because these functions are not frequently used by background processes, and most allocations are requested by the fuzzer process. The allocations requested by these functions occupy a small portion of the total allocations. Owing to selective sanitization, BoKASAN effectively reduced performance overhead incurred from irrelevant allocations without significant detriment to its bug detection capability.

We evaluated selective sanitization by comparing the number of sanitized syscalls among all executed syscalls during fuzzing. We hooked following functions for the experiments: *__x64_sys_open*, *ksys_read*, *ksys_write*, *__x64_sys_newstat* , and *__x64_sys_newfstat*. Table 11 lists the total number of system calls executed and the number of sanitized as a result of 24h of fuzzing using a single VM. As a result, only 16% of *open* is sanitized among all the executed *open* owing to selective sanitization. In Table 7, BoKASAN incurred the largest

Table 10: Number of the allocation functions executed during fuzzing and the number of sanitized allocations among them.

Function	Total	Sanitized	Rate (%)
kmem_cache_alloc	5,496 m	660 m	12.0
kmem_cache_alloc_trace	329 m	62 m	19.0
__kmalloc	129 m	25 m	19.3
kmem_cache_alloc_node	288 m	19 m	6.7
kmem_cache_alloc_node_trace	66 m	4 m	6.4
__kmalloc_node	98 m	11 m	11.3
__kmalloc_track_caller	257 m	13 m	5.2
__kmalloc_node_track_caller	16 m	9 m	61.3
kmalloc_order	57 k	56 k	99.2
kmalloc_large_node	24 k	17 k	71.0
Total	6,431 m	806 m	12.5

Table 11: Number of syscalls executed on fuzzing. Sanitized syscalls and their rates are also described.

Syscall	Total	Sanitized	Rate (%)
open	438 m	72 m	16.5
read	689 m	54 m	8.0
write	1,133 m	594 m	52.5
stat	95 k	10 k	10.6
fstat	413 k	27 k	6.6
Total	2,261 m	721 m	31.9

overhead in *open*, however, because only 16% of *open* was sanitized, the overhead incurred during fuzzing is much less than that measured using the syscall benchmark. Although some syscalls incur large overhead, BoKASAN effectively reduced such overhead by sanitizing only a part of the syscalls during fuzzing.

5.4 Binary-only Fuzzing

To answer RQ4, we conducted fuzzing using kAFL targeting a binary-only kernel driver on Ubuntu 16.04 and Windows 10 21H2. Ubuntu and Windows run Linux kernel 4.15-140 and Windows 10 Build 19044.1288 x86 kernel, respectively. To evaluate the efficiency (performance overhead) and effectiveness (bug detection capability) of BoKASAN for binary-only drivers, we decided to use the *kafl_vuln_test* provided by kAFL [49] as the target kernel driver. This driver contains three bugs, and the input values need to satisfy 'KERNELAFL', 'SERGEJ', and 'KASAN' to trigger each bug. To evaluate the performance overhead generated by BoKASAN, we selected the first two bugs that can be detected even in baseline fuzzing without applying the sanitizer. To evaluate the effectiveness of BoKASAN, we chose the last bug, which cannot be detected in baseline fuzzing. Because in the bug triggered by 'KASAN', *kfree* is called twice and crashed even without sanitizer, we modified it to return after the UAF is triggered to prevent double free. In Windows, we ported the Linux version of *kafl_vuln_test* to the Windows for evaluation. The source code and details of the Windows driver used in the experiments can be found in §E.

Table 12: Result of fuzzing binary-only driver by kAFL

Target	Sanitizer	Time to Crash (s) Bug1	Bug2	Bug3	#Execs
Linux	Baseline	8.72 (20)	3.88 (20)	1.07 (3)	1,482,702
	BoKASAN	20.56 (20)	4.87 (20)	6.68 (20)	997,259
Windows	Baseline	48.63 (20)	25.32 (20)	- (0)	722,437
	BoKASAN	54.23 (20)	28.71 (20)	75.50 (20)	288,771

Results. Table 12 lists the result of fuzzing using kAFL. Since KASAN is not applied to the kernel of Ubuntu, we compared BoKASAN with the baseline. As shown in the table, BoKASAN took 2.56, 1.26, and 6.24 times longer time than the baseline to trigger Bug1, Bug2, and Bug3 on Ubuntu. The number in parentheses shows the number of times the bug was detected out of 20 attempts. In the case of Bug3, the baseline detected only three times, while BoKASAN detected all 20 times. Without BoKASAN, the bug was detected only when freed memory is used by other processes, and when BoKASAN is applied, the bug was always detected when access to freed memory occurred.

The last column of Table 12 presents the number of tested inputs during 180 seconds. After finding all the bugs, there was a continuous kernel panic and executions hardly increased. Therefore, we measured the number of executions in 180 seconds. As a result, BoKASAN performed 33% fewer executions than the baseline. Since BoKASAN detected one more UAF bug, additional overhead is incurred compared to baseline for handling this crash (e.g., reset VM). Considering this, the 33% fewer executions is not a big problem in real-world fuzzing where crashes are not frequently occurred.

On Windows, the baseline failed to detect Bug3, while BoKASAN detected all three bugs. To trigger Bug1 and Bug2, BoKASAN took approximately 12% longer than baseline, and for 180 seconds, BoKASAN performed approximately 40% of execution compared to baseline. In Windows, resetting the virtual machine takes longer than in Linux, and BoKASAN detects more panic than the baseline, hence, the total number of executions is lower than that of Linux. When fuzzing a kernel module that does not panic a lot, the difference in the number of executions would be smaller than this.

In both Ubuntu and Windows, BoKASAN incurred slightly higher overhead compared to baseline, but more importantly, BoKASAN detected the UAF bug which is hard to detect without address sanitizer. Through this, it was shown that applying BoKASAN helps fuzzer to detect memory errors in binary-only fuzzing.

6 Discussion

In this section, we discuss the limitations and future directions for BoKASAN.

Support for other OS. The functions required to implement BoKASAN are function and memory access instrumentation. Kernel function hooking methods are already widely used in Windows [7, 43, 58, 65] and macOS [5], and various hooking methods are publicly available [30]. In addition, because all general-purpose OSs have a page fault handler, we can trap memory access by hooking this handler function. Actually, a method to manipulate the page fault handler on a running OS is already used in the rootkit [55]. Therefore, BoKASAN can be applied to COTS OSs such as macOS and this would require additional engineering effort.

Building BoKASAN without any source code. In Linux, we can obtain the header files necessary to build the driver. We can also apply BoKASAN by building a standalone kernel module [28], extracting required function definitions from kernel binary, or hooking the necessary functions using binary rewriting (easier than instrumenting the entire kernel) without source code. In COTS OS, since they provide SDK for driver development, we can easily build the driver for BoKASAN.

Stack and global variable. KASAN sanitizes not only slab objects but also stack and global objects by performing compile-time instrumentation, but BoKASAN only targets bugs that occur in slab objects. The memory space for the stack frame and global variables is determined at compile time, thus sanitizing them using binary approaches still remains a great challenge [10, 12, 54]. For example, Retrowrite creates a limited red zone in frame granularity (not object granularity) in the stack, and QASAN does not create red zones in both global and stack. Among the bugs discovered by KASAN, UAF and slab OOB account for the highest percentage, whereas stack and global OOB form a relatively small number. For example, among the bugs reported to Syzbot that were fixed, 535 and 203 were UAF and slab OOB, and 70 and 19 of the latter were stack and global OOB. Therefore, we focused on detecting UAF and slab OOB bugs, leaving stack and global OOB bugs for future studies.

7 Related Work

Sanitizing for memory error detection. The C and C++ programming languages are still favored for low-level system software, such as operating system kernels and runtime libraries; however, these languages remain notoriously insecure to memory errors that make the software vulnerable to exploitation. Dynamic bug-finding tools known as sanitizer have been proposed for the detection and analysis of memory bugs regardless of related crashes [11, 16, 18, 27, 40, 51, 63, 66, 67]. Readers are referred to the systemization work of Song et al. [54] for the types of detectable vulnerabilities and the strengths and weaknesses of each sanitizer. Among the various sanitizing techniques, ASAN leverages the red-zone insertion technique to assist with the detection of memory bugs that accessed invalid address space, such as OOB and UAF [51]. The detection accuracy of ASAN is proportional to the size

of the red zones; but, as the red zones are enlarged to improve the accuracy, memory usage also increases. MEDS [16] addresses this problem using *infinite gap* and *infinite heap*. These two techniques enable MEDS to detect both spatial and temporal errors more accurately and efficiently than ASANs.

Unlike the above compile-time instrumentation methods, RetroWrite [10] implements the address sanitizer using static binary rewriting that adds the necessary instrumentation at the binary level as if it were inserted at compile time, actually outperforming Valgrind's binary-only Memcheck. Thus, without severely sacrificing the performance, memory errors can be detected without the source code of target applications. KRetroWrite [47] extends RetroWrite, which is applied to user-space programs only, for a kernel module. However, for the execution of the rewritten binary kernel module, KRetroWrite requires the kernel to be recompiled with KASAN, which means that the source code of the kernel still needs to be available. Unfortunately, not all kernels are provided as open-source, and even if it is an open-source kernel, the corresponding version of the kernel source is not necessarily available [9]. In contrast, BoKASAN does not require a KASAN-compiled kernel, which indicates the truly binary-only nature of KASAN.

Techniques for detecting uninitialized variables used by the kernel are also being studied [4, 21, 33, 41]. Bochspwn Reloaded [21] used kernel taint tracking on the Bochs emulator to detect kernel memory leakages into the user space. TimePlayer [4] is a proposed *differential replay* technique that can effectively detect the use of uninitialized variables. Although BoKASAN currently targets OOB and UAF bugs only, we plan to extend our study to determine the usage of uninitialized variables in the future by using a binary-only function and memory access instrumentation.

Kernel fuzzing. Syzkaller, which is the most commonly used kernel fuzzer in Linux, has discovered thousands of Linux kernel bugs, showing that kernel fuzzing is effective [59]. Syzkaller is currently being extended to support other OS kernels, and many other advanced kernel fuzzers are being developed based on Syzkaller. These fuzzers are successfully discovering a large number of kernel bugs [57]. For example, DIFUZE [9] and Agamotto [53] successfully targeted kernel drivers, Razzer [19], KRace [62] focused on race condition bugs, HFL [24] applied hybrid fuzzing, and SyzVegas [61] applied reinforcement learning.

Apart from the Linux kernel, several kernel fuzzers targeting COTS OSs such as macOS and Windows had been studied [3, 6, 13, 15, 31, 37, 44]. LynxFuzzer [37] and kAFL [49] leverage a hypervisor to collect code coverage and use it for coverage-guided fuzzing. BSOD [36] targets binary-only device drivers. However, because a sanitizer applicable to COTS OSs has not yet been developed, most fuzzers targeted COTS OSs without applying address sanitizer. BoKASAN would be able to find more bugs more effectively with existing kernel fuzzers in a COTS OS.

8 Conclusion

This paper presented BoKASAN, which is the first practical binary-only KASAN to realize the novel selective sanitization idea with sophisticated engineering effort on the page-fault handling mechanism. BoKASAN was effective in bug-finding at compiler-level performance, e.g., even comparable to KASAN on the Linux kernel, by significantly reducing the performance overhead in dynamic instrumentation for sanitizing the entire kernel binaries (RQ1, RQ2). We also showed that selective sanitization contributed to the performance gain of BoKASAN in the ablation experiment (RQ3). Furthermore, compared to emulation-based KASAN emulating all instructions, BoKASAN significantly reduced the overhead by only instrumenting memory accesses related to the sanitized objects. Finally, we showed that BoKASAN applies to binary-only driver fuzzing on Ubuntu and Windows, respectively (RQ4). The observed performance of BoKASAN renders great opportunities for kernel memory sanitization and fuzzing in future studies

Acknowledgments

We thank the anonymous reviewers and our shepherd for helpful comments and suggestions on this work. This work was supported in part by the Institute for Information and Communications Technology Planning and Evaluation (IITP) funded by the Government of Korea, Ministry of Science & Information Technology (MSIT), under Grant 2018-0-00513 (Machine Learning-Based Automation of Vulnerability Detection on Unix-Based Kernel) and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2019R1A2C1088802).

References

[1] ASSEMBLY, L. Linux system calls, 2022. http://linasm.sourceforge.net/docs/syscalls/index.php.

[2] BAI, X., XING, L., ZHENG, M., AND QU, F. idea: Static analysis on the security of apple kernel drivers. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2020), pp. 1185–1202.

[3] BEN-SIMON, N., AND ALON, Y. Bugs on the windshield: Fuzzing the windows kernel. https://research.checkpoint.com/2020/bugs-on-the-windshield-fuzzing-the-windows-kernel/, 2020.

[4] CAO, M., HOU, X., WANG, T., QU, H., ZHOU, Y., BAI, X., AND WANG, F. Different is good: Detecting the use of uninitialized variables through differential replay. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2019), pp. 1883–1897.

[5] CASE, A., AND RICHARD III, G. G. Advancing Mac OS X rootkit detection. *Digital Investigation 14* (2015), S25–S33.

[6] CHANG, O. Attacking the Windows NVIDIA driver. https://googleprojectzero.blogspot.com/2017/02/attacking-windows-nvidia-driver.html, 2017.

[7] CHOI, J., KIM, K., LEE, D., AND CHA, S. K. NTFuzz: Enabling type-aware kernel fuzzing on Windows with static binary analysis. In *Proc. the IEEE Symposium on Security and Privacy (SP)* (2021), IEEE, pp. 677–693.

[8] CHRIS HAMILTON, S. E. Testing Chromium: Syzyasan, a lightweight heap error detector. https://blog.chromium.org/2013/05/testing-chromium-syzyasan-lightweight.html, 2013.

[9] CORINA, J., MACHIRY, A., SALLS, C., SHOSHITAISHVILI, Y., HAO, S., KRUEGEL, C., AND VIGNA, G. Difuze: Interface aware fuzzing for kernel drivers. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017), pp. 2123–2138.

[10] DINESH, S., BUROW, N., XU, D., AND PAYER, M. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *Proc. the IEEE Symposium on Security and Privacy (SP)* (2020), pp. 128–142.

[11] DU, Y., NING, Z., XU, J., WANG, Z., LIN, Y.-H., ZHANG, F., XING, X., AND MAO, B. HART: Hardware-assisted kernel module tracing on Arm. In *Proc. the European Symposium on Research in Computer Security (ESORICS)* (2020), Springer, pp. 316–337.

[12] FIORALDI, A., D'ELIA, D. C., AND QUERZONI, L. Fuzzing binaries for memory safety errors with QASan. In *IEEE Secure Development (SecDev)* (2020), IEEE, pp. 23–30.

[13] FLANKER. The python bites your apple fuzzing and exploiting osx kernel bugs. https://papers.put.as/papers/macosx/2016/xkungfoo.pdf, 2016.

[14] GOSWAMI, S. An introduction to kprobes. https://lwn.net/Articles/132196/, 2005.

[15] HAN, H., AND CHA, S. K. IMF: Inferred model-based fuzzer. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017), pp. 2345–2358.

[16] HAN, W., JOE, B., LEE, B., SONG, C., AND SHIN, I. Enhancing memory error detection for large-scale applications and fuzz testing. In *Proc. the Network and Distributed Systems Security Symposium (NDSS)* (2018).

[17] ILAMMY. Ftrace-hook. https://github.com/ilammy/ftrace-hook, 2018.

[18] JEON, Y., HAN, W., BUROW, N., AND PAYER, M. FuZZan: Efficient sanitizer metadata design for fuzzing. In *Proc. the USENIX Annual Technical Conference (USENIX ATC)* (2020), pp. 249–263.

[19] JEONG, D. R., KIM, K., SHIVAKUMAR, B., LEE, B., AND SHIN, I. Razzer: Finding kernel race bugs through fuzzing. In *Proc. the IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 754–768.

[20] JSHERMAN212. xnuspy, 2021. https://github.com/jsherman212/xnuspy.

[21] JURCZYK, M. Detecting kernel memory disclosure with x86 emulation and taint tracking, 2018.

[22] KERNEL, L. Complete virtual memory map with 4-level page tables, 2021. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.

[23] KERNEL, T. L. Kfence, 2022. https://docs.kernel.org/dev-tools/kfence.html.

[24] KIM, K., JEONG, D. R., KIM, C. H., JANG, Y., SHIN, I., AND LEE, B. HFL: Hybrid fuzzing on the Linux kernel. In *Proc. of the Annual Network and Distributed System Security Symposium (NDSS)* (2020).

[25] KLEES, G., RUEF, A., COOPER, B., WEI, S., AND HICKS, M. Evaluating fuzz testing. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018), pp. 2123–2138.

[26] LAWTON, K. P. Bochs: A portable pc emulator for unix/x. *Linux Journal 1996*, 29es (1996), 7–es.

[27] LETTNER, J., SONG, D., PARK, T., LARSEN, P., VOLCKAERT, S., AND FRANZ, M. PartiSan: Fast and flexible sanitization via run-time partitioning. In *Proc. the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)* (2018), Springer, pp. 403–422.

[28] LEVINZON, E. A standalone linux kernel module, 2020. https://itnext.io/a-standalone-linux-kernel-module-df54283d4803.

[29] LLVM. Gwpasan, 2022. https://llvm.org/docs/GwpAsan.html.

[30] LOPEZ, J., BABUN, L., AKSU, H., AND ULUAGAC, A. S. A survey on function and system call hooking approaches. *Journal of Hardware and Systems Security 1*, 2 (2017), 114–136.

[31] LOUREIRO, J., AND GESHEV, G. Platform agnostic kernel fuzzing. https://labs.f-secure.com/archive/platform-agnostic-kernel-fuzzing/, 2016.

[32] LU, K. Analysis: Inspecting mach messages in macos kernel-mode part i: Sniffing the sent mach messages, 2018. https://www.fortinet.com/blog/threat-research/inspecting-mach-messages-in-macos-kernel-mode--part-i--sniffing-.

[33] LU, K., SONG, C., KIM, T., AND LEE, W. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), pp. 920–932.

[34] LU, S., KUANG, X., NIE, Y., AND LIN, Z. A hybrid interface recovery method for Android kernels fuzzing. In *Proc. the IEEE International Conference on Software Quality, Reliability and Security (QRS)* (2020), IEEE, pp. 335–346.

[35] MAIER, D., RADTKE, B., AND HARREN, B. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *Proc. the USENIX Workshop on Offensive Technologies (WOOT)* (2019).

[36] MAIER, D., AND TOEPFER, F. BSOD: Binary-only Scalable fuzzing Of device Drivers. In *International Symposium on Research in Attacks, Intrusions and Defenses* (2021), pp. 48–61.

[37] MAZZONE, S. B., PAGNOZZI, M., FATTORI, A., REINA, A., LANZI, A., AND BRUSCHI, D. Improving mac os x security through gray box fuzzing technique. In *Proc. the European Workshop on System Security* (2014), pp. 1–6.

[38] MCVOY, L. W., STAELIN, C., ET AL. lmbench: Portable tools for performance analysis. In *Proc. the USENIX Annual Technical Conference (USENIX ATC)* (1996), San Diego, CA, USA, pp. 279–294.

[39] NCCGROUP. Triforceafl: Afl/qemu fuzzing with full-system emulation, 2019. https://github.com/nccgroup/TriforceAFL.

[40] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices 42*, 6 (2007), 89–100.

[41] ÖSTERLUND, S., KONING, K., OLIVIER, P., BARBALACE, A., BOS, H., AND GIUFFRIDA, C. kMVX: detecting kernel information leaks with multi-variant execution. In *Proc. the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019), pp. 559–572.

[42] PAWLOWSKI, A., CONTAG, M., VAN DER VEEN, V., OUWEHAND, C., HOLZ, T., BOS, H., ATHANASOPOULOS, E., AND GIUFFRIDA, C. MARX: Uncovering class hierarchies in C++ programs. In *Proc. of the Annual Network and Distributed System Security Symposium (NDSS)* (2017).

[43] PETRONI JR, N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proc. the ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 103–115.

[44] PLASKETT, A., AND LOUREIRO, J. Biting the apple that feeds you - macos kernel fuzzing. https://labs.f-secure.com/archive/biting-the-apple-that-feeds-you-macos-kernel-fuzzing/, 2017.

[45] RAJAGOPALAN, M., PERINAYAGAM, S., HE, H., ANDREWS, G., AND DEBRAY, S. Binary rewriting of an operating system kernel. In *Proc. the Workshop on Binary Instrumentation and Applications* (2006).

[46] REVJAY. Vergilius project, 2022. https://www.vergiliusproject.com/.

[47] RIZZO, M., DE LA PRAZ, C., AND VOISIN, J. Hardening and testing privileged code through binary rewriting.

[48] SCHMIDT, A., POLZE, A., AND PROBERT, D. Teaching operating systems: Windows kernel projects. In *Proc. the ACM technical symposium on Computer science education* (2010), pp. 490–494.

[49] SCHUMILO, S., ASCHERMANN, C., GAWLIK, R., SCHINZEL, S., AND HOLZ, T. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proc. the USENIX Security Symposium (SEC)* (2017), pp. 167–182.

[50] SCHWARTZ, E. J., COHEN, C. F., DUGGAN, M., GENNARI, J., HAVRILLA, J. S., AND HINES, C. Using logic programming to recover C++ classes and methods from compiled executables. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018), pp. 426–441.

[51] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *Proc. the USENIX Annual Technical Conference (USENIX ATC)* (2012), pp. 309–318.

[52] SONG, D., HETZELT, F., DAS, D., SPENSKY, C., NA, Y., VOLCKAERT, S., VIGNA, G., KRUEGEL, C., SEIFERT, J.-P., AND FRANZ, M. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Proc. the Network and Distributed Systems Security Symposium (NDSS)* (2019).

[53] SONG, D., HETZELT, F., KIM, J., KANG, B. B., SEIFERT, J.-P., AND FRANZ, M. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proc. the USENIX Security Symposium (SEC)* (2020), pp. 2541–2557.

[54] SONG, D., LETTNER, J., RAJASEKARAN, P., NA, Y., VOLCKAERT, S., LARSEN, P., AND FRANZ, M. SoK: Sanitizing for security. In *Proc. the IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1275–1295.

[55] SPARKS, S., AND BUTLER, J. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan 11*, 63 (2005), 504–533.

[56] SYZBOT. syzbot. https://syzkaller.appspot.com/upstream, 2021.

[57] SYZKALLER. Research work based on syzkaller. https://github.com/google/syzkaller/blob/master/docs/research.md, 2021.

[58] VOGL, S., GAWLIK, R., GARMANY, B., KITTEL, T., PFOH, J., ECKERT, C., AND HOLZ, T. Dynamic hooks: hiding control flow changes within non-control data. In *Proc. the USENIX Security Symposium (SEC)* (2014), pp. 813–328.

[59] VYUKOV, D. Syzkaller, 2015.

[60] VYUKOV, D. syzbot and the tale of thousand kernel bugs. *Linux Security Summit* (2018).

[61] WANG, D., ZHANG, Z., ZHANG, H., QIAN, Z., KRISHNAMURTHY, S. V., AND ABU-GHAZALEH, N. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *Proc. the USENIX Security Symposium (SEC)* (2021), pp. 2741–2758.

[62] XU, M., KASHYAP, S., ZHAO, H., AND KIM, T. Krace: Data race fuzzing for kernel file systems. In *Proc. the IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 1643–1660.

[63] XU, M., LU, K., KIM, T., AND LEE, W. Bunshin: Compositing security mechanisms through diversification. In *USENIX Annual Technical Conference (USENIX ATC)* (2017), pp. 271–283.

[64] XU, W., MOON, H., KASHYAP, S., TSENG, P.-N., AND KIM, T. Fuzzing file systems via two-dimensional input space exploration. In *Proc. the IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 818–834.

[65] YIN, H., LIANG, Z., AND SONG, D. HookFinder: Identifying and understanding malware hooking behaviors. In *Proc. the Network and Distributed Systems Security Symposium (NDSS)* (2008).

[66] ZHANG, J., WANG, S., RIGGER, M., HE, P., AND SU, Z. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs. In *Proc. the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2021), pp. 479–494.

[67] ZHANG, Y., PANG, C., PORTOKALIDIS, G., TRIANDOPOULOS, N., AND XU, J. Debloating address sanitizer. In *Proc. the USENIX Security Symposium (SEC)* (2022).

A Applying BoKASAN to Windows Kernel

Since BoKASAN utilizes the functions of the OS kernel such as paging, it can be applied to operating systems other than Linux. To implement BoKASAN, we should be able to do the followings: 1) select target process using PID, 2) hook necessary functions, 3) instrument memory access through page fault mechanism, and 4) manage shadow memory.

Unlike Linux, since the source code of Windows is not publicly released, it is more difficult to identify and modify the kernel function compared to Linux. Fortunately, we were able to implement BoKASAN on Windows with minimal engineering work required owing to two projects: Windows research kernel (WRK) [48] and Vergilius project [46]. WRK provides x86/x64 Windows Server 2003 SP1 kernel source code for teaching and research purposes. Although it is a previous version of Windows, the core of the kernel operates similarly to Windows 10, so it could be used as a reference for analyzing Windows 10. The Vergilius project provides information on various structures used in the Windows kernel, so we were able to obtain the structure information necessary for BoKASAN implementation.

Selective sanitization. Kernel fuzzing in Windows is also performed by calling the system call in the fuzzer process as in Linux. Therefore, we register the PID of the fuzzer process through IOCTL to BoKASAN for selective sanitization. In Windows, the PID can be obtained by calling the `GetCurrentProcessId` in the user process and the `PsGetCurrentProcessId` in the kernel process.

Function instrumentation. We instrument functions through inline hooking. Inline hooking is a method of inserting instructions by modifying an instruction at the beginning of functions to a jump. The original instruction is copied to the trampoline buffer, and the instruction that jumps to the hooking handler replaces it. Through this, when the target function is executed, the hooking handler of the target function is executed, and in the handler, the original function can be executed by calling the trampoline address. In Windows, BoKASAN hooks the following four functions.

- `nt!ExAllocatePoolWithTag` – A allocation function used in Windows. BoKASAN creates a red zone and clears the present bit when the request process is a target process. In Windows, BoKASAN allocates objects with page-aligned size including a red zone.
- `nt!ExFreePoolWithTag` – A deallocation function used in Windows. When a target object is requested to free, BoKASAN does not actually deallocate and updates shadow memory to BOKASAN_FREE.
- `nt!MmAccessFault` – A page fault handler in Windows. BoKASAN sets the page present bit of the object and the trap bit of the *eflags* register when the fault address is a target object (i.e., shadow value is BOKASAN_OBJECT).
- `nt!KiTrap01` – A debug exception handler in Windows. It is executed when single-step triggering. BoKASAN clears the present bit in this function.

Memory access instrumentation. BoKASAN utilizes the page fault mechanism in Windows as described in §3.3 to instrument memory access. To do this, we hook `nt!MmAccessFault` and `nt!KiTrap01` described above as page fault handler and debug exception handler.

Shadow memory. Windows does not provide a kernel API that allocates memory in a fixed area, so we use a dynamic

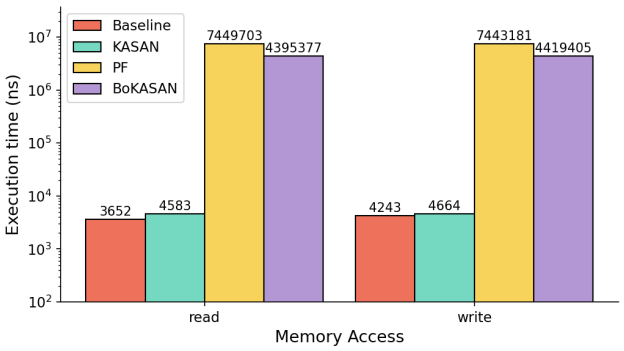


Figure 8: Time to access memory for 1,000 times. (Update of Fig. 2) PF uses a Linux handler while BoKASAN uses our own implementation. BoKASAN further overcomes this overhead by handling memory access of target processes only (e.g., 3.5%) for fuzzing.

shadow memory by utilizing a lookup table. For this, if shadow memory has not been allocated when an object is allocated, we use the `nt!ExAllocatePoolWithTag` with the `NonPagedPool` flag to allocate the shadow memory. The shadow memory is allocated in 2k granularity corresponding to 16 pages memory area. In 32 bits Windows, an address in the range of 0x80000000–0xffffffff is reserved for the kernel, so a maximum of 0x8000 (0x80000000/0x10000) shadow memory fragments are used.

B Page-level Sanitized Memory Region

Implementation of the sanitized memory region introduced in §2.4 requires the memory region in which objects are allocated to be separated into the memory allocation and the sanitized memory regions. However, to the best of our knowledge, a method that allocates memory objects to separated memory areas without changing the source code of the Linux kernel is not yet known to exist. The separation of memory regions requires sanitized allocation to always allocate objects to the sanitized region, but Linux does not provide a function to do this. Therefore, BoKASAN applied a *page-level sanitized memory region*.

The *page-level sanitized memory region* only sanitizes memory pages that contain sanitized objects. This situation allows sanitized and non-sanitized pages to coexist in the same memory area. Because the present bit of the sanitized page is cleared, a page fault occurs only when memory access occurs on the sanitized page. Non-sanitized pages have the present bit set, thus page faults do not occur because of sanitizer operation. When a process other than the target process requests memory allocation, BoKASAN checks the *PAGE_SPECIAL* flag of the page to which the object is allocated to determine whether the page is sanitized. If it is a sanitized page, BoKASAN re-requests memory allocation to enable the requested object to be allocated to a non-sanitized page. As a result, although this is not a separate memory region, page faults occur only on the sanitized page, resulting in the number of page faults that occur being almost the same as the number of when sanitized memory regions were used.

Table 13: Number of checking memory access in 1h fuzzing

	Mean	Median
KASAN	4,278,871,654	4,977,754,112
BoKASAN	151,354,982	157,100,032
BoKASAN / KASAN	3.5%	3.2%

C Micro Evaluation of BoKASAN

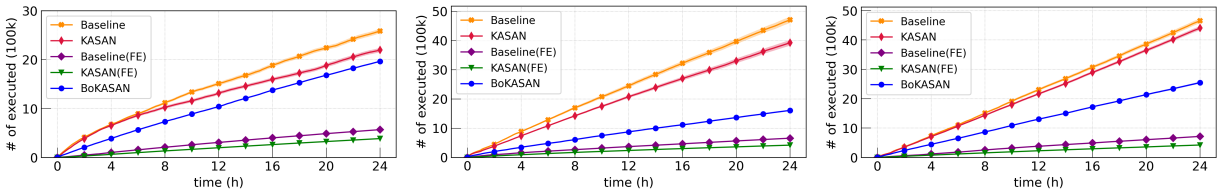
Table 13 lists the number of memory accesses that occurred for 1h of fuzzing. The number of memory accesses of KASAN is measured in the *check_memory_region_inline* that checks the shadow memory, and that of BoKASAN was measured in the hooked page fault handler. Note that we only counted the page fault caused by BoKASAN. As can be seen from Table 13, BoKASAN checked only approximately 3.5% of memory accesses compared to KASAN. Consequently, BoKASAN minimizes the overhead incurred by page faults by checking only a tiny number of memory accesses.

Figure 8 shows the elapsed time when read or write memory 1,000 times in Linux kernel v4.19. It can be seen that BoKASAN incurs 1042 and 948 times overhead compared to Baseline and KASAN in memory write due to overhead incurred by page fault as described in §2. Although the page fault incurs a large overhead, such overhead incurs only for the memory access instruction among all executed instructions, and because the memory access instruction occupies only a part of the total executed instruction, the entire overhead in fuzzing appears smaller than this. Moreover, as described above, among these memory access instructions, only 3.5% of the memory accesses compared to KASAN are instrumented by BoKASAN, hence, the BoKASAN incurs KASAN-level overhead in fuzzing.

PF is the page fault overhead without checking memory access validity using the default handler of Linux to handle debug exception. BoKASAN minimizes the overhead occurring in the debug handler by performing the minimum instruction for handling a single-step in the debug exception handler.

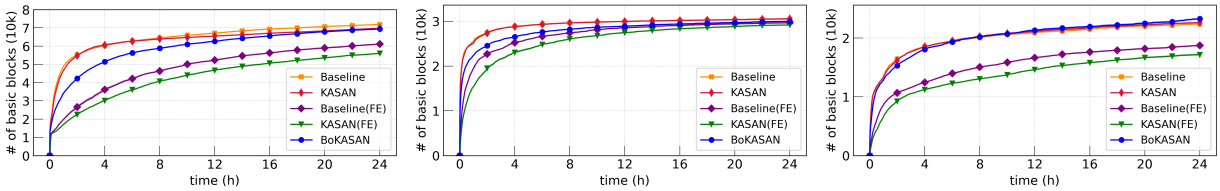
D Fuzzing Results

Figure 9 shows the number of syscalls executed as a result of 24h fuzzing for all, file system and network-related syscalls. When file system and network-related syscalls were targeted, BoKASAN executed approximately 34% and 58% of syscalls compared to KASAN, respectively, and executed 4.3 and 6.1



(a) all ($p < 10^{-19}$) (b) file system ($p < 10^{-40}$) (c) network ($p < 10^{-40}$)

Figure 9: Number of syscalls execution during 24h fuzzing. We state the p values with statistical tests of KASAN vs. BoKASAN.



(a) all ($p < 10^{-8}$) (b) file system ($p < 10^{-22}$) (c) network ($p < 10^{-8}$)

Figure 10: Number of discovered basic blocks during 24h fuzzing.

times more syscalls than KASAN (FE), respectively. Figure 10 shows the number of discovered basic blocks as a result of fuzzing. When file system and network-related syscall were targeted, BoKASAN discovered 2% lesser and 3% more basic blocks compared to KASAN, respectively, and discovered 3% and 35% more basic blocks compared to KASAN (FE), respectively. This shows that BoKASAN can perform binary-only kernel fuzzing much more effectively than full emulation-based KASAN for all categories of syscalls. Additionally, we used the Mann-Whitney U-test according to the guidelines in [25] and noted the p-values for 20 runs of KASAN and BoKASAN in the parentheses in Figures 9 and 10. Every p-value was significantly lower than 0.001, indicating statistical confidence in our experimental results.

E Binary-only Fuzzing Dataset

Listing 1 shows the code of the vulnerable function included in the target driver used in our Windows experiment. The original version of the Windows driver does not have the "KASAN" bug, but to evaluate the performance of BoKASAN's bug detection, the third bug was added to be the same as the Linux version. The first and second bugs, if the input value satisfies the string in the if statement, result in a kernel panic due to a null pointer dereference (lines 22 and 29), which can be detected even without a sanitizer. On the other hand, the third bug writes data to the memory address that has been freed by the `ExFreePool` function (lines 35-36) and immediately returns. In this case, if another object is not allocated in the freed memory address, the kernel panic does not occur and without the help of the sanitizer, the fuzzer cannot detect the bug even if it is triggered.

```
1  NTSTATUS crashMe(IN PIO_STACK_LOCATION IrpStack) {
2    SIZE_T size = 0;
3    PCHAR userBuffer = NULL;
4    int* array = (int*)ExAllocatePoolWithTag(NonPagedPool
         , 1332, 'tset');
5
6    userBuffer = (PCHAR) IrpStack->Parameters.
         DeviceIoControl.Type3InputBuffer;
7    size = IrpStack->Parameters.DeviceIoControl.
         InputBufferLength;
8
9    if (size < 0xe) {
10     return STATUS_SUCCESS;
11   }
12
13   if (userBuffer[0] == 'K')
14    if (userBuffer[1] == 'E')
15     if (userBuffer[2] == 'R')
16      if (userBuffer[3] == 'N')
17       if (userBuffer[4] == 'E')
18        if (userBuffer[5] == 'L')
19         if (userBuffer[6] == 'A')
20          if (userBuffer[7] == 'F')
21           if (userBuffer[8] == 'L')
22             ((VOID(*)())0x0)();
23   if (userBuffer[0] == 'S')
24    if (userBuffer[1] == 'E')
25     if (userBuffer[2] == 'R')
26      if (userBuffer[3] == 'G')
27       if (userBuffer[4] == 'E')
28        if (userBuffer[5] == 'J')
29          size = *((PSIZE_T)(0x0));
30   if (userBuffer[0] == 'K')
31    if (userBuffer[1] == 'A')
32     if (userBuffer[2] == 'S')
33      if (userBuffer[3] == 'A')
34       if (userBuffer[4] == 'N') {
35         ExFreePool(array);
36         array[0] = 1234;
37         return STATUS_SUCCESS;
38       }
39
40   ExFreePool(array);
41
42   return STATUS_SUCCESS;
43 }
```

Listing 1: Vulnerable function of Windows driver.