PET: Prevent Discovered Errors from Being Triggered in the Linux Kernel

Zicheng Wang*
wzc@smail.nju.edu.cn
Nanjing University

Yueqi Chen
yueqi.chen@colorado.edu
University of Colorado Boulder

Qingkai Zeng
zqk@nju.edu.cn
Nanjing University

Abstract

The Linux kernel is the backbone of modern society. When a kernel error is discovered, a quick remediation is needed. Whereas sanitizers greatly facilitate root cause diagnosis, fixing errors takes a long time, resulting in errors discovered but still exploited. In this work, we propose PET, a temporary solution to prevent discovered errors from being triggered and exploited before patches are available.

Technically, PET takes a sanitizer report as the input, constructing the triggering condition that can be evaluated at runtime. If the condition is met, PET takes a series of actions to prevent error triggering. PET is designed to be extensible to various error types. In our experiment, we demonstrated its effectiveness against the five most common errors that state-of-the-art sanitizers can report. PET is lightweight with performance overhead less than 3%. Further, PET is scalable in the presence of multiple errors with acceptable memory assumption. The kernel has run stably for more than 3 months under intensive use after errors are prevented.

1 Introduction

It is often stated that modern civilization runs on Linux, because it is widely used in cloud servers, mobile phones, transportation systems, and even nuclear plants [1, 2]. However, the Linux kernel is sophisticated and error-prone, as evidenced by the numerous bugs discovered by various automated tools (*e.g.*, [3–13]) and reported exploits (*e.g.*, [14–21]).

To fix these newly discovered errors or 0-day exploits caught in the wild, security analysts need to diagnose their root cause and develop patches to be merged to the upstream kernel. Sanitizers [22–25] like Kernel Address Sanitizer (KASAN) [26], Kernel Memory Sanitizer (KMSAN) [27], and Kernel Concurrency Sanitizer (KCSAN) [28] provide a crucial advantage, as they allow security analysts to reproduce the errors and obtain reports containing in-depth information about the error. The use of sanitizer reports helps to streamline the error debugging process and enables more efficient patch development. The success of sanitizers in the Linux kernel has led to their adoption on Microsoft platforms [29].

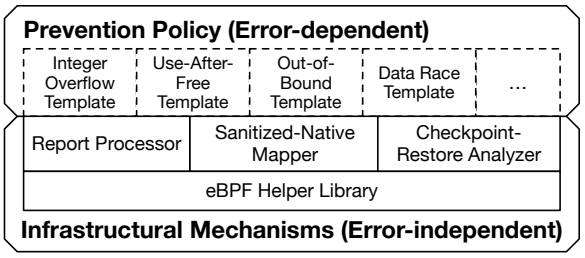

Figure 1: Two-layer architecture. The underlying layer is error-independent, providing a set of infrastructural mechanisms. The overlaying customizes prevention policies for different error types.

However, quickly remediating these errors remains challenging for the security community. Due to a lack of manpower and the time-consuming nature of diagnosis which requires specialized expertise, currently, fixing kernel vulnerabilities takes an average of 66 days [30]. During this time window, adversaries can craft Proof-of-Concept (PoC) programs or exploits to compromise the kernel or even escalate privilege to steal sensitive data. It has been reported multiple times that despite being discovered, errors are still exploited [31–33].

In this work, we design and implement PET, a solution that prevents discovered errors (including both memory errors and their root causes) from being triggered and exploited in the Linux kernel before patches are released. Unlike prior works (*e.g.*, [34–38]) that aim to precisely diagnose the root cause of the error from the report, PET takes a sanitizer report as the input and constructs the triggering condition of the corresponding error to be evaluated at runtime. If the condition is met, PET will take a series of actions including skipping the error sites. Since the error cannot be triggered, it is impossible to exploit it.

*The work was done while visiting the University of Colorado Boulder.

Technically, PET is designed to be extensible with two layers as shown in Figure 1. The overlaying layer (§5) is error-dependent, customizing prevention policies for different error types. These policies are expressed in eBPF program templates where the specifications of the error details can be filled in. The underlying layer (§6) is error-independent, providing a set of infrastructural mechanisms including ❶ a report processor which offers a uniform interface to extract critical information from sanitizer reports, ❷ a sanitized-native mapper which translates critical error information like the error site from sanitized kernel image to native kernel image that needs protection, ❸ a checkpoint-restore analyzer which attempts to return the kernel to normal states after skipping the error sites, ❹ a library of commonly used eBPF helper functions.

PET currently is shipped with prevention policies for the five most common errors that state-of-the-art sanitizers can report. They are integer underflow/overflow which accounts for all 68 Undefined Behavior Sanitizer (UBSAN) reports, out-of-bounds access, and use-after-free, making up 94% (995/1059) of KASAN reports, uninitialized access, accounting for all 295 KMSAN reports, and data race, which covers all 123 KCSAN reports. These errors are widely recognized to be exploitable and according to our extensive collection, 75.4% (184/244) public exploits or writeups targeting the Linux kernel over the past decade focus on these errors. Further, Microsoft reports that around 70% software vulnerabilities addressed every year pertain to those memory safety issues [39]. Our evaluation on v5.15 shows the effectiveness of PET in preventing these error types. In the future, PET can be further extended to support more types as new sanitizers are designed and released.

PET is lightweight with a performance overhead less than 3%. Further, PET is scalable in the presence of multiple errors. The largest amount of extra memory required by PET is 915 MB. We argue it is not a concern for modern OS kernel because the kernel has full access to physical memory which is typically more than 16GB nowadays.

In comparison with the prior work [30], which equips the kernel with the capability of undoing the error effect through instrumentation and recompilation, PET can be enabled at runtime without rebooting the system and interrupting kernel execution. In addition, as an end-to-end system, PET can automatically generate error-prevention immediately upon the vulnerability discovery while [30] requires human assistance for such s process.

PET is a temporary solution designed to prevent errors before the official patches are released. While empirical evaluation shows that after PET successfully prevent all errors in our testcase set, the kernel runs stably for over 3 months which is much longer than the typical 66 days time window for patching, the state recovery capabilities of PET have certain limitations, as we will discuss in §6.3. Therefore, it is recommended to apply official patches once they are available. In summary, this work makes the following contributions.

- We designed PET to prevent discovered errors from being triggered and exploited before official patches are released.
- We illustrated the customization of prevention policies using five commonly reported errors by state-of-the-art sanitizers.
- We implemented PET in the Linux kernel and thoroughly evaluated its effectiveness and performance by using real-world errors.

In the following, we first introduce the background in §2 and outlines the threat model and assumptions in §3, followed by the workflow in §4. We explore the error-dependent overlaying layer in §5 and detail the error-independent underlying layer in §6. §7 discusses our implementation. §8 evaluates the effectiveness and overhead of PET. §9 reviews related works. §10 provides future work and discussion, followed by the conclusion in §11.

2 Background

In this section, we provide background knowledge that is essential for understanding PET. First, we present the memory of the Linux kernel, common memory errors, and the kernel sanitizers for testing and debugging. Afterward, we delve into the features and capabilities of eBPF mechanisms.

2.1 Linux Kernel Memory and Errors

Data objects stored in different kernel memory regions have their own lifecycle and boundary.

Global & Static Regions. Kernel data objects in global and static regions (*e.g.*, `.data`, `.rodata` sections) are kept alive from kernel bootup to kernel shutdown. They are automatically initialized to a specific value or zeroed during compilation. The size of global and static objects is also pre-defined before they are loaded into the physical memory. Legal access is only allowed within the boundary.

Kernel Stacks. The kernel has a stack for each userspace process which is used when system calls are invoked, and an interrupt task per CPU to handle external interrupts. The life cycle and boundaries of objects in both stacks follow the same principle. In terms of life cycle, when a kernel function is called, its stack frame is created and stack objects come to life after the prologue. After the epilogue, the stack frame is destroyed, making the stack objects dead. During this life cycle, stack objects must be initialized before they are read.

In terms of boundaries, while the C99 standard introduces the variable-length array (VLA) feature, which allows the length of stack arrays to be determined at runtime, the Linux

kernel has discarded this feature since v4.20 for better security and lower overhead [40]. Therefore, the size of all stack objects is pre-defined at compilation time and access to them should never exceed the boundaries.

Kernel Heaps. The kernel heap is comprised of three memory regions managed by Buddy System, SLAB/SLUB allocator, and vmalloc allocator, respectively. Buddy System manages objects that are larger than one page. The SLAB/SLUB allocator obtains physically contiguous pages from Buddy System and uses them to store small objects. These pages are known as slab cache and are divided into slots with fixed sizes. The vmalloc allocator is also a customer of Buddy System, but the memory it manages is virtually contiguous.

The life cycle of heap objects begins and ends when an allocation interface and a deallocation interface are explicitly called. The three allocators have their own (de)allocation interfaces, such as `alloc_pages`-series and `free_pages`-series, `kmalloc`-series and `vmalloc`-series. The memory after deallocation will be reclaimed for future reuse. Similar to stack objects, heap objects must be initialized before they are read, but the size of heap objects can be either pre-defined at compile time or dynamically determined at runtime (*e.g.*, elastic object [17]). Once the size of a heap object is determined, access to it should never go beyond the boundary.

Memory Errors and Root Causes. A memory error occurs when either the life cycle or the boundary of a kernel object is broken and thus be categorized as a temporal error or a spatial error.

Use-after-free and its special instance, double-free, refer to the situation when a stack or heap object is accessed through a dangling pointer after its life cycle has ended.[1] Uninitialized access occurs when a stack or heap object is read before it has been initialized. Kernel objects in global and static regions are exempt from temporal errors as they are alive from kernel bootup and are automatically initialized by the compiler. Out-of-bound access can result from integer overflow, type confusion [41, 42], data race, and many other root causes. Since a buffer is not a necessity for such error, we use the term "out-of-bound access" to describe all situations where the read or write of a kernel object exceeds its legitimate boundary.

Exploitability of Kernel Memory Errors. We extensively collected in total 244 public exploits and writeups targeting the Linux kernel over the past decade, sourced from Github, BlackHat, BlueHat, Pwn2Own, and personal blogs (*e.g.*, [43]). The statistics show that 75.4% (184/244) of them exploit memory errors described above. To elaborate, these include 5.7% for integer underflow/overflow, 26.3% for out-of-bounds access, 37.3% for use-after-free, 4.9% for uninitialized access, and 1.2% for data race.[2] These errors can be detected by sanitizers and covered by `PET`.

2.2 Kernel Sanitizers

A variety of kernel sanitizers have been developed to detect memory errors and their root causes. The sanitizers are not intended for protection, but rather for testing and debugging, because they rely on shadow memory and heavy instrumentation which significantly slow down the kernel execution. When detecting memory errors, sanitizers will generate informative bug reports for further analysis. In this work, we leverage the information contained in the report to prevent attackers from triggering and exploiting the reported errors.

Sanitizers can be grouped into two categories: One is those that directly detect memory errors, such as Kernel Address Sanitizer (KASAN) for use-after-free and out-of-bound access and Kernel Memory Sanitizer (KMSAN) for uninitialized access. Another is those that detect root causes of memory errors, such as Undefined Behavior Sanitizer (UBSAN) [44] which can detect integer overflows before the integer is used as the buffer index and cause out-of-bound access, Kernel Thread Sanitizer (KTSAN) [45] and Kernel Concurrency Sanitizer (KCSAN) - both for data race with KCSAN being more widely used due to its simplicity[46].

2.3 The eBPF Ecosystem

Extended Berkeley Packet Filter (eBPF) is a Linux subsystem that allows safely executing untrusted user-defined extensions inside the kernel. The extension - eBPF program is installed to the kernel by attaching to specific instructions. These instructions are overwritten as either `int3` with the eBPF program registered in the interrupt handler or `jmp/call` which directly diverts the execution to the eBPF JIT engine. As such, eBPF programs can be attached to any kernel instruction at runtime, executed either before or after it.

eBPF programs are stateful across system calls with data stored in a kernel data structure named BPF maps. A BPF map is a set of key-value pairs. By specifying the type of key and value, the eBPF program developer can store arbitrary data. Holding a key, the eBPF program can look up and update the corresponding value. Therefore, BPF maps are usually used as a communication channel among eBPF programs and userspace applications. To enhance the expressiveness of eBPF programs, the eBPF subsystem provides a set of helper functions as the interface between eBPF programs and other kernel components. The eBPF program calls helper functions to interact with the kernel, getting additional information such as the elapsed time since kernel bootup and the PID of the current process.

[1] Though use-after-scope is the term for stack objects, here, we uniformly use use-after-free for simplicity.

[2] If one case, for example, is integer overflow caused by data race and finally results in out-of-bound access, we classify it as data race - the root cause, to avoid duplication

In recent years, eBPF ecosystem is rapidly evolving to be deployed to accelerate userspace services [47–49], as well as monitor and change kernel behaviors [50–54]. In this work, we employ eBPF ecosystem to design our run-time protection because it has the following advantages compared with other Linux tracing systems such as kprobes [55]: (1) safety - the Linux kernel has a static verifier to ensure its safety before loading. (2) expressive not only because of helper functions and BPF maps but also because of a mature toolchain consisting of a compiler that translates the eBPF bytecode to the native CPU instruction set. (3) only privileged users like system administrators can install and uninstall eBPF programs to the kernel. Therefore, attackers - unprivileged users are prevented from directly disabling our protection. In §10, we will discuss alternative implementations that might also work for our design.

3 Threat Model and Assumptions

Attacker-wise. Attackers can have PoC programs or functioning exploits for errors in any kernel subsystem, including the eBPF subsystem. These could be 0-day errors that have not been publicly disclosed but detected by the blue team, or n-day errors that have been disclosed but not yet patched. By executing the PoC or exploit, the attacker can trigger the error to panic the kernel or perform exploitation to escalate privilege and steal sensitive data.

The attacker is aware of the presence of `PET` on the victim's machine. They can indirectly influence the execution of `PET` through the use of system calls or sending network packets. However, as an unprivileged user, the attacker cannot directly disable `PET` by uninstalling eBPF programs, which requires privileged access.

Defender-wise. Defenders (*i.e.*, users of `PET`) are system administrators or blue teams who have the necessary privileges to access the debugging information and configuration files of the vulnerable kernel image, as well as the ability to install eBPF programs into the kernel space to deploy `PET`. We assume the eBPF programs are bug-free based on the fact that the Linux kernel performs a static safety check on all installed eBPF programs. This assumption doesn't contradict the assumption that the eBPF subsystem is vulnerable as a part of the kernel.

Defenders are aware of the error and have a report generated by the sanitizer. For 0-day exploits found in the wild, defenders can obtain their reports by reproducing them on a sanitized kernel. For n-day vulnerabilities, a significant portion of Linux kernel vulnerabilities are reported by Syzkaller [56] which generates and releases sanitizer reports upon discovering errors. For errors detected through static analysis, the defenders can manually verify them using the sanitized kernel to generate the report. Note that the report information may not always be accurate, such as benign

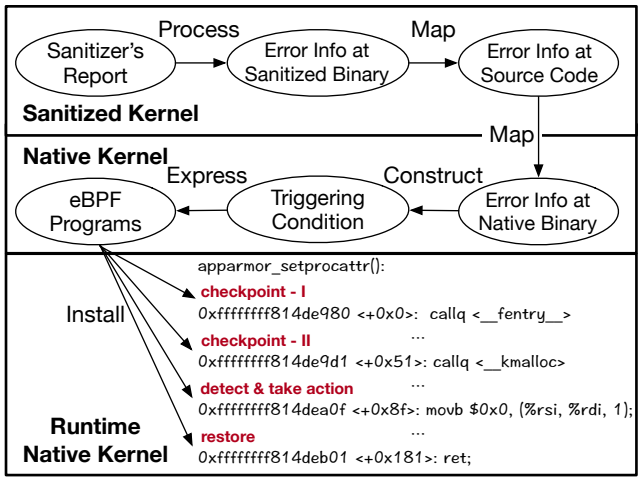

Figure 2: The workflow of `PET` consists of three major phases, as described in detail in §4.

```
1 KASAN: slab-out-of-bounds in
      ↪ apparmor_setprocattr+0x116/0x590
2 Write of size 1 at addr ffff888007449c80
3 Call Trace:
4    apparmor_setprocattr+0x116/0x590
5    proc_pid_attr_write+0x15f/0x1e0
```

Listing 1: The KASAN report snippet for the out-of-bound access vulnerability CVE-2016-6187 migrated and reproduced in v5.15.

data races being reported by KCSAN and KTSAN). To address this, defenders are encouraged to provide corrections using the methods discussed in previous research works [57]. Given that a single report only manifests one behavior of the error, defenders can use the techniques proposed in [58] to diversify error behaviors and their corresponding reports.

4 Workflow

We use CVE-2016-6187, a slab out-of-bound write vulnerability as an example, to illustrate the prevention workflow. As shown in Figure 2, it consists of three major phases.

In the first phase, `PET` analyzes the sanitizer report to extract key information about the error in the sanitized kernel. List 1 shows the partial KASAN report for CVE-2016-6187 when it was migrated and reproduced in v5.15. From line 1 and 2 in the report, `PET` determines that the error is an out-of-bound write that affects the memory managed by the SLAB/SLUB allocator. From line 1 and 4, `PET` pinpoints the error site where out-of-bound write is triggered is `apparmor_setprocattr+0x116` - offset `0x116` from the start of function `apparmor_setprocattr`. Further analysis of the sanitized kernel image reveals that the error site corresponds to the instruction `movb $0x0, (%r14)`. When the vulnerability is triggered, register `%r14` refers to an area outside the legitimate boundary of a kernel object in the slab cache.

In the second phase, `PET` maps the error sites from the sanitized kernel to the native kernel and constructs trigger-

ing conditions correspondingly. Since sanitizers instrument the kernel during compilation, the error site and triggering condition for the sanitized kernel and the native kernel are distinct at the binary level. In our example, the error instruction in the native kernel is `movb $0x0, (%rsi, %rdi, 1)` at `apparmor_setprocattr+0x8f`, which differs from its counterpart `movb $0x0, (%r14)` in the sanitized kernel. PET leverages the DWARF debug information of the sanitized kernel and the native kernel to map the sanitized instruction to the source code statement `args[size] = '0'` and then to the native instruction. Further analyzing the mapped `mov` instruction, PET learns that register `%rsi` refers to the overflowed buffer `args`, while `%rdi` stores the value of the excessive index `size`. The trigger condition for the reported out-of-bounds error is encoded as `%rsi+%rdi*1` being beyond the bounds of the `args` buffer. This trigger condition is expressed in an eBPF program with the help of our added eBPF helper functions.

The third phase leverages the advantages of the eBPF ecosystem discussed in §2 to install the eBPF program into the native kernel at runtime to prevent errors. This error-preventing eBPF program is executed right before the `mov` instruction in the native kernel. It uses BPF helper functions to access runtime information from the kernel execution environment, and checks if the trigger condition is met. If not, the kernel continues its normal execution. Otherwise, this indicates that the kernel memory will be corrupted by the `mov` instruction, and the eBPF program takes immediate action to prevent it.

The actions include: (1) record this malicious event by writing to `/sys/kernel/debug/tracing/trace_pipe`, (2) send a `SIGKILL` signal to the runtime kernel. Upon switching back to user mode, the kernel terminates the current process because it is behaving maliciously by constructing a kernel context that could trigger the out-of-bound write error, (3) skip the `mov` instruction and direct the kernel execution to the exit of the function `apparmor_setprocattr`. Another set of eBPF programs, *i.e.*, checkpoint and restore programs in Figure 2, will oversee execution and attempt to recover the kernel state.

If the CPU is in interrupt, PET will not send `SIGKILL` signal as the scheduled process during the interrupt may not be the source of the exploit. This does not weaken PET's security improvement. On the one hand, performing exploitation in the interrupt context is challenging due to the restriction of malicious payload and memory layout manipulation: none of our extensively collected exploits/writeups target errors in interrupt. On the other hand, since the error instruction is skipped, the error-triggering attempt is effectively thwarted.

5 Error-dependent Prevention Policies

In this section, we present prevention policies for the five most commonly reported errors by state-of-the-art sanitizers. As described in §2.1, these errors are widely exploited and account for almost all errors that sanitizers can report.

```
1 UBSAN: shift-out-of-bounds in
      ↪ drivers/usb/gadget/udc/dummy_hcd.c:2293:33
2 shift exponent 257 is too large for 32-bit type
      ↪ 'int'
3 Call Trace:
4    dummy_hub_control.cold+0x1a/0xbc
         ↪ drivers/usb/gadget/udc/dummy_hcd.c:2293
```

Listing 2: The UBSAN report snippet for an integer overflow error [59].

```
1  SEC("kprobe/?") // error site
2  int BPF_KPROBE(...) {
3     // retrieve register context
4     if (?) { // triggering condition
5        // record the error
6        // send SIGKILL if not in interrupt
7        // skip the error instruction
8        // direct to function exit
9        return -1;
10 }
```

Listing 3: Example of the eBPF template to prevent integer underflow/overflow where "?" represents the details to be filled in.

5.1 Integer Underflow/Overflow Policy

Integer underflows/overflows can be detected by UBSAN and categorized into three specific scenarios: (1) arithmetic overflow resulting from arithmetic operations, *e.g.*, `add %al, 0x2` where `%al` is equal to `0xffff`, (2) shift overflow caused by shift instructions, *e.g.*, `shl %al, 10` where `%al` is equal to 0x0012, and (3) implicit conversion from move instructions (*e.g.*, `mov %rbx, %eax`) [3].

Regardless of the scenarios, our prevention policies are consistent. In the eBPF program installed before the error instruction (*e.g.*, `dummy_hub_control.cold+0x1a` shown on line 4 in List 2.), we retrieve values of related operands and then double-size them to simulate the computation of the instruction. For example, if the original operand is 32-bit, it is scaled up to 64-bit. Thus adding two 32-bit values `%eax+%ebx` becomes adding two 64-bit values `%rax+%rbx` in the simulation. By comparing the results before and after double-sizing, we can determine if an underflow or overflow will occur in the error instruction. This condition is expressed as `(%eax+%ebx)==(%rax+%rbx)?false:true` to be filled into the eBPF template. For shift overflow, an additional check is performed on the shift length to make sure it is smaller than the original size. In the example in List 2, the shift length must be smaller than 32 for a 32-bit int type.

If the triggering condition is met, meaning the expression returns `true`, PET will take actions to prevent the error. These actions are provided by the underlying mechanism and predefined in the eBPF templates. The template only requires the error site and triggering condition to be filled in line 1 and 4 in List 3 to generate a concrete program.

[3] This instruction is for clarification purpose and doesn't exist in x86.

5.2 Out-of-bound Access Policy

Out-of-bound access can be reported by KASAN. The report, which is already shown in List 1, includes not only the error site where out-of-bound access happens but also the corrupted region - stack, heap, or global and static region. The eBPF program installed before the error site, similarly to the integer underflow/overflow, can be synthesized to examine if the access is within bounds or not. The legitimate boundaries of data objects in different regions, as explained in the background, have different definitions.

For objects in the stack or global and static region, their boundary can be directly obtained from DWARF debug information. In the case of heap objects, some of them are elastic objects [17] - their size is not static but dynamically determined. Therefore, instead of getting their size from sanitizer reports or through static analysis, we add two new BPF helper functions `bpf_get_start` and `bpf_get_len` to read kernel data related to heap management to ascertain their size during runtime.

More specifically, `bpf_get_start` takes an address and determines the start of the object referred to by that address. It calls kernel function `find_vm_area` and `virt_to_page` sequentially to examine which heap region that this address belongs to. If the address is the vmalloc region, it reads the `struct vm_struct->addr` to get the starting address. If the address is the memory managed by Buddy System which takes care of large objects that cross several pages, the start address of the overflowed object is the address of the first page represented by the `struct page` object. If the address of the memory is managed by SLAB/SLUB allocator which manages small objects within page granularity, it calculates the start address of the overflowed object based on the slot size.

While `bpf_get_start` obtains the start address, `bpf_get_len` obtains the length of heap objects. The length of vmalloc object is in `vm_struct->size`, SLAB/SLUB object is in `page->slab_cache->objec_size`. For Buddy System objects, if `page->flag` is clear, the size is one page, otherwise, the size is calculated via `page->compound_nr*PAGE_SIZE`.

For all out-of-bound access, the triggering condition can be uniformly expressed in the eBPF template as `(start<=addr)&&(addr<end)?false:true`. Still taking CVE-2016-6187 as an example, the `addr` is `%rsi+%rdx*1`, the `start` is `bpf_get_addr(%rsi)`, and the `end` address is `bpf_get_addr(%rsi)+bpf_get_len(%rsi)`. Just like integer overflow, this condition along with the error site are filled into the eBPF template (List **??** in Appendix C) to synthesize a concrete program.

5.3 Use-After-Free Policy

KASAN can also report use-after-free errors, which are temporal in nature and have two error sites. The first is the free site where a dangling pointer is created, such as the one

```
1 KASAN: use-after-free in route4_get+0x7d/0xc0
2 Read of size 4 at addr ffff888006358640
3 Call Trace:
4     route4_get+0x7d/0xc0
5 Allocated by task 1137:
6     route4_change+0x18f/0xde0
7 Freed by task 69:
8     kfree+0x90/0x220
9     route4_delete_filter_work+0x17/0x20
```

Listing 4: The KASAN report snippet for a use-after-free error [60].

at `route4_delete_filter_work+0x17`, as indicated in line 9 of List 4. The second site is where the dangling pointer is dereferenced, such as `route4_get+0x7d` in line 4 of List 4. To prevent use-after-free, two eBPF programs are installed - one before the free site and another before the use site.

The eBPF program before the free site records the object address to a BPF map and quarantines the object by skipping the free operation. This means that the object won't be freed and recycled back to the allocator for future reuse. Such quarantine prevents exploitation because attackers cannot reclaim the same piece of memory and overlap the freed object with another object under control, which is the must of use-after-free exploitation. We notate the action at the free site as `map=map+addr`. The eBPF program before the use site queries the BPF map to see if the dereferenced pointer is referring to a quarantined object. If so, the eBPF program takes actions to prevent error, similar to the other policies. This triggering condition is notated as `ptr ∈ map?true:false`. The eBPF programs are synthesized by filling the action and the triggering condition into templates. Note that double-free is a special case of use-after-free. Its detection policy is no different except that the use site is also a free site.

The real challenge in this policy is not the construction of the triggering condition but how to avoid memory exhaustion. If quarantined objects are never recycled, attackers can repeatedly invoke the free site, causing a large amount of memory to be quarantined and affecting kernel functionality. Therefore, the prevention policy must determine when it is safe to release quarantined memory.

Our solution to avoid memory exhaustion is to utilize two BPF helper functions, `bpf_timer_init` and `bpf_timer_start`, to start a timer at the free site. The timer will wake up a task periodically to execute a callback function, set by `bpf_timer_set_callback`. This callback function performs a sweep of the entire physical memory, checking for the existence of any dangling pointers referring to quarantined objects. If any are found, quarantine continues. Otherwise, it is deemed safe to recycle the quarantined memory because the dangling pointer no longer exists in the memory. This sweeper is integrated into the eBPF template for preventing use-after-free.

Physical memory in modern OS can be very large, making sweeping the entire memory an inefficient and slow process.

```
1 KMSAN: uninit-value in tcp_recvmsg+0x6cf/0xb60
2     tcp_recvmsg+0x6cf/0xb60
3 Local variable msg created at:
4     __sys_recvfrom+0x81/0x900
```

Listing 5: The KMSAN report snippet for an uninitialized access error [61].

To mitigate this issue, we introduce an optimization to reduce overhead. Technically, we extract the type of allocated objects at the allocation site, then investigate all structures that can be allocated to the SLAB/SLUB region. If one field of an investigated structure is in the pointer type of the allocated object, this pointer field may be a potential dangling pointer at runtime. The optimized sweeper only sweeps the slab caches that store these interesting structures and only checks the offset of the pointer field in the structure.

5.4 Uninitialized Access Policy

KMSAN reports uninitialized access. As another temporal memory error, it also has two error sites. One is the creation site where an object is created on the stack or heap, like `__sys_recvfrom+0x81` indicated in line 4 in List 5. Another is the access site where the object is read but not fully initialized, like `tcp_recvmsg+0x6cf` indicated in line 2 at List 5. As such, two eBPF programs are required to prevent uninitialized access. The first eBPF program is installed after the creation site. It stores the size of the created object in BPF map X and the uninitialized content in BPF map Y, using the object's address as the key for both maps. The second program uses the object's address to query map X and retrieve its size, then uses the size to retrieve the full content of the object from map Y.

By comparing the content of an object before access and after creation, we can determine if it has been properly initialized. The aggressive policy considers the triggering condition met if the contents have at least one byte in common. The conservative policy requires the contents to be exactly the same. The two policies have their own problems. The aggressive policy may result in false positives if some bytes remain unchanged after initialization, while the conservative policy may miss partial initializations, leading to false negatives. Experiment results indicate that the conservative policy is more effective and thus is used by default in the the eBPF template (List **??** in Appendix C) If manual efforts are allowed, the ideal policy would be to specify the uninitialized range within the object.

5.5 Data Race Policy

Data race occurs where two instructions executed at separate CPUs access the same memory simultaneously without proper synchronization. For instance, List 6 shows a data race where CPU 1 reads from `0xffffffff8713bbb0` at

```
1 BUG: KCSAN: data-race in tcp_send_challenge_ack /
      ↪ tcp_send_challenge_ack
2 read to 0xffffffff8713bbb0 of 4 bytes on cpu 1:
3     tcp_send_challenge_ack+0x116/0x200
4 write to 0xffffffff8713bbb0 of 4 bytes on cpu 0:
5     tcp_send_challenge_ack+0x15c/0x200
```

Listing 6: The KCSAN report snippet for a data race [62].

`tcp_send_challenge_ack+0x116` (line 3), while CPU 0 writes to the same address at `tcp_send_challenge_ack+0x15c` on CPU 0 (line 5). In `PET`, we utilize four eBPF programs to identify data races at runtime.

The four eBPF programs are organized in pairs and share a single BPF map. Each pair is responsible for monitoring one access in a potential data race. The first program in the pair is installed prior to the access and performs the P operation: call the `bpf_map_update_elem` helper function with `BPF_NOEXIST` argument to spin locks the shared map and do lookup-update atomically. If a record of the same memory in the shared BPF map exists, `EEXIST` is returned, indicating that another instruction from a different CPU is currently accessing the same memory, thus signaling a potential data race. In the absence of data race, the helper function returns 0, indicating that the update was successful and no data race has occurred. The second program in the pair is installed after the access and performs the V operation: call `bpf_map_delete_elem` to delete the record of the accessed memory atomically.

Since data race could be either benign or harmful [7, 63, 64], by default, the eBPF program for data race won't be like that for other errors, sending out `SIGKILL` signal to kill the current process if the P operation fails. However, the eBPF template offers an option if human experts confirm the damage of the reported data race.

6 Error-independent Mechanisms

6.1 Report Processor

The report processor extracts critical information from sanitizer reports. As we showed in previous sections, the formats of sanitizer reports vary for different error types: the error site of out-of-bound access can be found in the report title (line 1 in List 1) while that of integer underflow is in call trace (line 4 in List 2). In addition to error sites, some prevention policies need unique information. For example, the use-after-free error needs the allocation site (line 6 in List 4) to optimize the dangling pointer sweeper.

To eliminate the differences in information requirements, the report processor in `PET` provides a unified interface for overlaying policies. The interface is a set of regular expressions - each filtering out the desired keywords from report lines that are separated in advance.

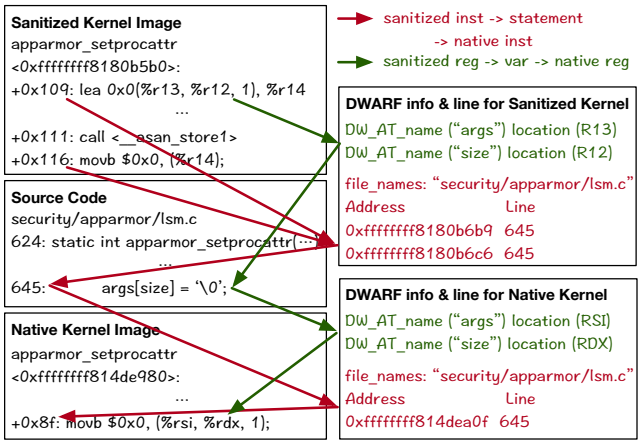

Figure 3: Two translation flows in the sanitized-native mapper that constructs triggering condition at the binary level to be expressed the eBPF program that detects errors in the native kernel.

6.2 Sanitized-Native Mapper

The extracted error information from the report needs to be translated from the sanitized kernel to the native kernel to construct the triggering condition for the eBPF programs to prevent errors. To demonstrate this mapping process, we continue to use the slab out-of-bound access error CVE-2016-6187 as an example and provide an illustration in Figure 3.

In this sanitizer-native mapper, there are two translation flows. The first flow identifies the error site where the eBPF program for prevention can be installed. It translates the instruction that triggers the error in the sanitized kernel, to the problematic statement in the source code, and finally the counterpart instruction in the native kernel. As shown in Figure 3, with `apparmor_setprocattr+0x116` outputted from the report processor, `PET` uses it as the clue to pinpoint the error instruction `movb $0x0, (%r14)` in the sanitized image. With the DWARF debugging information of the sanitized kernel, `PET` maps this error instruction to line 645 in file `securit/apparmor/lsm.c` in the source code. Without further analysis, `PET` retrieves the DWARF debug information of the native kernel to map this source code statement to the counterpart instruction in the native kernel image - the `mov` at `apparmor_setprocattr+0x8f` which is the instruction to install the eBPF program for prevention.

At this error site, the second translation flow constructs the triggering condition to be expressed in the eBPF program. In Figure 3, the destination operand `$r14` in the `mov` instruction records the written address in the sanitized kernel. This address is calculated from the `lea 0x0(%r13, %r12, 1)` instruction at `apparmor_setprocattr+0x109`. By analyzing the DWARF information for this `lea` instruction, `PET` learns that register `%r13` refers to a buffer named `args` in a slab cache. The kernel accesses this buffer using an index named `size` which is stored in register `%r12`. As such, `PET` concludes that the overflowed buffer is `args` and the excessive index is `size`. To continue the second translation flow, `PET` uses the DWARF information for the native kernel and connects the `args` variable with the `%rsi` register and the `size` variable with the `%rdx` register at the error site.

As described in §5.2, the triggering condition for out-of-bounds access is `(start<=addr)&&(addr<end)?true:false`. The eBPF program uses this condition to check if the accessed `addr $rsi+%rdx*1` is within the legitimate boundary with `bpf_get_addr(%rsi)` as the `start`, `bpf_get_addr(%rsi)+bpf_get_len(%rsi)` as the `end`. This condition is filled into the eBPF program, which is installed at the error site identified through the first translation flow, to prevent the out-of-bound access error.

6.3 Checkpoint-Restore Analyzer

When the triggering condition constructed by the sanitized-native mapper is met in the native kernel, `PET` takes a series of actions including a checkpoint-restore mechanism to continue the kernel execution smoothly. This is achieved through additional eBPF programs and BPF maps.

To be specific, we install an eBPF program at the entry of the function containing the error instruction to reserve the register context before the prologue. Another eBPF program is installed at the function exit to restore the register context after epilogue and rewrite register `%rax` with an error code which will be returned to the caller. This allows `PET` to take advantage of the error handling mechanism in the Linux kernel to quickly deliver the error message back along the calling chain to the system call entry, terminating the malicious process as quickly as possible. To determine the appropriate error code, our checkpoint-restore analyzer will count the error-handling paths in the function and choose the most frequently used one. If the most frequently used code is `EAGAIN` which indicates to re-execute the called function, we replace it with `EACCESS` to prevent endless error triggering. In situations where the return type is either a pointer or void, we use `PTR_ERR` to convert the return value to a negative figure or rewrite `%rax` to be 0.

In most cases, a checkpoint program at the function entry and a restore program at the function exit is enough. However, if paired operations, such as (de)allocations, (un)locks, device (un)registration, reference count (in|de)cement, and pointer nullification, exist in the execution path from the entry to the error site, additional eBPF programs are needed. These eBPF programs record the address of allocated objects, operated locks, registered device structures, reference count, and pointers in BPF maps. With BPF maps shared, the restore eBPF program at the function exit can deallocate objects, perform unlock, unregister devices, recover reference count and pointers. To locate these checkpoint sites, `PET` performs static analysis of all program paths from the function entry to the error site, identifying them first at the source

code level, and converting them to binary form using the previously described sanitized-native mapper.

Limitations. Though our empirical evaluation indicates that the kernel runs stably after PET prevents errors - presumably due to the short path from checkpoint to restore (44.17 instructions on average), it is crucial to underscore the limitations of PET as a temporary solution in recovering a consistent state. On the one hand, PET relies on human expertise to mark paired operations of interest. This could result in missing, especially during the kernel's ongoing development, which requires PET to be updated correspondingly. On the other hand, compared with prior works (*e.g.*, [65–69]) that employ logging and rollback techniques, PET will overlook memory cells that could influence kernel functionality[70]. Therefore, it is recommended to offline PET and apply the official patch upon its release.

6.4 Library of eBPF Helper Functions

PET employs three types of eBPF programs: (1) checkpoint installed at the function entry and paired operations towards the error, (2) error prevention placed before the error site, (3) restoration installed at the function exit. We enhance the existing eBPF ecosystem in the Linux kernel by adding more eBPF helper functions to support these three eBPF program types, in addition to helper functions previously discussed.

For checkpoint programs, `bpf_get_regs` is used to retrieve the register context, addresses of allocated objects, locks, device structures, storing them to separate BPF maps shared with the restoration program. Considering that the error function can be re-entered, the key for these BPF maps is the concatenation of the function name and the current process ID obtained using `bpf_get_current_pid_tgid` helper function.

For error-preventing programs, we record malicious events to `/sys/kernel/debug/tracing/trace_pipe` using `bpf_printk` and send a `SIGKILL` signal using `bpf_send_signal` - both helper functions already exist in the eBPF ecosystem. Then, we add `bpf_set_regs` to set register `%rax` to the magic number `0xdeadbeef`. This magic number notifies the restore program that the triggering condition is met and restoration is a must. Then, we call `bpf_set_regs` again to set register `%rip`, directing the kernel to the function exit.

Finally, for the restore program, we use `bpf_get_regs` to check register `%rax`. If `%rax` is not equal to `0xdeadbeef`, the program simply clears all related records in the shared BPF maps and returns to the caller. Otherwise, the program restores register context using `bpf_set_regs`, deallocates objects using the newly added `bpf_kfree`, performs unlock using the newly added `bpf_unlock`, (un)register devices using the newly added `bpf_register`, (in|de)crement reference count using the newly added `bpf_(in|de)refcnt`, and overwrites the value of register `%rax` with the error code determined by the checkpoint-restore analyzer.

7 Implementation

PET uses in total 500 lines of Python scripts to integrate all infrastructure mechanisms and eBPF program templates for prevention policies. A total of 639 lines of C code were added for BPF helper functions, along with an additional 440 lines of C code for all eBPF program templates. The DWARF analysis and static analysis over LLVM IR include roughly 3000 lines of C/C++ code. The implementation is open-sourced on Github [71].

eBPF Programs. Kernel Address Space Layout Randomization (KASLR) randomly changes the kernel base address every time the system boots up. It does not affect the decision of where to install the eBPF programs as the site can be specified using relative offset like `"func+offset"`. For other addresses that cannot be specified statically in the eBPF programs, we use the latest BPF toolchain Skeleton [72]. Skeleton defines these addresses as global variables and looks them up in `/proc/kallsyms` when loading eBPF programs. By adding relative offsets to these addresses, Skeleton rewrites the value of global variables to bypass KASLR.

In the prevention policy for use-after-free errors, for the sake of performance, the sweeper only scans a fixed memory range each time woke up, avoiding excessive CPU usage. In our experiments, we empirically tested different range sizes and time intervals, finding that a 256 MB range per 8 seconds was the optimal choice. Further information can be found in §8.3 and Appendix B.

DWARF Analysis. The sanitized-native mapper (§6.2) uses DWARF debugging information to identify the error instruction in the native kernel. However, its accuracy can be compromised by compiler optimizations - a single instruction in the sanitized kernel corresponds to multiple source code statements which further corresponds to multiple instructions in the native kernel. As a result, the mapper may identify a set of instructions in the native kernel, but only a few of them are actual errors. We resolve the issue by cross-checking two translation flows in the mapper. An instruction in the native kernel is considered an error if and only if its operator and the names of its operand variables match the error instruction in the sanitized kernel. This cross-checking reduces the false positive number for 2 to 0 per case in our evaluation test set.

Static Analysis. PET employs static analysis methods to locate checkpoint sites (§6.3) and optimize the sweeper in the use-after-free prevention policy (§5.3). Our implementation is based on the LLVM infrastructure and uses a customized Clang [73] to produce unoptimized IR files.

To identify paired operations that require checkpointing, we maintain a list of allocation interfaces (*e.g.*, `kmalloc`), lock interfaces (*e.g.*, `raw_spin_lock`), device registration interfaces (*e.g.*, `register_filesystem`), reference counter interfaces (*e.g.*, `refcount_inc`), and patterns of pointer nullification. Our static

analysis starts by building a control-flow graph of the function containing the error instruction at the IR level. By traversing all paths from the function's entry point to the error instruction, we collect all `Call` instructions of interest, marking them as checkpoint sites.

`PET` selects slab caches that potentially contain dangling pointers to optimize the sweeper sweeping. Our static analysis employs a worklist procedure and coordinately analyzes `GetElementPtr` and `Cast` instructions. This worklist procedure is sound and in the worst case where all slab caches are chosen, the sweeper is degraded to the entire-sweeping mode. More details can be found in Appendix A.

8 Evaluation

8.1 Testcase Set

We collect real-world vulnerabilities from two resources as the testcases of `PET`. One is those reported by Syzkaller from 2018 to 2023. Another is those used in kernel exploitation works published in the past 10 years.

All vulnerabilities in our testcase set must meet the following criteria: ❶ their PoC programs or exploits are publicly available for evaluation of `PET`' prevention success, and ❷ they can be migrated and reproduced in v5.15 - the latest long-term version at the time of our experiment- so that we can measure `PET`' scalability and overhead in the presence of multiple errors.

Following these criteria, we built a testcase set covering 5 different error types, including 2 integer overflows reported by UBSAN, 15 out-of-bound access and 10 use-after-free reported by KASAN, and 2 uninitialized access reported by KMSAN. This testcase set is representative and covers various common error types reported by existing sanitizers. In particular, our testcase set covers all types of memory regions that can be corrupted, including the stack, global and static regions, Buddy System, SLAB/SLUB allocator, and vmalloc regions.

Since KCSAN reports are not accompanied by any proof-of-concept (PoC) programs, we were unable to find any data races that met our selection criteria. Therefore, we randomly selected two race conditions that meet our criteria into our test case set: CVE-2017-2636 [74] and CVE-2021-4083 [75]. Additionally, we randomly selected 3 data races reported in v5.15 for performance measurement. We didn't find any report from KTSAN. As such, we collected a total of 34 vulnerabilities in our testcase set.

8.2 Effectiveness

For the effectiveness of `PET`, we aim to answer the following questions, in comparison with the most related prior work [30]: ❶ Can `PET` prevent the triggering and exploitation of errors? ❷ Does `PET` affect normal execution if the triggering condition is not met? ❸ Does the kernel remain stable and not panic after the error triggering is prevented?

Experiments Setup. To answer the first question, we migrated all errors in our testcase set to the v5.15 kernel. Then, we built a kernel image with sanitizers enabled to validate that the collected PoCs and exploits can trigger these errors and generate reports to be analyzed by `PET`. After this, we compiled another kernel image with sanitizers disabled and deployed the synthesized eBPF programs to this native kernel. We ran the PoCs and exploits on this kernel hardened by `PET` to observe if the errors could still be triggered and exploited. The setup of [30] is similar except that the second kernel image is instrumented with undo operations.

To answer the second question, we modified the collected PoCs and exploits to make sure that the error sites are still executed but the triggering condition is no longer met. We confirmed this on the sanitized kernel and then ran the modified PoCs and exploits on the hardened native kernel. We observed whether the eBPF programs would still record errors, which indicate false alarms. This modification took us approximately 150 man-hours.

To answer the last question, we extensively tested the kernel with migrated errors by repeatedly running all PoCs and exploits for 100 times to ensure that the kernel was thoroughly 'attacked'. After the eBPF programs prevented all the error triggering, we continued to use the machine for daily activities, such as browsing the web on Google, YouTube, Twitter, Overleaf, Email, and other internet services, participating Zoom meetings, messaging on Slack, playing music on Spotify, running Dockers containers for CTF challenges, plugining/ unplugining external monitors, etc. This experiment is carried on for 7×24 hours.

Experiment Results. Given an error, for `PET` and [30] to be considered effective, it must fulfill the following three criteria: ❶ successfully prevent the error from triggering when the condition is met, ❷ ensure normal execution when the condition is not met, and ❸ the kernel remains stable without panic after the error has been prevented.

The sampled effectiveness results are shown in Table 1 (More complete results in Appendix Table 4). In general, `PET` is effective for all errors in our testcase set, except three data races we cannot evaluate due to the lack of PoC programs. `PET` accurately located the error sites of all types and effectively synthesized triggering conditions and actions to prevent their triggering, in < 5 minutes. During the 17-344 days time window for kernel developers to patch these errors, `PET` can provide temporary protection for the kernel.

For out-of-bound access errors in the stack, global and static regions, `PET` utilizes its sanitized-native mapper to determine the boundaries accurately, *e.g.*, `[$rsp+0x50, ↪ $rsp+0xa0)` for stack objects and `[0xffffffff822479c0, ↪ 0xffffffff822479c0+0x18)` for global objects. For out-of-bound access errors on kernel heap, `PET` leverages BPF

CVE/SYZ ID	Sites for eBPF Installation	Action & Triggering Condition	Effectiveness (PET/[30])	Time Window (days)
Integer Underflow/Overflow				
b5b251b [76]	`dummy_hub_control+0x3f` (spinlock) `dummy_hub_control+0x225`	`lock_map[pid] = $rdi` `$eax<<$edx == $rax<<$edx & $edx<32 ? false : true`	●/◐	79
Out-of-bound Access on Stack				
2022-1015	`nft_do_chain+0x243`	`$rdi ∈ [$rsp+0x50, $rsp+0xa0)? false : true`	●/◐	147
Out-of-bound Access on Global and Static Region				
2017-18344	`show_timer+0x81`	`$rdx ∈ [0xffffffff822479c0, 0xffffffff822479d8)? false : true`	●/◐	90
Out-of-bound Access on Buddy System Heap				
2022-27666	`null_skcipher_crypt+0x4b`	`$rdi+$rdx ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/◐	17
Out-of-bound Access on vmalloc Heap				
2020-14386	`tpacket_rcv+0x21a` (spinlock) `tpacket_rcv+0x6f6`	`lock_map[pid] = $rdi` `%rax ∈ [start($r10), start($r10)+len($r10))? false : true`	●/◐	39
Out-of-bound Access on SLAB/SLUB Heap				
2022-34918	`nft_set_elem_init+0x3e`	`$rdi+$rcx ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/◐	38
797c55d [77]	`watch_queue_set_filter+0x81` (alloc) `watch_queue_set_filter+0x78d`	`alloc_map[pid]=$rdi` `$r15+0x8 ∈ [start($r15), start($r15)+len($r15))? false : true`	●/◐	344
Use-After-Free				
2022-2586	`nft_obj_destroy+0x3f` (free) `nf_tables_fill_setelem.isra.0+0x140` (use)	`map ⋃ $rdi; selective_sweep(kmalloc-256, 0x20)` `$rbx+$rax ∈ map ? true: false`	●/◐	97
be93025d [78]	`__route4_delete_filter+0x3c` (free) `__route4_delete_filter+0x3c` (use)	`map ⋃ $rdi; selective_sweep(kmalloc-192, 0x28)` `$rdi ∈ map ? true : false`	●/◐	73
Uninitialized Access				
2039c557 [61]	`__sys_recvfrom` (create) `tcp_recvmsg+0xb8` (use)	`map[$rsp+8-200] = mem($rsp-0xc0, 0x60)` `map[$r13] == mem($r13, 0x60)? false : true`	●(default conservative) ◐(aggressive) / ◐	248
Data Race				
2017-2636 [74]	`n_hdlc_send_frames+0x118` (write) `n_hdlc_tty_ioctl+6b` (write)	`P($rbp+0x310); V($rbp+0x310)` `P($rsp); V($rsp)`	●/◐	40

2021-4083 [75]	`unix_stream_read_generic+0xeb` (spinlock)
	`unix_stream_read_generic+0x120` (mutex)
	`unix_stream_read_generic+0x138` (read)
	`unix_gc+0x33` (spinlock)
	`unix_gc+0x28e` (write)
		`lock_map[pid]=$rdi`
		`mutex_lock[pid]=$rdi`
		`P($r13); V($r13)`
		`lock_map[pid]=$rdi`
		`P($rsp+0x38); V($rsp+0x38)`
			●/◐	224

Table 1: The sampled results for the effectiveness of PET- the three criteria are described in §8.2. ● indicates that all three criteria are satisfied; ◐ means criteria ❷ (*i.e.*, no false alarm) is not met; Complete results are in Table 4.

helper functions `bpf_get_start` and `bpf_get_len` from the eBPF helper library to determine the boundaries of accessed objects. The accessed address is stored in registers that are identified using the report processor and the sanitized-native mapper in PET.

For use-after-free errors, PET creates a sweeper at the free site which periodically scans for the presence of dangling pointers. The sweeper works in two modes: entire-sweeping mode which scans the entire physical memory (*e.g.*, be93025d [78]) and selective-sweeping mode which only scans slab caches that have the potential for containing dangling pointers (*e.g.*, the `kmalloc-256` cache for CVE-2022-2586 with pointers at `0x20` offset). In our testcase set, 3 out of 10 use-after-free cases employ the entire-sweeping mode while the remaining 7 cases use selective sweeping mode. The performance comparison between the two modes will be presented in §8.3.

For uninitialized access errors, our evaluation shows that the conservative policy is more effective than the aggressive policy - the aggressive policy reports false alarms when executing PoCs that are modified to not trigger errors. Therefore, a conservative approach is used by default, unless an aggressive approach is guaranteed to not result in false positives.

For data race, PET identified kernel code that races the same memory with at least one write operation and effectively detected race using P/V operations.

In addition to error sites, PET also locates paired operations for all testcases, *e.g.*, b5b251b [76], CVE-2020-14386, 797c55d [77], be93025d [78], and CVE-2021-4083 [75] in the sampled Table. PET employs the checkpoint-restore mechanism to return the kernel to normal after the triggering is prevented. Though this approach has certain limitations as we discussed in §6.3, in our evaluation, the kernel ran stably for 7 × 24 hours after PoCs and exploits for all testcases are run for 100 times. We continued using it for more than 3 months and to date, the kernel still functions properly.

In comparison, while [30] prevents the error triggering, it inevitably raises false alarms and impacts the normal execution because undo operations are always performed regardless of whether the condition is met or not. Therefore, under our defined criteria, [30] is considered partially effective.

8.3 Overhead & Scalability

Regarding the overhead of PET, we are seeking answers to the following questions, also in comparison with the most related prior work [30]: ❶ To which extend does PET slow down the kernel? ❷ How much latency do eBPF programs introduce to the kernel execution? ❸ What is the optimal configuration for sweeper in use-after-free prevention policy? ❹ Is PET scalable in the presence of multiple errors? ❺ How much extra memory is needed to support PET?

Experiments setup. We using a series of benchmarks, including OSBench for measuring the performance of basic OS operations such as process and thread creation, perf-bench for testing scheduler and IPC mechanisms, and a range

	Integer	Slab OOB	Page OOB	Stack OOB	Global OOB	VmallocOOB	UAF		Uninitialized	Data Race		Std. dev.
	b5b251b [76]	2021-34693	2022-27666	2c09122 [79]	2017-18344	2020-14386	5d5bb09c [60]	2022-4154	2039c5 [61]	2017-2636	2021-4083	
OS Core primivtives (PET/[30])												
OSBench	0.8% / 2.7%	0.0% / -1.4%	0.7% / -0.3%	0.4% / -0.4%	0.1% / -0.4%	4.2% / 0.5%	2.1% / -0.8%	3.1% / -0.4%	1.9% / -0.7%	-0.4% / -0.7%	-0.9% / 0.0%	0.02 / 0.01
perf-bench	0.6% / 3.4%	0.4% / -0.1%	0.0% / 0.1%	-0.2% / 0.1%	0.1% / 0.1%	6.1% / -0.2%	3.4% / -0.4%	5.9% / 2.1%	1.4% / -0.1%	0.0% / 2.0%	1.7% / 0.2%	0.02 / 0.01
CPU intensive (PET/[30])												
OpenSSL	1.9% / -0.8%	0.0% / -1.4%	-0.1% / -0.5%	0.2% / -0.8%	0.2% / -0.8%	0.2% / -0.1%	1.2% / -0.6%	0.4% / -0.8%	2.1% / -0.2%	0.3% / 0.0%	0.0% / -0.6%	0.01 / 0.00
GIMP	-0.5% / -0.5%	-1.1% / 0.1%	1.3% / -1.2%	-1.1% / 1.9%	-3.0% / 1.9%	0.9% / 0.5%	-0.2% / -0.7%	1.1% / 0.7%	0.7% / 0.4%	0.2% / -1.0%	1.4% / 1.8%	0.01 / 0.01
MP3 Encoding	0.8% / -0.2%	0.2% / -0.4%	0.2% / -0.4%	1.0% / 0.6%	0.6% / 0.6%	1.4% / 0.2%	0.7% / -1.0%	1.6% / 0.4%	1.9% / -0.2%	-0.1% / 1.7%	0.0% / 0.3%	0.01 / 0.01
I/O intensive (PET/[30])												
SQLite Speedtest	-1.9% / -2.6%	-0.7% / -1.0%	-0.2% / -1.7%	-0.4% / 2.4%	-1.5% / 2.4%	-1.2% / 0.2%	0.0% / -1.3%	1.9% / -0.3%	1.6% / 1.2%	-1.1% / 0.3%	-0.1% / 0.1%	0.01 / 0.02
WireGuard Stress	-0.9% / -0.6%	0.1% / -0.1%	0.1% / -0.6%	-0.2% / 2.0%	-0.5% / 2.0%	11.5% / 0.8%	1.1% / -0.2%	1.6% / 0.4%	-0.1% / -1.1%	0.8% / 0.9%	-2.9% / 0.8%	0.04 / 0.01
Git	0.9% / -0.4%	0.1% / -0.3%	0.2% / -0.1%	0.4% / 0.9%	0.2% / 0.9%	14.5% / 0.9%	0.6% / -0.1%	0.5% / 0.9%	1.7% / 0.8%	0.3% / -0.7%	-0.4% / 0.2%	0.04 / 0.01
Linux Kernel Compile	1.9% / 1.3%	-0.1% / 0.0%	0.1% / 1.9%	0.0% / 3.6%	0.3% / 3.6%	1.4% / 1.7%	2.2% / 1.8%	3.2% / 0.4%	2.8% / 2.6%	1.6% / 2.0%	1.0% / 2.2%	0.01 / 0.01
XZ Compression	-0.7% / -0.3%	0.7% / 0.2%	0.9% / -0.9%	0.0% / 0.4%	0.5% / 0.4%	1.7% / -0.7%	1.6% / -1.1%	2.3% / -0.3%	1.2% / -0.3%	0.1% / -0.6%	-0.6% / -0.9%	0.01 / 0.01
Server Tasks (PET/[30])												
Apache	-1.9% / -3.7%	0.4% / -3.1%	0.4% / -2.5%	-1.1% / -1.9%	-0.4% / -1.9%	10.6% / -2.1%	4.1% / -4.0%	3.6% / -4.1%	0.6% / 3.7%	1.2% / -2.1%	1.6% / -3.0%	0.03 / 0.02
Nginx	1.1% / -2.8%	0.8% / -2.5%	-0.2% / -2.2%	0.2% / -0.8%	0.6% / -0.8%	10.3% / -2.0%	6.0% / -1.6%	5.3% / -1.2%	1.2% / -1.3%	0.2% / -0.6%	-1.1% / -1.7%	0.03 / 0.01
perl-benchmark	-0.4% / 1.2%	-0.4% / -0.8%	0.0% / 0.9%	-1.2% / 1.1%	-0.5% / 1.1%	-1.1% / 0.6%	0.1% / -0.2%	3.0% / 0.4%	-0.4% / 0.6%	0.5% / 1.9%	0.6% / 1.0%	0.01 / 0.01
Redis	1.7% / 1.7%	1.7% / 1.0%	-0.8% / 1.8%	0.5% / 3.7%	-1.7% / 3.7%	-0.6% / -0.6%	0.8% / 5.0%	3.0% / -2.0%	1.7% / 1.2%	0.5% / -0.1%	-1.2% / 3.9%	0.01 / 0.02
average	0.3% / -0.1%	0.1% / -0.7%	0.2% / -0.4%	-0.1% / 0.9%	-0.4% / 0.9%	4.3% / 0.0%	1.7% / -0.4%	2.6% / -0.3%	1.3% / 0.5%	0.3% / 0.2%	-0.1% / 0.3%	0.01 / 0.01

Table 2: The sampled performance of PET in preventing different errors using representative benchmarks. Complete results are in [80].

	Operations	Time
Checkpoint-Restore	checkpoint@func entry	2702.72 ns
	restore@func exit if no error	2617.72 ns
	restore@func exit if error occurs	2898.06 ns
Integer overflow/underflow	simulation	1796.61 ns
Out-of-Bound	bpf_get_start	1935.16 ns
	bpf_get_len	1829.12 ns
Uninitialized Access	store uninitialized content@creation site	2820.52 ns
	compare content@access site	3386.4 ns
Data Race	P/V operation	2608.23 ns
Use-After-Free	store object address@free site	2806.69 ns
	selective-sweeping (async)	13.96 ms
	entire-sweeping (async, 256MB/8s)	277.46 ms

Table 3: The latency of critical operations in eBPF programs.

of real-world applications like MP3 encoding for calculation-intensive tasks, SQLite for IO-intensive tasks, and Redis, perl-benchmark, Apache, and Nginx for server workload.

The eBPF programs to be installed into the kernel were compiled with JIT enabled for the sake of performance. To determine the latency of critical eBPF operations during kernel execution, we inserted two `rdtsc` instructions before and after the operations of interest and collected the time difference between them.

We conducted experiments to determine the most efficient sweeping mode for the sweeper in our use-after-free prevention policy. We varied the sweep interval (1s, 2s, 4s, 8s) and range (128 MB, 256 MB, 512 MB) to identify the optimal configuration for deployment. Furthermore, we compared the performance of the entire-sweeping mode and selective-sweeping mode to demonstrate the effectiveness of our optimization efforts. We randomly chose 5d5bb09c [60] as the representative of selective-sweeping and CVE-2021-4154 as the representative of entire-sweeping as our testbeds.

In the presence of multiple errors, eBPF programs for different errors need to be installed simultaneously. To access the scalability of PET, our experiment randomly sorted out all errors in the testcase set and measured performance when there are 2, 4, 8, 16, and all errors present.

All of the above experiments were conducted automatically three times, and the average results are presented. The machine for evaluation runs Ubuntu 22.04 LTS with an Intel(R) Core(TM) Intel i7-6700HQ @ 3.50GHz (4 Cores / 8 Threads) CPU, 16GB RAM and 1000GB SSD.

Experiments result. In Table 2, we present the sampled performance overhead of different prevention policies as the answer to question ❶. Readers can refer to [80] for all results. The average performance overhead of PET is below 3% in all cases and on par with [30], except for CVE-2020-14386 which is out-of-bound access in the vmalloc region. This exception is caused by the overhead of networking-related benchmarks including WireGuard, Git, Apache, and Nginx which are all over 10%. This is because the error site is in kernel function `tpacket_rcv` which is heavily used in the network stack to receive packets from NAPI NIC. Even so, the average overhead is 4.27%, much lower than other Linux kernel protections proposed in recent years (*e.g.*, [81]). The overhead of Redis is not heavily impacted because, in our evaluation, it runs in a "standalone" mode which uses loopback rather NIC.

Table 3 presents the latency introduced by eBPF programs that perform critical actions in various prevention policies, as per question ❷. The latency for checkpoint and restore, simulation in integer underflow/overflow, dynamic boundary determination for heap objects, storing and comparison for uninitialized access errors, P/V operation in data races, and storing object address in use-after-free are all below 3500 ns, which is far beyond the range of human perception. For selective-sweeping and entire-sweeping, their latencies are less than 300 ms and they are performed in the background asynchronously over one single CPU. So, the overall overhead of use-after-free errors, regardless of the sweeping mode, are all around 2% as presented in Table 2. Additionally, the latency of less than 300 ms is well within the acceptable range, compared to similar works like Shuffler [82].

Furthermore, Table 3 shows that the selective-sweeping mode is 21 times faster than the optimal configuration of the entire-sweeping mode: 13.96 ms vs. 277.46 ms, which is concerned in question ❸. This optimal configuration for the entire-sweeping mode, which is scanning 256 MB per

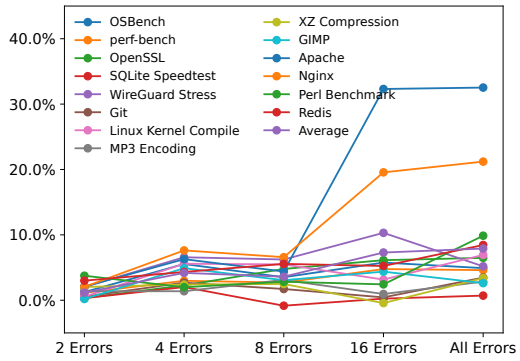

Figure 4: Scalability of PET in the presence of multiple errors. The spikes in Apache and Nginx were caused by CVE-2020-14386 - the vmalloc OOB.

8s is obtained from empirical results shown in Figure 5 in Appendix B.

Figure 4 shows the scalability of PET in the presence of multiple errors, as per question ❹. As demonstrated, the performance overhead increases linearly as more errors need to be prevented, which is on par with the results of [30]. The total accumulated overhead for all 34 errors tested by most benchmarks is approximately 10%. The overhead tested by Apache and Nginx increases significantly due to the vmalloc OOB, as previously discussed.

For question ❺, the extra memory used by PET is BPF maps. The largest BPF map among all policies is the one used to quarantine objects for use-after-free. It includes 20 million entries, corresponding to 915 MB. When there are multiple use-after-free errors, this map is shared by all sweepers, thus eliminating the need for additional memory. Such memory overhead isn't an issue for modern OS kernel because it has full access to physical memory which is very large thanks to the advancements in SDRAM technology.

9 Related Works

Kernel Patching. As a widely-used approach to remediating kernel errors, patching has been extensively researched in previous works (*e.g.*, [83–86]). Especially, assuming patches are available, Livepatch techniques [87, 88] can fix kernel errors without rebooting the system. PET also offers the advantage of on-the-fly enabling. However, PET does not rely on the availability of patches and prevents discovered errors before patches are available.

Kernel Hardening. Developers harden the kernel to raise the bar of attack. These include structure layout randomization [89], freelist randomization [90], KASLR [91], free pointer obfuscation [92] in Kernel Self-Protection Project [93], AutoSlab [94], PAX_USERCOPY [95] developed by PaX/Grsecurity [96], and XPFO [97], redleaf [98], VirtualGhost [99], KCoFI [100], NestedKernel [101], μScope [102], MemSentry [103], xMP [81], and Kasper [104], proposed in academic papers. PET complements kernel hardening by preventing error triggering at the early stage of exploitation.

Syscall Filtering & Kernel Debloating. System fitering limits its vulnerable system calls that are available to userspace applications. Representative works include Lock-in-Pop [105] which only allows programs to use system calls with fewer vulnerabilities, Confine [106] which limits the container image to using system calls that are necessary, C2C [107] which filters system calls based on the application configuration, and Temporal specialization filters [108] which disables unused system calls at different phases of server applications. Kernel code debloating also limits the kernel code that is available to userspace applications. FACE-CHANGE [109], KASR [110], and SHARD [111] only provide the necessary kernel code for specific userspace applications. These techniques either have assumptions on the userspace applications or can negatively affect the kernel functionality. In comparison, PET is invisible to userspace and only skips the execution of error sites if the error is about to be triggered, without affecting the overall functionality of the kernel.

Error Recovery. Through techniques like checkpoint and rollback, the system state can be recovered after errors. Landmarks include Akeso [65] which logs and rolls back all state changes including modifications and dependencies to the runtime kernel, ASSURE [66] which optimizes heavy logging by predefining rescue points, FGFT [67] which focuses on on rolling back isolated buggy drivers, Phoenix [68] which records both external device and internal software states for embedded system, and Ares [69] synthesizes multiple JAVA JVM exception handlers to select the proper one for recovery. Compared with these works, PET employs a straight checkpoint-restore approach which is limited and only serves as a temporary solution.

10 Discussion & Future Works

Alternatives of eBPF ecosystem. We chose eBPF ecosystem to design our solution because eBPF program is safer, more expressive, and requires the privilege to install. Aside from these advantages, there are alternatives of eBPF ecosystem in kernel. For instance, we can inject Kprobe modules to the error sites and checkpoint-restore sites, storing raw data in kernel memory, and relying on a userspace agent process for data sharing among different modules. In comparison, using Kprobe modules needs to reinvent the wheels already provided by the eBPF ecosystem. This requires more engineering efforts and incurs higher performance overhead due to the additional kernel-user communication for data sharing.

Support More Error Types & Manual Efforts. PET is designed to be extensible. It currently supports five most common errors reported by sanitizers. The remaining error types include null pointer dereference, which can be detected

by checking if the pointer is equal to NULL; wild access and user-memory access, which can be detected by checking if the address is in kernel space or not; and memory leak which can be detected by reusing the use-after-free sweeper to check if there are still pointers referencing the objects of interest. In the future, we plan to extend PET to support them. If readers wish to contribute, sample templates provided in the Appendix can be used as a reference.

PET is unable to handle out-of-bound access scenarios where the start address already goes beyond the range, as PET relies on the start address to determine the legit object size. To accommodate this, manual efforts are needed. One possible scenario might be specifying that the start address is calculated from `addr+offset` and `offset` is extensively large, leading to a wrong start address. As such, eBPF programs can be installed to examine if `addr+offset` is correct or not at runtime.

Potential Bypassing Methods. PET can be bypassed by attackers in the following scenarios: (1) rootkits injected at boot time before PET is deployed, (2) physical attackers through plugging in malicious peripherals like USB devices [112] and not triggering errors, (3) exploiting errors that cannot be detected by sanitizers and thus not covered by PET.

11 Conclusion

This work presents PET which prevents discovered errors from being triggered before patches are available. We demonstrated its effectiveness against the five most common errors that sanitizers can report. PET is lightweight and scalable in the presence of multiple errors with acceptable memory consumption.

References

[1] "Super Long-term Kernel Support [LWN.net] — lwn.net." https://lwn.net/Articles/749530/. [Accessed 06-Feb-2023].

[2] "The Perenial Nuclear Power Plantquot; example [LWN.net] — lwn.net." https://lwn.net/Articles/106179/. [Accessed 06-Feb-2023].

[3] M. Erdos, S. Ainsworth, and T. M. Jones, "MineSweeper: a "clean sweep" for drop-in use-after-free prevention," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*.

[4] H. Cho, J. Park, A. Oest, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G. Ahn, "ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*.

[5] L. He, H. Hu, P. Su, Y. Cai, and Z. Liang, "FreeWill: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries," in *31st USENIX Security Symposium, SEC 2022, Boston, MA, USA, August 10-12, 2022*.

[6] C. Liu, Y. Chen, and L. Lu, "KUBO: Precise and Scalable Detection of User-triggerable Undefined Behavior Bugs in OS Kernel," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*.

[7] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data Race Fuzzing for Kernel File Systems," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*.

[8] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for c," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*.

[9] M. Cao, X. Hou, T. Wang, H. Qu, Y. Zhou, X. Bai, and F. Wang, "Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*.

[10] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "kMVX: Detecting Kernel Information Leaks with Multi-variant Execution," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*.

[11] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*.

[12] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel," in *31st USENIX Security Symposium, SEC 2022, Boston, MA, USA, August 10-12, 2022*.

[13] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. A. Sani, "Demystifying the Dependency Challenge in Kernel Fuzzing," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*.

[14] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability," in *31st USENIX Security Symposium, SEC 2022, Boston, MA, USA, August 10-12, 2022*.

[15] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities," in *29th USENIX Security Symposium, SEC 2020, August 12-14, 2020*.

[16] Y. Chen and X. Xing, "SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*.

[17] Y. Chen, Z. Lin, and X. Xing, "A Systematic Study of Elastic Objects in Kernel Exploitation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS 2020, Virtual Event, USA, November 9-13, 2020*.

[18] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities," in *27th USENIX Security Symposium, SEC 2018, Baltimore, MD, USA, August 15-17, 2018*.

[19] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities," in *28th USENIX Security Symposium, SEC 2019, Santa Clara, CA, USA, August 14-16, 2019*.

[20] Y. Lee, C. Min, and B. Lee, "ExpRace: Exploiting Kernel Races through Raising Interrupts," in *30th USENIX Security Symposium, SEC 2021, August 11-13, 2021*.

[21] Z. Lin, Y. Wu, and X. Xing, "DIRTYCRED: Escalating Privilege in Linux Kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*.

[22] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for Security," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*.

[23] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. B. Abu-Ghazaleh, "SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning," in *30th USENIX Security Symposium, SEC 2021, August 11-13, 2021*.

[24] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing," in *31st USENIX Security Symposium, SEC 2022, Boston, MA, USA, August 10-12, 2022*.

[25] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS 2021, Virtual Event, Republic of Korea, November 15 - 19, 2021*.

[26] "The Kernel Address Sanitizer (KASAN) x2014; The Linux Kernel documentation — kernel.org." https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html. [Accessed 06-Feb-2023].

[27] "The Kernel Memory Sanitizer (KMSAN) x2014; The Linux Kernel documentation — docs.kernel.org." https://docs.kernel.org/next/dev-tools/kmsan.html. [Accessed 06-Feb-2023].

[28] "The Kernel Concurrency Sanitizer (KCSAN) x2014; The Linux Kernel documentation — kernel.org." https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html. [Accessed 06-Feb-2023].

[29] "eBPF for Windows: Main Page — microsoft.github.io." https://microsoft.github.io/ebpf-for-windows/. [Accessed 06-Feb-2023].

[30] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin, "Undo Workarounds for Kernel Bugs," in *30th USENIX Security Symposium, SEC 2021, August 11-13, 2021*.

[31] "The More You Know, The More You Know You Dont Know — googleprojectzero.blogspot.com." https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html. [Accessed 07-Feb-2023].

[32] "2022 0-day In-the-Wild Exploitationso far — googleprojectzero.blogspot.com." https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html. [Accessed 07-Feb-2023].

[33] "In-the-Wild Series: Android Post-Exploitation — googleprojectzero.blogspot.com." https://googleprojectzero.blogspot.com/2021/01/in-wild-series-android-post-exploitation.html. [Accessed 07-Feb-2023].

[34] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, Vienna, Austria, October 24-28, 2016*.

[35] D. Mu, Y. Du, J. Xu, J. Xu, X. Xing, B. Mao, and P. Liu, "POMP++: Facilitating Postmortem Program Diagnosis with Value-Set Analysis," *IEEE Trans. Software Eng.*, vol. 47, no. 9, pp. 1929–1942, 2021.

[36] "REPT: Reverse Debugging of Failures in Deployed Software," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*.

[37] X. Ge, B. Niu, and W. Cui, "Reverse Debugging of Kernel Failures in Deployed Systems," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*.

[38] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, "AURORA: Statistical Crash Analysis for Automated Root Cause Explanation," in *29th USENIX Security Symposium, SEC 2020, August 12-14, 2020*.

[39] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape." BlueHat, 2019.

[40] "Variable-length arrays and the max() mess [LWN.net] — lwn.net." https://lwn.net/Articles/749064/. [Accessed 06-Feb-2023].

[41] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "TypeSan: Practical Type Confusion Detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security,CCS 2016, Vienna, Austria, October 24-28, 2016*.

[42] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "HexType: Efficient Detection of Type Confusion Errors for C++," 2017.

[43] A. Popov, "Alexander Popov's Blog." https://a13xp0p0v.github.io/, 2023.

[44] "UBSan: run-time undefined behavior sanity checker [LWN.net] — lwn.net." https://lwn.net/Articles/617364/. [Accessed 06-Feb-2023].

[45] "Kernel Thread Sanitizer (KTSAN) — google.github.io." http://google.github.io/kernel-sanitizers/KTSAN.html. [Accessed 06-Feb-2023].

[46] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *Proceedings of the workshop on binary instrumentation and applications*, pp. 62–71, 2009.

[47] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, "Syrup: User-Defined Scheduling Across the Stack," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*.

[48] "GitHub - xdp-project/xdp-tutorial: XDP tutorial — github.com." https://github.com/xdp-project/xdp-tutorial. [Accessed 07-Feb-2023].

[49] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing," in *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*.

[50] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. B. Butler, "LBM: A Security Framework for Peripherals within the Linux Kernel," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*.

[51] Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, *et al.*, "XRP:In-Kernel Storage Functions with eBPF," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 22)*.

[52] S. Park, D. Zhou, Y. Qian, I. Calciu, T. Kim, and S. Kashyap, "Application-Informed Kernel Synchronization Primitives," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*.

[53] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li, "RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices," in *31st USENIX Security Symposium, SEC 2022, Boston, MA, USA, August 10-12, 2022*.

[54] "Cilium - Linux Native, API-Aware Networking and Security for Containers — cilium.io." https://cilium.io/. [Accessed 07-Feb-2023].

[55] "Tracing: Attach eBPF Programs to Kprobes [LWN.net] — lwn.net." https://lwn.net/Articles/636224/. [Accessed 06-Feb-2023].

[56] "syzbot — syzkaller.appspot.com." https://syzkaller.appspot.com/. [Accessed 06-Feb-2023].

[57] D. Mu, Y. Wu, Y. Chen, Z. Lin, C. Yu, X. Xing, and G. Wang, "An In-depth Analysis of Duplicated Linux Kernel Bug Reports," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*.

[58] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*.

[59] syzbot, "UBSAN: shift-out-of-bounds in __qdisc_calculate_pkt_len." https://syzkaller.appspot.com/bug?id=70c77abc177053bca3b6ce82cefe5f05ad67c9f2.

[60] syzbot, "KASAN: use-after-free read in route4_get." https://syzkaller.appspot.com/bug?id=5bb09c0c5b65ab2ce628ba26fe7cbd06144bd952.

[61] syzbot, "KMSAN: uninit-value in tcp_recvmsg." https://syzkaller.appspot.com/bug?id=2039c557a4f369ad05baf0c6d0c9b9b8caf3acd5.

[62] syzbot, "KCSAN: data-race in tcp_send_challenge_ack / tcp_send_challenge_ack." https://syzkaller.appspot.com/bug?id=f6e95af74472292ab1c50af3d6ac36cd4a683432.

[63] B. Kasikci, C. Zamfir, and G. Candea, "Data Races vs. Data Race Bugs: Telling the Difference with Portend,"

[64] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*.

[65] A. Lenharth, V. S. Adve, and S. T. King, "Recovery Domains: An Organizing Principle for Recoverable Operating Systems," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*.

[66] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "ASSURE: Automatic Software Self-healing Using Rescue Points," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*.

[67] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*.

[68] R. Smith and S. Rixner, "Surviving Peripheral Failures in Embedded Systems," in *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*.

[69] T. Gu, C. Sun, X. Ma, J. Lü, and Z. Su, "Automatic Runtime Recovery via Error Handler Synthesis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*.

[70] K. Boos, E. D. Vecchio, and L. Zhong, "A Characterization of State Spill in Modern Operating Systems," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*.

[71] P. authors, "Open Sourced Implementation." https://github.com/purplewall1206/PET.

[72] "Add Code-generated BPF Object Skeleton Support [LWN.net] — lwn.net." https://lwn.net/Articles/806911/. [Accessed 06-Feb-2023].

[73] "GitHub - Markakd/LLVM-O0-BitcodeWriter: patch for LLVM to generate O0 bitcode — github.com." https://github.com/Markakd/LLVM-O0-BitcodeWriter. [Accessed 07-Feb-2023].

[74] A. Popov, "CVE-2017-2636: Exploit the race condition in the n_hdlc Linux kernel driver." https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html.

[75] J. Jorn, "Issue 2247: Linux: unix GC memory corruption by resurrecting a file reference through RCU." https://bugs.chromium.org/p/project-zero/issues/detail?id=2247.

[76] "UBSAN: shift-out-of-bounds in dummy_hub_control." https://syzkaller.appspot.com/bug?id=b5b251b9bcc4653c39164dfee969dafb903ae25e.

[77] syzbot, "KASAN: slab-out-of-bounds write in watch_queue_set_filter." https://syzkaller.appspot.com/bug?id=797c55d2697d19367c3dabc1e8661f5810014731.

[78] syzbot, "KASAN: use-after-free read in vb2_mmap." https://syzkaller.appspot.com/bug?extid=be93025dd45dccd8923c.

[79] syzbot, "KASAN: stack-out-of-bounds write in bitmap_from_arr32." https://syzkaller.appspot.com/bug?id=2c09122a1f7edf61aa6fb5dbb6cd19766b5daaa1.

[80] "google-sheet More Evaluation Results." https://tinyurl.com/yv9spkpp. [Accessed 07-Feb-2023].

[81] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective Memory Protection for Kernel and User Space," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*.

[82] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code Re-Randomization," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*.

[83] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An Investigation of the Android Kernel Patch Ecosystem," in *30th USENIX Security Symposium, SEC 2021, August 11-13, 2021*.

[84] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in Linux (experience paper)," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*.

[85] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux Bug Fixing Patches," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*.

[86] "Applying Patches To The Linux Kernel x2014; The Linux Kernel documentation — kernel.org." https://www.kernel.org/doc/html/v4.10/process/applying-patches.html. [Accessed 07-Feb-2023].

[87] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic Rebootless Kernel Updates," in *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*.

[88] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, "Automatic Hot Patch Generation for Android Kernels," in *29th USENIX Security Symposium, SEC 2020, August 12-14, 2020*.

[89] "Randomizing structure layout [LWN.net] — lwn.net." https://lwn.net/Articles/722293/. [Accessed 06-Feb-2023].

[90] "mm: SLUB Freelist randomization [LWN.net] — lwn.net." https://lwn.net/Articles/688749/. [Accessed 06-Feb-2023].

[91] "Function Granular KASLR [LWN.net] — lwn.net." https://lwn.net/Articles/824307/. [Accessed 06-Feb-2023].

[92] "slub: Improve bit diffusion for freelist ptr obfuscation - Patchwork — keescook." https://patchwork.kernel.org/project/linux-mm/patch/202003051623.AF4F8CB@keescook/. [Accessed 06-Feb-2023].

[93] "Kernel Self Protection Project - Linux Kernel Security Subsystem — kernsec.org." https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project. [Accessed 06-Feb-2023].

[94] "grsecurity - How AUTOSLAB Changes the Memory Unsafety Game — grsecurity.net." https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game. [Accessed 06-Feb-2023].

[95] "Hardened Usercopy [LWN.net] — lwn.net." https://lwn.net/Articles/695991/. [Accessed 06-Feb-2023].

[96] "grsecurity — grsecurity.net." https://grsecurity.net/. [Accessed 07-Feb-2023].

[97] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in *Proceedings of the 23rd USENIX Security Symposium, SEC 2014, San Diego, CA, USA, August 20-22, 2014*.

[98] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev, "RedLeaf: Isolation and Communication in a Safe Operating System," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*.

[99] J. Criswell, N. Dautenhahn, and V. S. Adve, "Virtual Ghost: Protecting Applications from Hostile Operating Systems,"

[100] J. Criswell, N. Dautenhahn, and V. S. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*.

[101] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. S. Adve, "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*.

[102] N. Roessler, L. Atayde, I. Palmer, D. P. McKee, J. Pandey, V. P. Kemerlis, M. Payer, A. Bates, J. M. Smith, A. DeHon, and N. Dautenhahn, "μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts," in *RAID '21: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 6-8, 2021*.

[103] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*.

[104] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*.

[105] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos, "Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*.

[106] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*.

[107] S. Ghavamnia, T. Palit, and M. Polychronakis, "C2C: Fine-grained Configuration-driven System Call Filtering," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*.

[108] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*.

[109] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*.

[110] Z. Zhang, Y. Cheng, S. Nepal, D. Liu, Q. Shen, and F. A. Rabhi, "KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels," in *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*.

[111] M. Abubakar, A. Ahmad, P. Fonseca, and D. Xu, "SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening," in *30th USENIX Security Symposium, SEC 2021, August 11-13, 2021*.

[112] K. Nohl and J. Lell, "BadUSB On accessories that turn evil." Black-Hat, 2014.

[113] syzbot, "KASAN: slab-out-of-bounds write in sha512_final." https://syzkaller.appspot.com/bug?id=e4be30826c1b7777d69a9e3e20bc7b708ee8f82c.

[114] syzbot, "KMSAN: kernel-infoleak in _copy_to_iter (6)." https://syzkaller.appspot.com/bug?id=e476b01dd5a1075a281c26069ebf677b019bf6d8.

[115] syzbot, "KCSAN: data-race in netlink_recvmsg / netlink_recvmsg (5)." https://syzkaller.appspot.com/bug?id=cb2264a0f3b303a24e4c4a88752d551e35bae757.

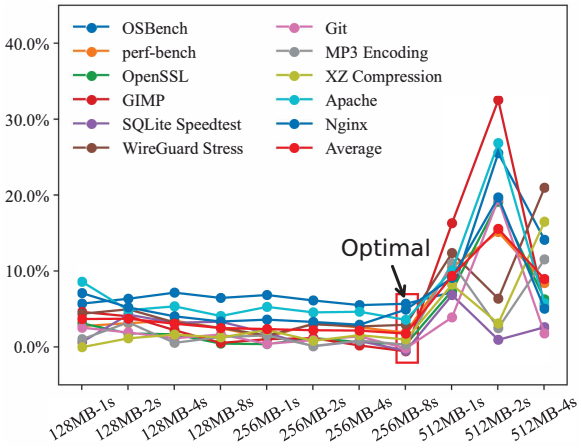

Figure 5: Impact of configuration for entire-sweeping mode in the use-after-free policy with CVE-2021-4154 as the testbed. The optimal configuration is scanning 256 MB per 8s.

[116] syzbot, "KCSAN: data-race in netlink_getname / netlink_insert (4)." https://syzkaller.appspot.com/bug?id=a834b993b63ed43938194af3accb08c0a5042877.

A More Implementation Details

eBPF Programs. In the prevention policy for use-after-free errors, the sweeper periodically scans physical memory for dangling pointers. Although all physical memory is mapped in the kernel, there are some unmapped holes in between such as `0xffff8880c0079000` in `Zone_Normal` and accessing these holes can cause kernel panic. Therefore, the sweeper only scans mapped pages using `for_each_zone` instead of straightly traversing all page frames. Besides, to avoid the sweeper from mistakenly treating object addresses stored in the BPF map as dangling pointers, we negate every bit in the object address in the BPF maps. For instance, `0xffffffff80123456` is negated to `0x000000007fedcba9`. Since all kernel object addresses are larger than `0xffffffff80000000`, the negated address won't be treated as a valid kernel pointer value.

Static Analysis. To optimize the sweeper, `PET` only selects slab caches that potentially contain dangling pointers for sweeping. Our static analysis includes: (1) use `GetElementPtr` instruction to extract the type of allocated objects at the allocation site, (2) add kernel structures that contain a field in the pointer type of the allocated objects into our candidate set - these fields are potential dangling pointers, (3) analyze `Cast` instructions to identify additional structures that can be cast to or from structures in the candidate set, and add them to the set, (4) repeat above steps until the candidate set reaches a fixed point. With all the candidate structures determined, we further determine which slab caches can store them and only these caches are selected for sweeping. This analysis is sound. In the worst case where all slab caches are swept, the sweeper is degraded to the entire-sweeping mode.

B More Evaluation Results

The optimal configuration for the use-after-free sweeper is to scan 256 MB per 8s which is obtained from Figure 5. More complete effectiveness results can be found in Table 4.

C More eBPF Templates

Due to the space limit, we only showed the partial eBPF template for integer underflow/overflow in Section 5.1. More detailed eBPF templates are in [71].

CVE/SYZ ID	Sites for eBPF Installation	Action & Triggering Condition	Effectiveness (PET/[30])	Time Window (days)
Integer Underflow/Overflow				
70c77ab [59]	`__qdisc_calculate_pkt_len`	`$eax>>$cl == $rax>>$cl & $cl<32 ? false : true`	●/ ◐	415
b5b251b [76]	`dummy_hub_control+0x3f` (spinlock) `dummy_hub_control+0x225`	`lock_map[pid] = $rdi` `$eax<<$edx == $rax<<$edx & $edx<32 ? false : true`	●/ ◐	79
Out-of-bound Access on Stack				
2022-1015	`nft_do_chain+0x243`	`$rdi ∈ [$rsp+0x50, $rsp+0xa0)? false : true`	●/ ◐	147
2c09122 [79]	`ethnl_parse_bitset+0x45f`	`$rdi+$rdx ∈ [$rdi, $rdi+0x40*8)? false : true`	●/ ◐	104
Out-of-bound Access on Global and Static Region				
2017-18344	`show_timer+0x81`	`$rdx ∈ [0xffffffff822479c0, 0xffffffff822479d8)? false : true`	●/ ◐	90
Out-of-bound Access on Buddy System Heap				
2017-7308	`tpacket_rcv+0x2ff`	`$rdi+$rsi*$rcx ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/ ◐	1015
2022-27666	`null_skcipher_crypt+0x4b`	`$rdi+$rdx ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/ ◐	17
Out-of-bound Access on vmalloc Heap				
2020-14386	`tpacket_rcv+0x21a` (spinlock) `tpacket_rcv+0x6f6`	`lock_map[pid] = $rdi` `%rax ∈ [start($r10), start($r10)+len($r10))? false : true`	●/ ◐	39
Out-of-bound Access on SLAB/SLUB Heap				
2010-2959	`bcm_sendmsg.cold+0x568`	`$rdi ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/ ◐	13
2021-22555	`xt_compat_target_from_user.cold+0x23`	`$rdi+$rdx ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/ ◐	86
2021-43276	`tipc_crypto_msg_rcv.cold+0x6d`	`$rdi+$rdx ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/ ◐	12
2022-34918	`nft_set_elem_init+0x3e`	`$rdi+$rcx ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/ ◐	38
2016-6187	`apparmor_setprocattr+0x8f`	`$rdi ∈ [start($rdi), start($rdi)+len($rdi))? false : true`	●/ ◐	111
2017-7184	`xfrm_replay_advance+0x250`	`$rbx+0x18 ∈ [start($rbx), start($rbx)+len($rbx))? false : true`	●/ ◐	108
2022-0185	`legacy_parse_param+0x27e`	`$rbp ∈ [start($r12), start($r12)+len($r12))? false : true`	●/ ◐	0
797c55d [77]	`watch_queue_set_filter+0x81` (alloc) `watch_queue_set_filter+0x78d`	`alloc_map[pid]=$rdi` `$r15+0x8 ∈ [start($r15), start($r15)+len($r15))? false : true`	●/ ◐	344
e4be308 [113]	`sha512_final+0x34a/0x3e0`	`$r12+$rax ∈ [start($r15), start($r15)+len($r15))? false : true`	●/ ◐	30
Use-After-Free				
2019-18683	`__vb2_queue_free+0x13e` (free) `vid_cap_buf_queue+0x49` (use)	`map ⋃ $rdi; full_sweep(0, 16GB)` `$rbp+0x3a8 ∈ map ? true: false`	●/ ◐	29
2021-23134	`nfc_llcp_local_put` (free) `nfc_llcp_sock_unlink` (use)	`map ⋃ $rdi; full_sweep(0, 16GB)` `$rdi ∈ map, ? true: false`	●/ ◐	201
2021-4154	`put_fs_context+0xec` (free) `filp_close` (use)	`map ⋃ $rdi; full_sweep(0, 16GB)` `$rdi ∈ map ? true: false`	●/ ◐	233
2022-2586	`nft_obj_destroy+0x3f` (free) `nf_tables_fill_setelem.isra.0+0x140` (use)	`map ⋃ $rdi; selective_sweep(kmalloc-256, 0x20)` `$rbx+$rax ∈ map ? true: false`	●/ ◐	97
2017-8824/ 2020-16119	`ccid_hc_rx_delete+0x2e` (free) `ccid_hc_rx_delete+0x2e` (use)	`map ⋃ $rsi; selective_sweep(DCCPv6, 0x628)` `$rdi ∈ map ? true: false`	●/ ◐	128
2021-3715/ 5d5bb09c [60]	`__route4_delete_filter+0x3c` (free) `route4_get+0x58` (use)	`map ⋃ $rdi, selective_sweep(kmalloc-192, 0x28)` `$rax+0x40 ∈ map ? true: false`	●/ ◐	59

be93025d [78]	`__vb2_queue_free+0x13e` (free)
	`vb2_mmap+0x52` (mutex)
	`vb2_mmap+0xa29` (use)
		`map ⋃ $rdi; full_sweep(0, 16GB)`
		`mutex_lock[pid]=$rdi`
		`r8 ∈ map ? true: false`
			●/ ◐	73

2022-2588	`__route4_delete_filter+0x3c` (free) `__route4_delete_filter+0x3c` (use)	`map ⋃ $rdi; selective_sweep(kmalloc-192, 0x28)` `$rdi ∈ map ? true : false`	●/ ◐	38
Uninitialized Access				
2039c557 [61]	`__sys_recvfrom` (create) `tcp_recvmsg+0xb8` (use)	`map[$rsp+8-200] = mem($rsp-0xc0, 0x60)` `map[$r13] == mem($r13, 0x60)? false : true`	●(default conservative) ◐(aggressive) / ◐	248
e476b01d [114]	`__alloc_slab+0x237` (create) `simple_copy_to_iter+0x11` (use)	`map[$rdi] = mem($rdi, 0x80)` `map[$rdi] == mem($rdi, 0x80)? false : true`	●(default conservative) ◐(aggressive) / ◐	111
Data Race				
2017-2636 [74]	`n_hdlc_send_frames+0x118` (write) `n_hdlc_tty_ioctl+6b` (write)	`P($rbp+0x310); V($rbp+0x310)` `P($rsp); V($rsp)`	●/ ◐	40

2021-4083 [75]	`unix_stream_read_generic+0xeb` (spinlock)
	`unix_stream_read_generic+0x120` (mutex)
	`unix_stream_read_generic+0x138` (read)
	`unix_gc+0x33` (spinlock)
	`unix_gc+0x28e` (write)
		`lock_map[pid]=$rdi`
		`mutex_lock[pid]=$rdi`
		`P($r13); V($r13)`
		`lock_map[pid]=$rdi`
		`P($rsp+0x38); V($rsp+0x38)`
			●/ ◐	224

f6e95af7 [62]	`tcp_send_challenge_ack.constprop.0+0x5d` (read) `tcp_send_challenge_ack.constprop.0+0x7b` (write)	`P($rip+0x2108765); V($rip+0x2108765)` `P($rip+0x2108765); V($rip+0x2108765)`	N/A / ◐	178
cb2264a [115]	`tcp_send_challenge_ack.constprop.0+0x65` (read) `tcp_send_challenge_ack.constprop.0+0x7b` (write)	`P($rip+0x2108765); V($rip+0x2108765)` `P($rip+0x2108765); V($rip+0x2108765)`	N/A / ◐	340

a834b99 [116]	`netlink_getname+0x44` (read)
	`netlink_insert+0x3c` (lock_sock)
	`netlink_insert+0x87` (write)
		`P($rbp+0x310); V($rbp+0x310]);`
		`lock_map[pid]=$rdi`
		`P($rbp+0x310); V($rbp+0x310]);`
			N/A / ◐	35

Table 4: Complete results for the effectiveness of PET. ● indicates that all three criteria are satisfied; ◐ means criteria ❷ (*i.e.*, no false alarm) is not met; N/A means Proof-of-concept is unavailable.