RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections

Kyle Zeng
Arizona State University
zengyhkyle@asu.edu

Zhenpeng Lin
Northwestern University
zplin@u.northwestern.edu

Kangjie Lu
University of Minnesota
kjlu@umn.edu

Xinyu Xing
Northwestern University
xinyu.xing@northwestern.edu

Ruoyu Wang
Arizona State University
fishw@asu.edu

Adam Doupé
Arizona State University
doupe@asu.edu

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Tiffany Bao
Arizona State University
tbao@asu.edu

ABSTRACT

Leveraging a control flow hijacking primitive (CFHP) to gain root privileges is critical to attackers striving to exploit Linux kernel vulnerabilities. Such attack has become increasingly elusive as security researchers propose capable kernel security mitigations, leading to the development of complex (and, as a trade-off, brittle and unreliable) attack techniques to regain it. In this paper, we obviate the need for complexity by proposing *RetSpill*, a powerful yet elegant exploitation technique that employs user space data *already present on the kernel stack* for privilege escalation.

RetSpill exploits the common practice of temporarily storing data on the kernel stack, such as when preserving user space register values during a switch from the user space to the kernel space. We perform a systematic study and identify four common practices that spill user space data to the kernel stack. Although this practice is perfectly within the kernel's security specification, it introduces a new exploitation path when paired with a control flow hijacking (CFH) vulnerability, enabling RetSpill to turn such vulnerabilities directly into privilege escalation reliably. Moreover, RetSpill can bypass many defenses currently deployed in the Linux kernels. To demonstrate the severity of this problem, we collected 22 real-world kernel vulnerabilities and built a semi-automated tool that abuses intentionally-stored, on-stack user space data for kernel exploitation in a semi-automated fashion. Our tool generated end-to-end privilege escalation exploits for 20 out of 22 CFH vulnerabilities. Finally, we propose a new mechanism to defend against the attack.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; *Software security engineering*.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623220

KEYWORDS

OS Security; Kernel Exploitation; Privilege Escalation

ACM Reference Format:
Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2023. RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623220

1 INTRODUCTION

As Linux becomes increasingly ubiquitous in cloud services, IoT devices, and mobile phones[1], Linux kernel vulnerabilities have emerged to be a major threat to computer security. The successful exploitation of a Linux kernel vulnerability can allow attackers to elevate privileges and induce critical issues in victim systems.

Among Linux kernel vulnerabilities, those giving attackers the ability to carry out Control Flow Hijack (CFH) is a prevalent and serious type of flaw. Attackers exploit such vulnerabilities to achieve *control flow hijacking primitives* (CFHP) (e.g., by overwriting a function pointer stored in the kernel heap), then use creative techniques to bypass modern defenses (e.g., SMAP) before finally escalating the attacker's privilege level from normal users to root. CFH is popularly used for Linux kernel exploitation *despite modern defenses*. For example, 15 out of 16 public real-world Linux kernel exploits [23] reported to 2022 Google's kCTF kernel bug bounty program are control-flow hijacking attacks.

However, there is a significant gap from CFHP to a full kernel privilege escalation exploit. In most modern cases, CFHP stems from flaws on the heap (e.g., a function pointer overwrite stemming from a use-after-free), instead of the stack. This is problematic for exploitation, as building and triggering an ROP payload on the heap is hard. Current solutions to this problem attempt to pivot the stack pointer to a fake stack on the heap [19, 29, 46] or in the flat physical mapping [70]. Such techniques are brittle because they rely on either specific kernel memory layouts, the existence of precise bespoke gadgets, or additional primitives (such as register control) that are not always granted alongside the CFHP. Unfortunately, for attackers, the decline of straightforward stack

[1] And perhaps, this year, on the desktop.

overflow vulnerabilities (and the adoption of effective mitigations, such as stack canaries), leaves no other option than to accept this complexity, brittleness, and additional primitive requirements.

In this paper, we bridge this exploitation gap by developing a general technique to prepare ROP payloads on the stack. Our technique exploits the *intended design* of the kernel stack: Linux's kernel stack is designed to save various forms of data, *including user-controlled data from user space.* Attackers can leverage this intended storage to store carefully-crafted malicious data in kernel space, that they can access after triggering vulnerabilities through stack-pivoting.

In studying the implications of these capabilities, we realized that the Linux kernel stack design can be utilized as an attack surface for malicious attackers. We built on our discovery to propose both a novel exploitation technique and a new mitigation to defend against it. Our technique, RetSpill, enables exploits that bypass most existing kernel security mitigations (and emerging mitigations such as Function Granular Kernel Address Space Layout Randomization [34]) with no other requirements (such as register control or specific memory configuration requirements that limit other techniques). Importantly, RetSpill can be semi-automated, demonstrating that the bar for successful Linux kernel attacks is significantly lower than where it is currently assumed to be by the security community.

To comprehensively understand RetSpill's potential impact, we identify the scenarios where user-controlled data appears on the kernel stack (§4.2), and we quantify the size of user-controlled data to check if they are sufficient for a privilege escalation exploit. We discover four different scenarios (one of them is discovered by another researcher in parallel to this research [54]), three of which are Linux kernel's intended operations which are inevitable by design. We also find that, Linux system calls typically leave enough user-controlled data for attackers to launch a privilege escalation exploit. Specifically, our measurement shows that, given Linux kernel's intended operations, the user-controlled data in a Linux system call can hold 11 ROP gadgets on average. Even worse, we found that given Linux kernel's current design, 11 ROP gadgets are sufficient for privilege escalation (§4.3). Through clever downstream applications, one can take advantage of the 11 gadgets to obtain unlimited Arbitrary Read/Write/Exec primitives in kernel space and can even call and retrieve return values from kernel functions.

We show the power of RetSpill by applying it to 22 real-world vulnerabilities, demonstrating its ability to achieve successful exploitation on 21 of them, with fewer primitive requirements and more reliability than existing public exploits for these vulnerabilities. Critically, because the technique is so applicable, and very few mitigations are effective against it to any extent, we were able to implement RetSpill as a semi-automated pipeline that directly synthesized 20 of these exploits. The only manual effort needed is providing the CFHP and integrating the output of the pipeline into the exploit.

Finally, we investigate RetSpill from a defensive perspective, checking it against all existing commonly-adopted protections (which cannot defend against it) and proposing new protection to mitigate the impact of this dangerous exploitation technique and re-raise the bar of Linux kernel exploitation.

In summary, this paper makes the following contributions:

- We identify the kernel stack as an attack surface in Control Flow Hijacking exploitation and study the methods to place user-controlled data on the kernel stack.
- By accessing user-controlled data on the kernel stack, we devise a novel and powerful exploitation technique, RetSpill, that can bypass commonly-adopted protections in the Linux kernel and enable powerful exploitation of Control Flow Hijacking vulnerabilities.
- We evaluate RetSpill on 22 real-world vulnerabilities and demonstrate its ability to break the security boundary between user space and kernel space.
- We propose a new protection to harden Linux kernel security. Combined with existing protections, the new defense can mitigate the impact of the powerful RetSpill exploitation technique.

To foster future research, we released the artifacts, including RetSpill and all the experiment data[2]. This will benefit the community by letting researchers understand the severity of RetSpill and inspiring new protections to harden the Linux kernel.

Ethics. We have contacted the Linux kernel security team and responsibly disclosed the exploitation technique. We are collaborating with the Linux community on the contribution of our defense. The kernel security team has given us permission to publish this research to the community.

2 BACKGROUND

In this section, we present the technical background of Linux kernel security, including primitives, vulnerabilities, attacks, and modern defenses.

Linux Kernel Exploits. Typically, the goal of a Linux kernel exploit is to use kernel vulnerabilities to escalate privilege from a less privileged user (*e.g.*, `nobody`) to `root`. Compared to user space-level exploitation, where, for example, an attack is considered successful once a shell is achieved, Linux kernel exploitation needs to additionally consider safely returning to the user space without triggering a kernel panic or killing the escalated user space process.

Exploitation Primitives and the Control Flow Hijacking Primitive. An *exploitation primitive* represents a machine state transition where a security policy is violated. With a *control flow hijacking primitive* (CFHP), the attacker gains control of the kernel's program counter (PC). In this paper, we will use "CFHP" and the "PC-control" terminology interchangeably. Note that we do not assume the capability of controlling registers other than the program counter, or controlling the contents pointed to by any register.

Control Flow Hijacking Exploits. A control flow hijacking (CFH) exploit starts with CFHP. It amplifies CFHP to better primitives, such as Return Oriented Programming (ROP) and shellcode execution, and eventually obtains privilege escalation. Before modern defenses, obtaining CFHP in the Linux kernel directly led to privilege escalation, but it is no longer this simple in the modern era.

Existing Protections Against CFH Attacks. With increasingly powerful defenses developed by the Linux kernel, exploiting

[2] https://github.com/sefcom/RetSpill

Technique	Required Primitives	Data Region	Reliable*	Payload Rewrite	Still Applicable	Mitigation
ret2usr [37]	CFHP	user space	✓	✓	✗	SMEP
ret2dir [31]	CFHP+physmap leak	physmap	✗‡	✓	✗	NX-physmap
pivot to user [18]	CFHP+codebase leak	user space	✓	✓	✗	SMAP
change CR4 [49]	CFHP+codebase leak+rdi control	user space	✗§	✓	✗	CR-Pinning
run_cmd [24]	CFHP+codebase leak+heap info leak+rdi control	kernel heap	✓	✓	✗△	-
pivot to heap [19]	CFHP+codebase leak+heap info leak+register control	kernel heap	✓	✗	✓	-
KEPLER [70]	CFHP+codebase leak+physmap leak+register control	physmap	✗	✓	✓	-
set_memory_x [15]	CFHP+codebase leak+heap info leak+rdi&rsi control	kernel heap	✗§	✗	✓	-
RetSpill	CFHP+codebase leak	kernel stack	✓	✓	✓	-

Table 1: With the development of modern protections in the Linux kernel, exploiting CFH vulnerabilities requires many primitives and becomes brittle. Our work, RetSpill, is a powerful exploitation technique that requires fewer primitives and yet is still reliable. * Given all the required primitives, whether the technique can derive the target primitive without failures. ‡ ret2dir is reliable only if given a page frame info leak primitive. § need to obtain CFHP twice. △ run_cmd function can get inlined, thus not exist, in some kernel builds.

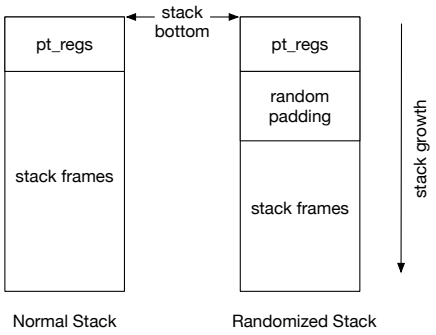

Figure 1: Kernel Stack Layout With and Without RANDKSTACK Protection

CFH vulnerabilities has become difficult. In a modern Linux kernel, obtaining CFHP does *not* imply the ability to obtain root or even carry out Return-Oriented-Programming (ROP) or arbitrary code execution because of modern kernel hardening techniques.

Most protections aim to prevent attackers from accessing user space data improperly. `SMEP` [47] and `KPTI` [64] prohibit executing user space code from kernel space. `SMAP` [12] disallows the kernel from reading/writing user space data directly. `NX-physmap` [31] marks the `physmap` region as not executable. These protections limit attackers' capability to turn CFHP into arbitrary code execution.

Other protections were designed to limit attackers' capabilities despite limited code execution: `CR Pinning` [63] prevents unauthorized modification to the CR4 register (which contains bits to enable certain mitigations), thus preventing attackers from disabling mitigations. `STATIC_USERMODE_HELPER` [35] prohibit abusing "user mode helpers" to perform privilege escalation. `RKP` [56] disallows directly modifying process credentials. `pt-rand` [14] invalidates the technique of directly modifying kernel page tables.

Notably, `RANDKSTACK` [53] is a protection that aims to randomize kernel stack layout, preventing attackers from exploiting use-of-uninitialized-variable vulnerabilities. Its change to kernel stack layout is shown in Figure 1. Similar to their use in user space applications, `STACK CANARY` mitigates stack-overflow vulnerabilities, removing the possibility of overwriting the return address (and higher memory addresses) to achieve ROP.

Bypassing exploitation protections. Because of strong stack protections, most modern CFHPs happen as a result of kernel heap misuse. Typically, an attacker overwrites (*e.g.*, via heap overflow) a function pointer in a victim object in the heap. The attacker then invokes a *trigger system call* that calls the overwritten function pointer and obtains CFHP. As direct shellcode execution is no longer viable (due to NX-physmap and SMEP), and stack control is elusive (due to the use of stack canaries and death of stack overflow vulnerabilities), a typical CFH exploit needs to combine the CFHP with other primitives, such as a heap address disclosure, to obtain stack control and start an ROP chain [13, 19, 29, 46] as shown in Table 1.

Imperfect bypass: "pivot to heap". Anecdotally, we observed that most modern kernel exploits use heap-related vulnerabilities and must prepare and trigger attack payloads on the heap. For example, all 15 public kCTF CFHP exploits pivot to a fake stack in the heap to achieve ROP. This has drawbacks: it does not support rewriting payloads directly because the payload is in the kernel heap, a region users do not have direct access. In other words, to execute a different payload, the attacker needs to swap the heap object containing the payload or even trigger the vulnerability again, which significantly reduces reliability for exploits that need to trigger different payloads to achieve privilege escalation [21, 33]. Additionally, these exploits rely on either specific kernel memory layouts, the existence of precise bespoke gadgets, or additional primitives (such as register control). As shown in Table 1, "pivot to heap" requires more primitives than RetSpill, which significantly increases the difficulty of developing exploits.

Kernel ROP. Kernel ROP, similar to that in user space, grants attackers the ability to perform arbitrary execution by chaining ROP gadgets together. Compared to user space ROP, kernel ROP has two additional requirements. First, leaked information in the ROP chain needs to be transferred back to user space. This can be done by using the fact that registers are not cleared during CPU privilege transition [38]. Attackers can load the leaked information into a register, perform a privilege transition back to user space, and obtain the leaked value. Second, ROP chains in kernel space need to end gracefully to avoid panic. Currently, there are three existing ways to achieve this: 1. return back to user space, such as using KPTI trampoline [38]; 2. sleep indefinitely [74], which freezes the current task; 3. kill the current task by invoking `do_task_dead` function.

Kernel Stack and System Calls. In the Linux kernel, each task has its own kernel space stack to facilitate the execution of system calls invoked from user space. The kernel stack is allocated as a fixed-size buffer at the creation of each kernel task [61]. The kernel stack layout is shown in Figure 1. When the user space invokes a system call, the program counter will switch to kernel space functions and the stack pointer will be set to the kernel stack. Then, the kernel will push the user space context, including all user space registers, onto the bottom of the stack, and invoke the corresponding system call handler. The saved user space context is named `pt_regs`. When returning back to user space, the kernel will restore the user space context and resume the execution of the user space program.

Due to the fixed-sized nature of the kernel stack, developers usually store large chunks of data in dynamically allocated memory regions (*e.g.*, heap) to avoid stack exhaustion. However, for performance considerations, some performance-critical system calls [66, 67] still store large amounts of data on the kernel stack.

3 THREAT MODEL

In our threat model, the target Linux kernel has all the protections as the model in Kepler [70], in addition to Function-Granular Kernel Address Space Layout Randomization (`FG-KASLR`) [34], which is a cutting-edge kernel hardening technique implemented recently.

Specifically, we allow the target Linux kernel enabling `SMEP`, `SMAP`, `KPTI`, `NX-physmap`, `CR Pinning`, `STATIC_USERMODE_HELPER`, `RKP`, `pt-rand`, `RANDKSTACK`, `STACK CANARY`, and `FG-KASLR`.

In terms of exploitation primitives, our model requires two primitives: control-flow hijacking and kernel image base address leakage. Both primitives are reasonable and can be achieved through a reasonable number of Linux vulnerabilities (Table 4). Our required primitives are fewer than all existing work (Table 1), as well as many latest real-world kernel exploitation techniques such as those published through Google kCTF [23].

4 EXPLOITATION TECHNIQUE DESIGN

A modern CFH exploit has two steps: 1) injecting user-controlled data into kernel memory, and 2) using this controlled data for ROP. Modern protections prevent attackers from directly accessing user space data, leaving physmap and kernel heap the only known choices for storing user-controlled ROP chains. The difficulty of accessing these two regions induces the requirement of additional exploitation primitives (such as register control or additional address disclosures) onto modern CFH exploits.

Our proposed technique, RetSpill, is based on the insight that potentially attacker-controlled user data is loaded, *by design*, onto the kernel stack during system calls. When PC-control is obtained, which happens inside the triggering system call after the attacker sets up the conditions for the Control Flow Hijacking Primitive (CFHP), the attacker can use this controlled user data readily on the kernel stack to launch code reuse attacks (*e.g.*, ROP).

However, this is not enough for full exploits: the individual resulting ROP attacks are limited, and our study of 22 kernel vulnerabilities (§7.1) shows that some triggering system calls have as few as seven controllable gadgets for ROP payload. This leads to RetSpill's other contribution: because stack layouts of functions are fixed during compilation, attackers can repeatedly invoke the triggering system call unlimited times to trigger different ROP payloads.

With this, RetSpill can obtain *unlimited* arbitrary read/write/exec in kernel space, thus completely breaking the security boundary between user and kernel space despite the presence of modern kernel protections. This is more severe than existing "single-shot" exploitation techniques [70]: as shown in §7.3, RetSpill can bypass more protections than existing techniques.

4.1 Data Spillage

Attacker-controlled user space data can spill onto the kernel stack either directly or indirectly.

Direct Data Spillage refers to the situation where user data is loaded directly onto the kernel stack in one system call. The `poll` system call shown in Listing 1 performs direct data spillage when it uses `copy_from_user` to copy user space data onto the kernel stack directly.

Indirect Data Spillage happens when data is loaded onto the kernel stack through multiple system calls. For example, with carefully crafted arguments, a call to the `open` system call stores data from user space to the kernel heap, and a subsequent call to `readlink` loads this controlled data to the kernel stack.

In this work, we mainly focus on systematically analyzing direct data spillage. We first investigate how user space data can flow into kernel space during system calls. We identify two methods: 1) user space registers and 2) user space memory. More specifically, when a user space program invokes a system call, its general-purpose registers contain data that can potentially flow into kernel space. During the lifetime of a system call, the kernel may need additional information from user space memory to complete the system call, which leads to userspace data flowing into kernel space.

To comprehensively discover direct user data spillage causes, we perform taint analysis on user space data (both user space registers and memory) and observe the traces on kernel stack in triggering system calls to identify data spillage causes as detailed in §6.2. By manually analyzing the taint analysis results, we identify three causes of direct user data spillage.

Depending on where user data is temporarily stored, there are potentially many variants of indirect data spillage methods. In this work, we empirically identify one general case of indirect data spillage where user data is temporarily stored on the kernel stack and shared across system calls because of uninitialized memory. We leave the systematic study of indirect data spillage as future work.

4.2 Data Spillage Sources

We identified four data spillage sources as follows:

Valid Data. In a Linux kernel, user space data is sometimes directly copied onto the kernel stack for performance reasons. Take the simplified `poll` system call handler in Listing 1 as an example. It copies a considerable amount (`0x1e0` bytes in our build) of user space data into `stack_pps`, which is on the kernel stack. If an attacker controls a `file` object in user space, when CFHP is obtained through the `file->f_op->poll` call, they will have control of `0x1e0` bytes on the kernel stack. The attacker can then use an `add rsp, X; ret`

```c
int do_sys_poll(...)
{
    long stack_pps[POLL_STACK_ALLOC/sizeof(long)];
    struct poll_list *walk = (struct poll_list *)stack_pps;

    // copy user space data onto kernel stack
    copy_from_user(walk->entries, ufds, sizeof(walk->entries);

    // invoke poll handler for each fd
    pfd = walk->entries;
    for(; ... ; pfd++) {
        file = fdget(pfd).file;
        file->f_op->poll(...);
    }
    ...
}
```

Listing 1: The simplified code snippet of poll system call handler in the Linux kernel.

gadget to pivot into the controlled region and launch code reuse attacks. During the lifetime of the `poll` system call, the on-stack data is valid (*i.e.*, its presence on the kernel stack is *not* a security violation), but it is used for malicious purposes.

Preserved Registers. Each user space thread has its own kernel stack. When the user space thread invokes a system call, the kernel will switch to using the associated kernel stack by setting the `rsp` register. Immediately following the stack pointer change, the kernel pushes the user space context onto the kernel stack to preserve the context as shown in Figure 1. Here, the "user space context" is a data structure called `pt_regs` [65] that includes all of the user space registers. These values can be carefully set by malicious users before invoking the system call.

In other words, a fully user-controllable region is at the bottom of the kernel stack. When the attacker triggers a CFHP, they can use the controlled `pt_regs` region as a ROP payload.

Calling Convention. In the Linux kernel's calling convention [62], callee and caller functions both need to preserve sets of registers, termed "callee-saved" and "caller-saved" registers. The way compilers implement the calling convention is by emitting code to push the registers that will be potentially clobbered in the function to the kernel stack at the function prologue and pop them at the epilogue. This mechanism gives attackers two ways to place controlled data onto the stack.

First, registers can be pushed onto the kernel stack when the kernel invokes the actual system call handler function to serve the system call. System call handlers are invoked right after userspace registers are pushed onto the kernel stack as `pt_regs`. At the time of invoking system call handlers, the registers *still* contain userspace data. Since system call handlers are calling convention compliant, they will save registers that will be clobbered within the handler by pushing them onto the stack as well.

Second, the system call handlers might call other helper functions, which will also be calling convention compliant, and registers still containing user space data (or reloaded with user space data in the course of system call servicing) might be saved as a result.

Uninitialized Memory. Since there is only one kernel stack for each thread, data used in previous system calls within the same thread can be left on the kernel stack if not overwritten by the triggering system call. Due to the fact that the kernel's control flow is usually hijacked in the middle of a function, variables used *after* the hijacking point will not be initialized by design. In other words, hijacking control flow can create unexpected uninitialized memory situations on the kernel stack that would not be a security problem in normal operation and could result in a considerable number of usable data from the attackers' perspective.

```c
int __sys_recvfrom(...)
{
    struct sockaddr_storage address;
    ...
    // initialize `address`
    sock->ops->recvmsg(...);
    ...
    // copy address to user space
    copy_to_user(uaddr, &address, ulen);
}
```

Listing 2: The simplified code snippet of recvfrom system call handler in the Linux kernel.

An attacker can then carry out a targeted stack spray attack [45] to place controlled data on specified stack locations.

As shown in Listing 2, the `address` variable will be initialized inside the target socket's `recvmsg` handler, which means it will always be initialized and copied to user space in the normal control flow. But if the `sock` data structure is fully overwritten by an attacker through a vulnerability, when CFHP is obtained at `sock->ops->recvmsg`, the `address` variable is not initialized, thus susceptible to the targeted stack spray attack.

4.3 Weaponizing Spilled User Space Data

After obtaining enough data spillage on the kernel stack, the moment an attacker invokes the triggering system call and acquires CFHP, they have stack control as well. In other words, the attacker can immediately launch code reuse attacks. In this work, we focus on executing a ROP payload.

Starting ROP. In the optimal case, the stack pointer might be pointing directly to user-controlled data when the CFHP is triggered. However, even if it is not, the typical proximity of user-controlled data to the stack pointer means that the attacker can "reach" it by using the CFHP to redirect execution to a stack-shifting gadget such as `add rsp, X; ret`. After this redirection, the attacker's ROP chain, stored in user-controlled portions of the kernel stack, will be triggered.

Small Chains. However, the initial ROP payloads enabled by RetSpill are too small for full exploits. This is due to three factors.

First, different triggering system calls exhibit different data-spilling behaviors depending on how they process userspace data. While the amount of Preserved Registers is stable across different system calls, the amount of Valid Data, data spilled due to Calling Conventions, and Uninitialized Memory varies widely.

Second, the CFHP itself might require specific inputs in some registers and memory to trigger successfully. For example, triggering the CFHP for CVE-2010-2959 requires a specific value in the `fd` argument (passed via the `rdi` register) to the triggering `ioctl` system call. This means that spilled values from `rdi`, regardless of the spillage method, cannot be used in the ROP chain (as they cannot have arbitrary values). Other vulnerabilities require more arguments to have precise values, reducing the amount of controllable spilled data further.

Third, spilled user space data is not contiguous, and thus some of the resulting gadgets must be dedicated to shifting the stack to the next controllable spilled data.

With resulting ROP chains as short as 7 gadgets (as measured in §7.1), some cleverness is required to achieve full exploitation.

Independent ROP. Recall that the CFHP that initiates the ROP attack is provided by an overwritten victim object representing a system resource in kernel heap. Thus, all threads in the exploit process have access to the system resource, and also the victim object. An attacker can create a separate thread to execute the ROP chain so that ROP chain termination does not affect the main thread. This approach makes each ROP chain execution isolated in each thread, which means each ROP chain execution is performed under a different task context.

Notice that this step does not involve re-triggering vulnerabilities for additional ROP chain iterations. Hence, in practice, it does not introduce extra unreliability for exploits (for example, from having to otherwise repeat heap massaging efforts).

There are two advantages to this independent ROP chain execution. First, as described in §2, it enables the use of the `do_task_dead` function as an epilogue gadget in the ROP chain to end the chain gracefully. Second, since the ROP payload is executed in a new thread, even if the ROP payload fails, the kernel will only terminate the executing thread instead of the main thread. Therefore, as we will describe in §7.3, one could repetitively try the execution of ROP payloads to bypass certain probabilistic kernel protections without exploit reliability degradation.

Unlimited Arbitrary Read/Write/Exec. The victim object that gives the attacker CFHP is typically on the kernel heap and the attacker can put data on the kernel stack. Each time the attacker invokes the triggering system call, which grants the attacker CFHP, they can put different controlled data on the kernel stack and then invoke a different ROP chain. Combined with the Independent ROP Chain primitive, this feature of RetSpill effectively gives the attacker the ability to trigger different ROP chains as many times as desired without swapping payload in the kernel heap. Then the attacker can have unlimited Arbitrary Read/Write primitive, as well as the ability to invoke arbitrary functions in kernel space and get back the return values unlimited times.

In other words, RetSpill turns one single CFHP into unlimited Arbitrary Read/Write/Exec without sacrificing exploit reliability.

Putting it all together. At this point, the controlled victim object in the kernel heap becomes something analogous to a *backdoor*. By using the triggering system call on the victim object, the attacker can do Arbitrary Read/Write in kernel space and invoke Arbitrary Kernel Function Calls and obtain the return values. Moreover, they can perform all these actions in kernel space cleanly for *unlimited times* from separate task contexts. In other words, the boundary between the user space and the kernel space is completely broken.

5 RETSPILL VS MODERN DEFENSES

In this section, we state how RetSpill manages to bypass a series of modern defenses in the Linux kernel. These defenses cover all existing protections deployed in kernels shipped in major Linux-based operating systems (listed in Table 3). We also evaluate the effectiveness of emerging protections that are not implemented, not production-ready, or not deployed yet. A brief summary can be found in Table 2.

Defense	CFHP Achievable?	RetSpill Works?	Deployed?†
SMEP/SMAP/KPTI	✓	✓	✓
RANDKSTACK	✓	✓	✓
STACKLEAK	✓	✓	✗
FG-KASLR	✓	✓	✗
KCFI/IBT	✓	✓	✗
Shadow Stack*	✓	✓‡	✗
CFI+Shadow Stack*	✗	✗	✗

Table 2: RetSpill's applicability against all modern defenses. * Shadow Stack is unavailable in Linux on x86_64 architecture. † Whether the defense is deployed in at least one of the major Linux distros mentioned in Table 3. ‡ RetSpill works with restrictions.

OS	Version	Status
Ubuntu	22.04	✗
Debian	11	✗
CentOS	9	✓
Fedora	36	✗
Arch Linux	2022.10.01	✗
Mageia	8	✗
Mint	21	✗
Open SUSE	Leap 15.4	✗

Table 3: The `panic_on_oops` configuration in major Linux-based operating systems.

SMEP/SMAP/KPTI. These protections aim to prevent attackers from directly accessing user space data when obtaining CFHP in kernel space. But RetSpill does not rely on data in the user space. Instead, it solely relies on user-controlled data on the kernel stack. In other words, RetSpill shows that user-controlled data on the kernel stack alone is enough to compromise the security of the full system, thus bypassing these three protections.

RANDKSTACK. RANDKSTACK [53] introduces a randomized offset between function stack frames and the `pt_regs` region as shown in Figure 1. This protection aims to randomize the kernel stack layout at each system call entry to prevent deterministic data control on the kernel stack when CFHP is obtained. However, this protection only mitigates attacks that access data spillage from Preserved Registers and Uninitialized Memory. Attackers can still deterministically access controlled data spilled by Valid Data and Calling Convention because they are part of the stack frames.

Moreover, we discover that Preserved Registers data spillage can still be used for exploitation despite the presence of this protection. Due to the low entropy (5-bit [53]) in the randomized offset of the current design, attackers can hardcode an offset and prepend a "ret-sled" (a series of `ret` gadgets) to the actual ROP chain to increase the chance of executing the actual ROP payload correctly.

More importantly, if the target system disables `panic_on_oops`, which is true by default for most major Linux-based operating

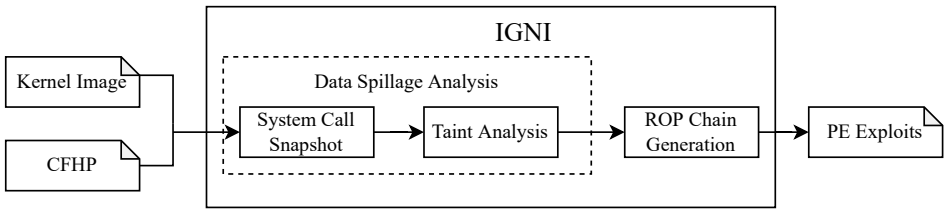

Figure 2: IGNI Design Overview

systems, as shown in Table 3, RANDKSTACK can be fully bypassed. In other words, RetSpill can bypass RANDKSTACK on most of the Linux distros. The reason is that with `panic_on_oops` disabled, an oops in task contexts will not crash the kernel. By using threads to trigger the payload, if an attempt fails, only the offending thread will be terminated. The exploit process will be kept intact and can continue attempts until the exploit succeeds. Such a create-thread-and-retry attack works only if failed attempts only generate oops and do not crash the kernel. We demonstrate that this is realistic—by calling a system call that is known to place "safe" data, such as NULL pointers, on the kernel stack multiple times to ensure that failed attempts can only trigger invalid-memory access and generate oops. To the best of our knowledge, we are the first to demonstrate that RANDKSTACK can be fully bypassed through recovery from oops.

STACKLEAK/STRUCTLEAK/INITSTACK. These protections prevent attackers from accessing uninitialized memory. More specifically, STACKLEAK [50] clears the kernel stack after each system call and INITSTACK [51]/STRUCTLEAK [43] ensure stack variables are initialized before use. In other words, they only prevent user data spillage from Uninitialized Memory. RetSpill is still effective by using other user data spillage methods to obtain stack control.

FG-KASLR. Function Granular Kernel Address Space Layout Randomization (FG-KASLR) [34] shuffles all kernel function addresses during boot time to protect the kernel from CFH attacks. However, function-level randomization is not enough to mitigate RetSpill. Other code snippets, such as assembly stubs, can also be compiled into executable sections with fixed offsets in the kernel. With RetSpill, attackers can look for ROP gadgets in the position-invariant code and use the arbitrary-read primitive to dynamically resolve function addresses, as demonstrated by Dang Khac Minh Le [38]. Then they can trigger the payload again and use the resolved function addresses to achieve the exploitation using RetSpill.

KCFI/IBT. KCFI [60] and Indirect Branch Tracking (IBT) [76] are two control-flow integrity (CFI) protection schemes merged into the Linux kernel recently. They both aim to protect the Linux kernel from forward-edge control-flow hijacking attacks. While they make it harder, they do not eliminate CFHP attacks.

Due to compatibility issues, even with the presence of these strong protections, some call targets in the Linux kernel are still not protected, susceptible to CFHP attacks. For example, in a kernel compiled with KCFI, `__efi_call` function does not verify its call targets. An attacker can overwrite the call target and obtain PC-control. We confirmed with the Linux kernel team that some call targets are not verified due to incompatibility between some subsystems and the CFI schemes.

Even with perfect implementation, attackers can still obtain PC-control by hijacking backward-edge control flow because KCFI/IBT does not protect them. For example, attackers have demonstrated the capability of turning a heap-based vulnerability to the kernel stack (dubbed kernel stack overflow primitive) [30], which can tamper the backward-edge control flow to obtain PC-control.

Once PC-control is obtained, attackers can carry out RetSpill despite the presence of KCFI and IBT because RetSpill itself does not rely on forward-edge control-flow hijacking.

Shadow Stack. Shadows Stack is proposed to protect backward-edge control in the Linux kernel. However, it is not implemented yet in the Linux kernel for the x86_64 architecture. We only analyze its security guarantee against RetSpill in theory. In kernels protected by Shadow Stack, attackers can obtain PC-Control from forward-edge control flow hijacking, but they cannot ROP. To carry out RetSpill, attackers can use other code reuse attacks that do not rely on backward-edge control flow hijacking, such as JOP[3]/PCOP[55] to utilize data spillage from user space,

CFI+Shadow Stack. In theory, on a kernel deployed with perfectly implemented CFI schemes and Shadow Stack, attackers cannot obtain CFHP anymore, which prevents RetSpill.

6 SEMI-AUTOMATIC RETSPILL

RetSpill has a critical impact on the Linux kernel because it can not only escalate privilege but also has the potential to achieve the escalation in a semi-automated pipeline. To show the feasibility of semi-automation, we represent IGNI, a framework that can semi-automatically generate RetSpill attacks for many real-world CFH cases. As the goal of this work is to show the feasibility rather than creating a tool to facilitate malicious attacks, we only take one attack payload for privilege escalation, namely executing `commit_creds(init_cred)` and returning back to user space gracefully. Creating attacking tools that can comprehensively leverage different attack vectors is out of the scope of this paper.

The design of IGNI is shown in Figure 2. IGNI takes a proof-of-concept (PoC) binary that demonstrates CFHP and a target kernel image as input, and outputs all the information needed for carrying out RetSpill. More specifically, it outputs 1) the address of a stack-shifting gadget that can transfer the initial CHFP into kernel stack and 2) an `ignite` function that performs data spillage, invokes the triggering system call, and leads to PE automatically. By setting the initial CFHP to the stack-shifting gadget and changing the original triggering system call to `ignite` function, the initial crashing PoC is turned into a full-fledged PE exploit.

Our approach relies on virtual machine (VM) snapshots. We take a VM snapshot when the triggering system call just enters kernel space, which ensures deterministic CFHP. Then, we analyze all user

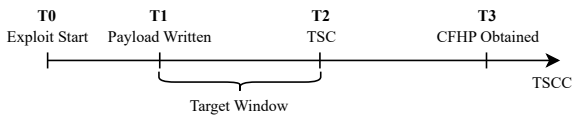

Figure 3: CFH exploits timeline

data that flows into the kernel space and identify user data that can be used for data spillage. Finally, we use symbolic execution to generate a ROP chain using the spilled data and link it with the initial CFHP to perform a full exploitation.

The design of IGNI involves the three following challenges:

- How to identify the triggering system call when thousands of irrelevant system calls will be invoked during the lifespan of an exploit process?
- Once we locate the triggering system call, how to distinguish user-controllable data from others?
- After finding the kernel data provenance, how to generate a PE payload on the discrete controlled region?

We detail how to conquer these challenges in this section.

6.1 Snapshotting the Triggering System Call

In Linux kernel exploits, CFHP is usually obtained through system calls. We call the system call that obtains CFHP the triggering system call (TSC). During the life span of the exploit process, thousands of irrelevant system calls will be invoked, it is challenging to identify the correct system call that will lead to CFHP and take a VM snapshot at its entrance. For example, in the exploit for CVE-2017-7533 [1], attackers use `write` system call to hijack the kernel control flow. The one `write` system call leading to CFHP is TSC, while another `write` system call displaying some debug messages is irrelevant.

We define triggering system call candidates (TSCC) as system calls invoked by the exploit process and having the same syscall number as TSC. The challenge becomes: how to identify the TSC among all TSCCs while the number and order of TSCCs may differ across runs (because of threading).

Intuitively, among all TSCCs, the last one invoked before obtaining CFHP is TSC. However, this assumption is wrong. In practice, the exploit process can issue other system calls, including other TSCCs, after issuing the TSC. The reasons are twofold. First, after the TSC is invoked, before it reaches the code that grants CFHP, other threads in the exploit process may race with the execution and invoke more TSCCs. Second, even when the triggering system call is executing, the kernel may stop its execution halfway and switch to execute other threads, which can issue more system calls, because of the voluntary preemption mechanism in the Linux kernel. Depending on how the exploit is implemented, the number of extra TSCCs after invoking the TSC can be drastically different.

One naive solution will be taking a VM snapshot at the entrance of each TSCC and verifying whether it is the TSC. This solution does not work because constantly taking VM snapshots alters the timing difference between kernel threads, thus failing time-sensitive exploits (*i.e.*, race condition exploits). Notice that binary search does not apply either because the number of TSCCs varies across runs.

We show the timeline of CFH exploits with respect to TSCC in Figure 3. Our heuristic-based search algorithm is built upon an insight that TSC can be deduced from a snapshot taken between T1-T2. This is because after the exploit payload is placed correctly in the heap, CFHP can be obtained deterministically [75]. To deduce the TSC, we run the snapshot to T3, obtain information about the TSC thread (`task_struct`), rewind the snapshot, and take a snapshot at TSC using the thread-specific information.

To take a snapshot between T1-T2, we use a search algorithm with increasing granularity. More specifically, at the start of the search, we take a snapshot of every `N` TSCCs until we obtain CFHP. Then we revert the execution to the last snapshot and start taking a snapshot of every `N/2` TSCCs. We repeat this process until we obtain CFHP by executing one single TSCC. This algorithm ensures at least one snapshot is taken between T1-T3. Since T2-T3 is a time frame less than the time it takes to finish one system call (TSC), it is likely to obtain snapshots between T1-T2. As demonstrated in Section 7.2, the algorithm succeeds for all 22 exploit samples.

6.2 Identifying User-controllable Data

In this step, we mutate all potential userspace inputs (*i.e.*, registers and user memory) that 1) flow into the kernel space and 2) are not relevant in obtaining CFHP. In other words, we want to identify all userspace inputs that can be used for data spillage.

Notice that the VM snapshot taken from the previous step ensures the deterministic execution of TSC: it will always lead to CFHP without external intervention. We mutate all user space inputs word by word (because they are used for placing gadgets) directly in a fresh VM state (by rewinding VM states using the snapshot) and continue the execution. If the execution still leads to CFHP, the word can be used for data spillage since it does not affect CFHP. The mutation is performed by directly modifying user space registers and hooking kernel functions that consume user space data, namely, `copy_from_user` and `get_user`.

Then we analyze the provenance of all data spillage for payload generation in the next step. Specifically, we taint the whole kernel stack and CFHP-irrelevant userspace input with different special values in the VM state and then continue its execution to CFHP. By analyzing the tainted data on the kernel stack when CFHP is obtained, we can learn the amount of data spillage we control and their provenance.

6.3 ROP Chain Generation

In this step, we generate a ROP chain based on the user-controlled data on the kernel stack. There are two challenges in this task. First, the user-controlled data scatter over the kernel stack because they are spilled on the stack through various methods (Section 4.2), as shown in Figure 4. Second, there are implicit constraints on the user-controlled data because several data spillages may come from the same source (*e.g.* `rbx` can lead to both Saved Registers and Calling Convention spillages), and they will be of the same value on the kernel stack.

ROP chain generation on discrete controlled regions is an open challenge: all existing automatic ROP chain generation solutions assume consecutive data control on the stack [2, 9, 26, 57, 58, 69].

CVE	Version	Trigger System Call	P.R.	C.C	V.D.	U.M.	Total	Total w/o U.M.	Original Technique Still Applicable	RetSpill Automated
2010-2959	4.15	ioctl	9	7	0	10	22	12	✗	✓
2016-0728	4.15	keyctl	4	3	0	0	7	7	✗	✓
2016-4557	4.15	close	5	3	0	0	8	8	-	✓
2016-6187	4.15	read	3	6	0	2	11	9	✓	✓
2017-2636	4.15	recvfrom	4	8	0	14	26	12	✗	✓
2017-6074	4.15	recvfrom	4	8	0	14	26	12	✗	✓
2017-7184	4.15	read	3	6	0	2	11	9	✓	✓
2017-7308	4.15	lseek	4	5	0	0	9	9	✗	✓
2017-7533	4.15	write	3	6	0	3	12	9	-	✓
2017-8824	4.15	getsockopt	2	5	0	2	9	7	✗	✓
2017-10661	4.15	clock_adjtime	4	7	24	1	36	35	✗	✓
2017-11176	4.15	setsockopt	3	7	0	0	10	10	✗	✓
2018-6555	4.15	getsockopt	2	6	0	1	9	8	✗	✓
2021-3490	5.11.0	prctl	12	4	0	3	19	16	-	✓
2021-3492	5.4.94	read	9	0	0	0	9	9	✓	✓
2021-4154	5.4.120	read	9	0	0	6	15	9	✓	✓
2021-27365	5.11.0	sendmsg	10	0	1	32	43	11	✗	✗*
2021-43267	5.11.0	ioctl	11	1	0	15	27	12	-	✓
2022-0185	5.4.120	read	9	0	0	6	15	9	✓	✓
2022-1786	5.10.90	execve	9	0	0	0	9	9	-	✓
2022-25636	5.11.0	read	9	0	0	0	9	9	✓	✗
2022-29581	5.4.170	ioctl	10	3	0	9	22	13	✓	✓
	AVERAGE		6.1	3.9	1.1	5.5	16.5	11.1		

Table 4: Data spillage is prevalent in system calls. The abundance of data spillage allows 20 out of 22 proof-of-concept programs that manifest CFHP to be semi-automatically turned into full privilege escalation exploits. Among the 20 success samples, five do not have public end-to-end CFH exploits. For ten samples, the techniques used in the original exploits no longer work with the presence of modern protections, but IGNI successfully make them work again automatically using RetSpill. Detailed information on the original techniques is shown in Table 7 in Appendix. Preserved Registers (P.R.), Calling Convention (C.C.), Valid Data (V.D.), and Uninitialized Memory (U.M.), the original exploit does not provide CFHP or there is no public end-to-end exploit (-). * RetSpill works on CVE-2021-27365 with manual modification.

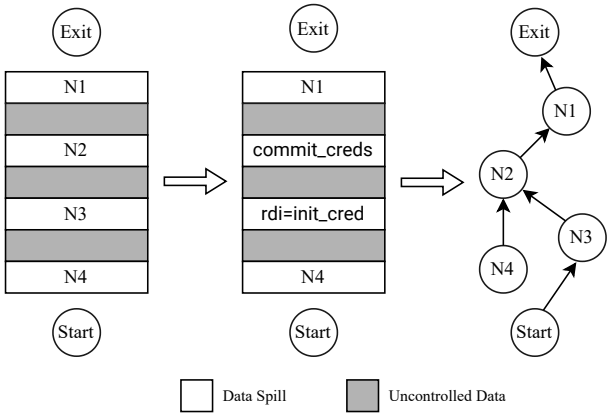

Figure 4: IGNI turns the discrete ROP chain generation problem into a graph search problem. Each node in the graph is a controlled data spill and each edge is a bridging gadget that can perform the control flow transfer.

The difficulty lies in how to connect multiple controlled areas while maintaining CFH.

To tackle this problem, we propose to use graph search to solve the controlled area connection problem and find potential ROP chain candidates, then use symbolic execution to verify ROP chains, similar to BOPC [28]. Notice that the longer the needed ROP chain payload is, the more implicit data constraints will be added to the graph. Thus, our approach may not scale to long ROP chains. In this work, we only focus on executing `commit_creds(init_cred)` and returning back to user space gracefully to demonstrate RetSpill.

We first fit independently-generated (using `angrop`[2]) ROP chain pieces into controlled memory and stitch them together using stack-shifting gadgets (*e.g.*, `add rsp, 0x10; ret`). For example, in Figure 4, we fit `| pop rdi; ret | init_cred |` into `N3` first. Then the stitching algorithm turns each ROP chain piece and other controlled gadgets into nodes in a Directed Acyclic Graph. Two nodes are connected if there is a stack-shifting gadget that transfers the stack control from one node to the other. The start node is a special node that does not represent any controlled memory. Instead, it represents the initial CFHP. An exit node is a Preserved Register data spillage. We explain its significance later. At this stage, each path in the graph represents an execution flow that starts from the initial CFHP provided by the given PoC program (start node), traverses all ROP chain pieces, and ends with a Preserved Register spillage node. So far, the algorithm does not guarantee a successful ROP execution because of implicit data constraints. For example, if a `pop rdi; add rsp, 0x10; ret` stack-shifting gadget is chosen after setting `rdi` to `init_cred`, the ROP chain will fail. But the algorithm reduces the amount of potential ROP chain candidates because it solves the controlled area connection problem. We then use symbolic execution (*i.e.*, `angr`) to verify the execution results of the chains to select a working chain that can escalate privilege and reach a controlled Preserved Register spillage.

Every Preserved Register data spillage can end ROP chains in kernel space gracefully. This is because preserved registers are stored in `pt_regs` and KPTI trampoline [38] is a code snippet that pops all the `pt_regs` and returns back to user space. By carefully calculating offsets in KPTI trampoline, we can use one Preserved Register gadget to resume the normal execution of the KPTI trampoline and return back to user space gracefully. For example, suppose the data in `pt_regs` is `r10|r9|r8` and the trampoline needs to restore the registers by `pop r10; pop r9; pop r8`. If we set `r10` (part of our ROP chain) to the address of `pop r9; pop r8;`, when `r10` is consumed by our ROP chain, the stack content will become `r9|r8` and the PC points to `pop r9; pop r8`, which resumes the normal execution flow of the KPTI trampoline and can return back to user space gracefully.

As a result, our algorithm ensures a payload that starts from the initial CFHP, runs the privilege escalation payload, and returns back to user space gracefully.

7 EVALUATION

In this section, we evaluate the RetSpill attack by answering three questions as follows:

- How feasible is the RetSpill attack when applied to real-world vulnerabilities?
- How effective is the semi-automatic RetSpill attack applied to real-world vulnerabilities?
- How effective is RetSpill in bypassing existing Linux kernel protections?

All the experiment artifacts are open-source.

7.1 Data Spillage Feasibility

We evaluate RetSpill's applicability by measuring the amount of data on the kernel stack that is fully controllable by attackers when a triggering system call is invoked. We only include fully controllable gadgets, because partial control on stack data (*e.g.*, 32-bit control on a 64-bit machine) does not necessarily enable attackers to execute gadgets, thus it does not reflect the applicability of RetSpill. The numbers are obtained by using the taint analysis described in §6.2.

Dataset. We create a dataset consisting of 22 proof-of-concept exploits that provide CFHP for evaluating the effectiveness of RetSpill. It includes 13 exploits from previous work [75] and nine publicly accessible exploits. In total, we exclude 4 exploits from the dataset of previous work. CFHPs in 2 of 4 excluded samples are not triggerable in system calls, applying RetSpill to those 2 exploits requires significant engineering effort, thus we exclude them. The other 2 excluded samples only work when the SMAP protection is disabled, which requires a weaker threat model than our paper's. As a result, we exclude them from our dataset.

In the dataset, we enable all the protections compatible with the kernels, namely, `SMEP`, `SMAP`, `KPTI`, `NX-physmap`, `CR pinning`, and `STACK CANARY`. We do not enable `STATIC_USERMODE_HELPER` because it is not compatible with kernels used in the dataset. FG-KASLR, RKP, and PT-Rand are not enabled because they are not in the Linux kernel main tree, thus not in our kernel source code.

As shown in Table 4, there is a considerable amount of user-controllable data on the kernel stack when CFHP is obtained in the triggering system calls. On average, when attackers obtain CFHP, there are 16.5 gadgets readily on kernel stack, which is more than enough for carrying out privilege escalation.

CVE	P.R.	C.C.	V.D.	U.M.	Total
2010-2959	3	6	29	75	113
2016-4557	3	6	29	75	113
2017-7308	3	6	29	75	113

Table 5: The number of fully controllable gadgets by trigging payloads using the `poll` system call. Preserved Registers (P.R.), Calling Convention (C.C.), Valid Data (V.D.), and Uninitialized Memory (U.M.).

Notice that in the same exploit with the same payload, using different triggering system calls to obtain CFHP will lead to a different exploitation scenario because of the difference in stack layouts. Thus, by carefully choosing triggering system calls, the amount of fully controlled data on the kernel stack can be greatly increased. We demonstrate this by using `poll` system call to trigger payloads for a subset of the exploits, the result is shown in Table 5.

7.2 Semi-Automation Effectiveness

We evaluate RetSpill's ability to semi-automatically turn proof-of-concept programs into privilege escalation exploits. We contacted the authors of targeted stack spray [45] and leak-kptr [8], but failed to port their prototypes to work against the modern Linux kernel. Thus, without a method to automatically perform targeted stack spray, we only use Valid Data, Preserved Registers, and Calling Conventions for data spillage in this evaluation. We tested the refined IGNI on the same set of real-world vulnerabilities built for the data spillage feasibility experiment (§7.1). As shown in Table 4, our prototype can generate end-to-end exploits for 20 out of the 22 samples even without using Uninitialized Memory for data spillage. This result demonstrates that RetSpill can be automated, and is thus that much more of a dangerous exploitation technique.

We further investigated and determined that CVE-2021-27364 *is* susceptible to RetSpill attack. Our prototype fails to automatically generate an exploit for it because of the lack of methods to perform targeted stack spray automatically. We successfully create an end-to-end exploit for CVE-2021-27364 using RetSpill by performing the targeted stack spray, which takes advantage of spilled data from uninitialized memory, manually. Our system fails on CVE-2022-25636 because of the lack of a specific bridging gadget to transfer control flow into a controlled region. In other words, RetSpill fails on this vulnerability for this specific kernel build. In our experiment, this kernel is compiled with the default configuration plus the aforementioned protections, and thus does not have much reusable code. However, in real-world kernel builds compiled with many additional functionalities enabled, we may be able to find the missing bridging gadget in the larger reusable code region and make RetSpill possible.

7.3 Protection Bypass

Due to the fact that RetSpill provides a probabilistic success rate in bypassing the most strict randomized-based protections, to

```
long vuln_ioctl(...)
{
    switch(cmd) {
    case CMD_VULN:
      obj = kzalloc(sizeof(vuln_obj_t), GFP_KERNEL);
      copy_from_user(obj, arg, sizeof(obj));
    case CMD_TRIGGER:
      obj->func();
    }
}
```

Listing 3: The simplified code snippet of the contrived vulnerable kernel module.

accurately measure its success rate in bypassing protections, we use a synthetic vulnerability to guarantee CFHP. More specifically, we write a vulnerable kernel module that contains a contrived vulnerability that provides CFHP to simulate real-world exploit scenarios. We do not use real-world exploits for this evaluation because the unreliability of real-world kernel exploits in providing CFHP [75] will affect our measurement of RetSpill's effectiveness. The gist of the vulnerable kernel module is shown in Listing 3.

RANDKSTACK. We compile a Linux kernel on `x86_64` architecture with the default configuration and enable RANDSTACK. Since RANDKSTACK adds a random offset on kernel stack, which can only mitigate Preserved Registers and Uninitialized Memory data spillage methods, in our RetSpill exploit, we only use Preserved Registers to put user data on the kernel stack to evaluate this mitigation objectively. Our RetSpill exploit aims to obtain privilege escalation using the `commit_creds(init_cred)` payload.

We perform 5000 trials each on the kernel with `panic_on_oops` on and off. In each trial, we insert the vulnerable kernel module and then repetitively run the exploit until it succeeds or crashes the kernel, following the previous work's method [75]. The result shows that RetSpill can achieve 100% success rate on systems with `panic_on_oops` off. Even with `panic_on_oops` on, the RetSpill exploit can still achieve 25.44% success rate, which is non-negligible. Since `panic_on_oops` is off by default on most Linux distributions as shown in Table 3, this suggests that even with RANDKSTACK on, RetSpill can achieve 100% success rate on most Linux distributions.

Notice that the 25.44% is achieved when the attacker only relies on Preserved Registers or Uninitialized Memory for data spillage. If they use Valid Data or Calling Convention for data spillage, this mitigation cannot provide any protection. In other words, if attackers can use Valid Data and Calling Convention to perform data spillage (which is common according to Table 4), they can still achieve 100% success rate despite the presence of RANDKSTACK.

KCFI/IBT. We simulate a vulnerability that grants PC-control in kernels compiled with CFI schemes by modifying the vulnerable kernel module used in the previous setup. More specifically, the function pointer is directly invoked using `__x86_indirect_thunk_rdi` that transforms forward-edge control-flow transition to backward-edge control-flow transition, which is not protected by the CFI schemes. This setting is realistic, as discussed in §5.

We successfully performed CFH attacks and obtained privilege escalation on kernels compiled with KCFI and IBT using RetSpill. The success is because KCFI and IBT only protect forward-edge control flow. Once PC-control is obtained, RetSpill does not rely on the ability to hijack forward-edge control flow again because one initial CFHP is enough for using data spilled on the kernel stack to carry out malicious payload through ROP. We demonstrate RetSpill's ability to bypass KCFI/IBT with experiments [3].

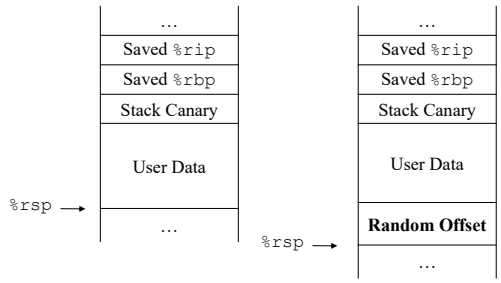

Figure 5: Unhardened and Hardened Stack Frames.

FG-KASLR. We contacted the authors of FG-KASLR and obtained its latest source code. Similar to the setup in RANDKSTACK, we compile the kernel with default configuration with FG-KASLR enabled. We write a tool to parse the kernel's FG-KASLR-related metadata and extract a position-invariant region of 4MB from the kernel image we build. As mentioned in §5, this region consists of executable code that is not within the boundary of any functions (*e.g.*, manually written assembly stubs). In total, we obtained 42631 unique gadgets from the position-invariant region, which is more than enough for dynamically resolving kernel function address. We succeed in using RetSpill to perform privileged escalation against the kernel compiled with FG-KASLR using the vulnerability in the contrived kernel module.

Case study: Breaking Security Boundary. We demonstrate the ability of RetSpill to break the security boundary between user space and kernel space using a real-world example. We implement an end-to-end proof-of-concept exploit for CVE-2022-1786 that performs Arbitrary Read/Write in kernel memory and Arbitrary Function Call in one single exploit by only changing the payload on the stack. Besides the individual primitives, this experiment also confirms RetSpill's ability to provide unlimited Arbitrary Read/Write/Exec primitive. In our analysis, Independent ROP Chain proves to be vital in obtaining CFHP multiple times. This is because, in this specific exploit, CFHP is obtained after the exploit thread acquires a lock. Since the control flow returns back to user space without releasing the lock, if we try to obtain CFHP again in the same thread, the exploit will stop indefinitely waiting for the lock to be released. Independent ROP Chain provided by RetSpill avoids the locking issue because each CFHP is obtained in a different thread.

8 DEFENSE

In this section, we explore plausible protections against RetSpill besides CFI+Shadow Stack, which is not ready yet and requires special hardware. To mitigate this exploitation technique, the key is to eliminate *all* data spillage on the kernel stack.

Preserved Register. RANDKSTACK is the only existing protection that mitigates Preserved Register data spillage. It comes with near-zero performance overhead [17] but offers probabilistic protection. In our experiment, exploits can still have a 25.44% chance of succeeding against kernels with this protection. Moreover, this

[3] https://github.com/sefcom/RetSpill/tree/main/experiments/kcfi_eval

protection should be coupled with `panic_on_oops`, or it can be fully bypassed (§7.3). To completely mitigate this data spillage, we suggest preserving user space registers elsewhere instead of on the kernel stack. For example, the registers can be saved in the associated task's `task_struct` data structure. This way, even if CFHP is obtained by attackers, user space registers will not be directly accessible on kernel stacks to facilitate further exploitation.

Uninitialized Memory. For Uninitialized Memory data spillage, all STACKLEAK, STRUCTLEAK, INITSTACK, and RANDKSTACK can provide different levels of protection. STACKLEAK clears the kernel stack after each system call, which ensures zero usable Uninitialized Memory data spillage when CFHP is obtained. However, STACKLEAK has an average overhead of more than 40% [44]. INITSTACK and STRUCTLEAK use compilers' pattern initialization feature to initialize stack variables, which also ensures no Uninitialized Memory data spillage while having only a performance overhead of 2.7%-4.5% [72]. As mentioned above, RANDKSTACK has near-zero performance overhead but only provides probabilistic protection. Considering protection and performance, we suggest enabling INITSTACK or STRUCTLEAK to avoid Uninitialized Memory data spillage on kernel stack.

Calling Convention and Valid Data. The latest version of the Linux kernel has deployed a protection that clears user space registers that are not part of the ABI in the system call entry stub. As shown in Table 4, new kernel builds have significantly less data spillage caused by Calling Convention compared with old kernel builds because of this protection. However, data spillage caused by Calling Convention is still not eliminated.

Data spillage caused by Valid Data and Calling Convention is part of the design, and thus cannot be mitigated by existing protections in the Linux kernel. However, they are enough for successfully launching RetSpill to achieve full system compromise. We demonstrate it by successfully exploiting CVE-2016-4557 with RetSpill only using Valid Data and Calling Convention spillage. To fill the gap, we developed a protection to mitigate this data spillage. As shown in Figure 5, our protection inserts a random offset at the bottom of each stack frame. To prevent the random offset region from becoming the source of Uninitialized Memory data spillage, we clear the region right after the insertion. As a consequence, when CFHP is obtained by attackers, they will not be able to reuse user data directly without guessing the random offset, which drastically reduces exploit reliability.

Denote the average call-stack depth as D, the entropy of the proposed protection is $5 + log(D)$ bits, which is slightly higher than RANDKSTACK(5 bits). More importantly, the protection prevents attackers from accessing *all* data spillage, whereas RANDKSTACK only prevents Preserved Registers and Uninitialized Memory.

Due to its lightweight nature, the proposed protection only introduces an average performance overhead of 0.61% as evaluated on benchmarks from LMBench and Phoronix. The detailed evaluation result can be found in Table 6.

Benchmark	Vanilla	Hardened	Overhead
Phoronix			
Apache (Reqs/s)	149470.35	150295.67	0.55%
PHPBench (Score)	830189	840718	1.27%
PyBench (ms)	889	885	0.45%
OpenSSL-SHA256 (byte/s)	20984220257	20797618197	-0.89%
OpenSSL-SHA512 (byte/s)	6859393693	6812764710	-0.68%
OpenSSL-RSA4096 (sign/s)	3119.7	3105.9	-0.44%
OpenSSL-RSA4096 (verify/s)	204263.3	203177.1	-0.53%
LMBench			
Context Switch (ms)	2.036428571	2.047	-0.52%
UDP (ms)	7.3116	7.4854	-2.32%
TCP (ms)	9.2525	9.3546	-1.09%
10k File Create (ms)	10.99	10.89	0.92%
10k File Delete (ms)	5.89229	5.71896	3.03%
pipe (MB/s)	3648.6	3587.8	-1.67%
AF Unix (MB/s)	8968.9	8712.5	-2.86%
TCP (MB/s)	8081.6	7964.8	-1.45%
mmap Reread (MB/s)	18.61	17.95	-3.55%

Table 6: Performance overhead evaluation of the proposed defense on Phoronix and LMBench.

9 DISCUSSION

Automatic Exploit Context Analysis. In this work, we only analyze RetSpill under limited exploit contexts. As mentioned in §7.1, different system calls can be used to trigger the payload on the heap, which create different exploit contexts. To fully evaluate the severity of RetSpill, an automatic solution is needed to explore all possible exploit contexts for each vulnerability. One such work is FUZE [71], which combines fuzzing and symbolic execution to facilitate exploit generation automatically. We can potentially use its fuzzing and symbolic execution engines to explore different exploit contexts automatically. We leave the integration with FUZE for automatic exploit context analysis as future work.

Uninitialized Memory Spillage Evaluation. In our current evaluation for IGNI, we do not include data spillage caused by uninitialized memory. This is because we lack a way to automatically and deterministically control uninitialized memory on the kernel stack. The most relevant research lately on this topic are targeted stack spray [45] and leak-kptr [8]. We contacted the authors but failed to port their prototypes to the modern Linux kernel because of its rapid development in the past few years. Since the evaluation requires a new automatic approach to perform stack spray, which is out of the scope of this work, we leave it as future work.

Fine-Grained Randomization Protections. Existing such protections in the Linux kernel (*e.g.*, FG-KASLR [34] and kR^X [48]) cannot randomize all kernel executable code, which leaves the kernel vulnerable to RetSpill. Even with perfect fine-grained randomization protection, attackers can still use speculative probing [21] or an arbitrary read primitive to disclose usable ROP gadgets and launch RetSpill in JIT-ROP [59] style. Thus, we believe fine-grained randomization may not be a good solution to mitigate RetSpill.

Stealthy Rootkit. Primitives provided by RetSpill allow attackers to reliably program the kernel from user space. This opens up the possibility of rootkits that manipulate the kernel without escalating its privilege. As a consequence, this hypothetical rootkit can bypass intrusion detection techniques based on detecting suspicious privilege escalation behaviors [27, 56]. Worse still, its every action involves only the triggering system call, which means malicious

logic can be executed in kernel space while only leaving a trace of seemingly benign system calls such as `close`.

Applicability to Other Systems. RetSpill is not specific to Linux. Any system that involves exchanging data between a trusted entity and an untrusted one needs to avoid untrusted data spillage on the trusted stacks. For example, other operating systems or hypervisors are potentially susceptible to RetSpill attack. We leave the verification and severity evaluation of RetSpill on other systems as future work.

10 RELATED WORK

Exploring Memory Corruption Capability. Memory corruption capability is obtained after a vulnerability is triggered. In order to reliably trigger race condition bugs, Lee *et al.* [39] use interrupts to extend race windows for winning races and eventually achieve memory corruption capability. FUZE [71] explores different use sites of UAF vulnerabilities with under-context fuzzing. GREBE [41] proposes an object-driven fuzzing technique to trigger vulnerabilities in different contexts and explore different memory corruption capabilities. Syzscope [77] symbolizes kernel memory and extracts memory corruption capabilities through symbolic execution.

Obtaining Exploitation Primitive. With the memory corruption capability in hand, researchers propose various techniques to obtain exploitation primitives. Cho *et al.* [8] utilizes eBPF to achieve pointer leak from uninitialized stack variables. Lu *et al.* [45] introduce targeted stack spray techniques to exploit use-before-initialization vulnerabilities. There are also many research works on kernel heap exploitation. SLAKE [7] introduces different techniques to manipulate heap memory layout. With the help of elastic objects, ELOISE [6] proposes a general method to obtain information leak primitive from an overwrite capability. KOOBE [5] explores different exploit primitives from heap out-of-bound write capability. KHeaps [75] systematically studies the reliability problem in kernel exploitation and proposes a new stabilization technique. In order to prevent heap exploitation, many defenses are proposed. AUTOSLAB [40] proposes to isolate different types of objects into different slab caches and thus mitigates memory layout manipulation. Hardened Usercopy [11] is introduced to the Linux kernel to prevent out-of-bound access exploitation.

Escalating Privilege. Privilege escalation could be done through control flow hijacking (CFH) attacks. Originally, attackers could directly hijack the control flow to userspace [32]. The introduction of kernel space and user space isolation (e.g. KPTI [25, 64], SMEP [12]/SMAP [47]) mitigates such attacks. With the presence of isolation, attackers have to inject payload into kernel space. ret2dir [31] breaks this isolation with `physmap`, which is mapped in a predictable region in kernel space. To mitigate CFH attacks, KASLR [10] and KCFI [16, 20, 36, 73] are proposed. Specifically, KASLR randomizes kernel address space layout, making addresses of existing code unpredictable and thus downgrading the reliability of code reuse attacks. Recently, KASLR has been further extended as FG-KASLR [34], which randomizes address space layout at function granularity. CFI preserves the control-flow integrity to prevent CFH attacks. There are different implementations of CFI in the Linux kernel, including software-based [20, 22] and hardware-assisted [16, 36, 73]. Other than CFH attacks, data-only attacks are also capable of escalating privilege. Through an overwriting primitive, attackers could tamper the credentials of the exploit process to escalate privilege [4]. DirtyCred [42] demonstrates privileged escalation by replacing unprivileged credentials with privileged credentials. To prevent data-only attacks, xMP [52] protects sensitive data utilizing hardware virtualization techniques. PrivGuard [68] creates duplicate memory to ensure the modifications to sensitive data are legitimate.

Our work systematically studies the data spillage problem on kernel stack and discovers a novel and powerful attack: RetSpill. We thoroughly evaluate RetSpill and show that it can be automated with our prototype, IGNI.

RetSpill could bypass FG-KASLR [34] which previous research – KEPLER [70] is not capable of. In addition, compared to KEPLER and conventional ROP attacks [26, 69], RetSpill has no requirement of register control and heap control, which significantly simplifies the process of launching attacks. We also do not assume the information of the base of `physmap` as KEPLER. Furthermore, RetSpill enables unlimited CFHP without changing the payload on the heap, which introduces no degradation to exploit reliability for further exploitation. This feature of RetSpill makes it a stable exploitation technique, which other "single-shot" exploitation techniques, such as KEPLER, are unable to achieve.

11 CONCLUSION

We show that untrusted user data on kernel stack are threats to Linux kernel security. In this research, we discover RetSpill, a dangerous Linux kernel exploitation technique that requires minimal exploit primitives. Provided with CFHP, RetSpill can utilize seemingly harmless user space data on kernel stack to break the boundary between user space and kernel space despite the presence of all commonly deployed protections in Linux. We thoroughly study the technique and show that it is applicable in 21 out of 22 public exploits. Worse still, we demonstrate RetSpill can be automated, which makes it even more dangerous. Our proof-of-concept prototype, IGNI, semi-automatically turns 20 out of 22 public exploits that manifest CFHP into full privilege escalation exploits. Finally, we explore potential protections to minimize the impact of RetSpill.

12 ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their insightful feedback.

This material was supported by 2023 Google PhD Fellowship Program, NSF CNS-2000792, and grants from Defense Advanced Research Projects Agency (DARPA) under contracts HR001118C0060, FA8750-19C-0003, and N66001-22-C-4026.

REFERENCES

[1] sefcom/kheaps. https://github.com/sefcom/KHeaps/blob/master/exploit_env/CVEs/CVE-2017-7533/poc/poc_cfh_combo.c.

[2] angr team. angr/angrop. https://github.com/angr/angrop.

[3] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.

[4] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *11th USENIX Security Symposium (USENIX Security 02)*, 2002.

[5] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. {KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1093–1110, 2020.

[6] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1165–1184, 2020.

[7] Yueqi Chen and Xinyu 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*, pages 1707–1722, 2019.

[8] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[9] Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. Riscyrop: Automated return-oriented programming attacks on risc-v and arm64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 30–42, 2022.

[10] Kees Cook. Kernel address space layout randomization. https://lwn.net/Articles/546035/, 2013.

[11] Kees Cook. Hardened usercopy. https://lwn.net/Articles/693745/, 2016.

[12] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475/, 2012.

[13] Dino Dai Zovi. Practical return-oriented programming. *Source boston*, 2010.

[14] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *NDSS*, 2017.

[15] Vincent Dehors. Exploitation of a double free vulnerability in ubuntu shiftfs driver. https://www.synacktiv.com/en/publications/exploitation-of-a-double-free-vulnerability-in-ubuntu-shiftfs-driver-cve-2021-3492.html, 2021.

[16] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. Camouflage: Hardware-assisted cfi for the arm linux kernel. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[17] Marco Elver. stack: Introduce config_randomize_kstack_offset. https://lore.kernel.org/lkml/YfQ54x8zglPT%2FYnL@dev-arch.archlinux-ax161/t/#u, 2022.

[18] Nicolas FABRETTI. Cve-2017-11176: A step-by-step linux kernel exploitation. https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html.

[19] FizzBuzz101. Will's root: Cve-2022-0185. https://www.willsroot.io/2022/01/cve-2022-0185.html, 2022.

[20] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194. IEEE, 2016.

[21] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1871–1885, 2020.

[22] Google. Kernel control flow integrity. https://source.android.com/docs/security/test/kcfi.

[23] Google. Kernel exploit recipes notebook - google docs. https://docs.google.com/document/d/1a9uUAISBzw3ur1aLQqKc5JOQLaJYiOP5pe_B4xCT1KA/edit#.

[24] Grimm. Notquite0dayfriday/2021.03.12-linux-iscsi at trunk · grimm-co/notquite0dayfriday. https://github.com/grimm-co/NotQuite0DayFriday/tree/trunk/2021.03.12-linux-iscsi.

[25] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[26] Garrett Gu and Hovav Shacham. Return-oriented programming in risc-v. *arXiv preprint arXiv:2007.14995*, 2020.

[27] Isovalent. Detecting a container escape with cilium and ebpf - isovalent. https://isovalent.com/blog/post/2021-11-container-escape/, 2021.

[28] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882, 2018.

[29] David Bouman Jayden Rivers. Cve-2022-29582: An io_uring vulnerability. https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/, 2022.

[30] Xingyu Jin, Christian Resell, Clement Lecigne, and Neal Richard. Monitoring surveillance vendors: A deep dive into in-the-wild android full chains in 2021. https://i.blackhat.com/USA-22/Wednesday/US-22-Jin-Monitoring-Surveillance-Vendors.pdf, 2022.

[31] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 957–972, 2014.

[32] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. {kGuard}: Lightweight kernel protection against {Return-to-User} attacks. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 459–474, 2012.

[33] Andrey Konovalov. Project zero: Exploiting the linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[34] Alexander Lobakin Kristen Carlson Accardi. Function granular kaslr [lwn.net]. https://lwn.net/Articles/832434/, 2020.

[35] Greg Kroah-Hartman. Introduce static_usermodehelper to mediate call_usermodehelper() - patchwork. https://lore.kernel.org/all/20170116165044.GC29693@kroah.com/, 2017.

[36] Donghyun Kwon, Jiwon Seo, Sehyun Baek, Giyeol Kim, Sunwoo Ahn, and Yunheung Paek. Vm-cfi: Control-flow integrity for virtual machine kernel using intel pt. In *International Conference on Computational Science and Its Applications*, pages 127–137. Springer, 2018.

[37] Dang Le. Learning linux kernel exploitation - part 1. https://lkmidas.github.io/posts/20210123-linux-kernel-pwn-part-1/#the-simplest-exploit---ret2usr.

[38] Dang Le. Learning linux kernel exploitation - part 3 - midas blog. https://lkmidas.github.io/posts/20210205-linux-kernel-pwn-part-3/, 2021.

[39] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. {ExpRace}: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380, 2021.

[40] Zhenpeng Lin. How autoslab changes the memory unsafety game. https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game, 2021.

[41] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *2022 IEEE Symposium on Security and Privacy (S&P)*, pages 2078–2095. IEEE, 2022.

[42] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[43] Linxz. Pax - structleak. https://linxz.tech/post/compilers/2021-10-10-structleak/, 2021.

[44] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 920–932, 2016.

[45] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *NDSS*, 2017.

[46] Andy Nguyen. Cve-2021-22555: Turning \x00\x00 into 10000$ | security-research. https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html, 2021.

[47] OSDev.org. Supervisor memory protection - osdev wiki. https://wiki.osdev.org/Supervisor_Memory_Protection.

[48] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. krˆ x: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 420–436, 2017.

[49] Alexander 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, 2017.

[50] Alexander Popov. Stackleak: A long way to the linux kernel mainline. https://events19.linuxfoundation.org/wp-content/uploads/2017/11/STACKLEAK-A-Long-Way-to-the-Linux-Kernel-Mainline-Alexander-Popov-Positive-Technologies.pdf, 2018.

[51] Alexander Potapenko. security: allow using clang's zero initialization for stack variables. https://lwn.net/Articles/823152/, 2020.

[52] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xmp: Selective memory protection for kernel and user space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577. IEEE, 2020.

[53] Elena Reshetova. randomize kernel stack offset upon syscall. https://lwn.net/Articles/785484/, 2019.

[54] Matteo Rizzo. Kernote writeup. https://org.anize.rs/0CTF-2021-finals/pwn/kernote, 2021.

[55] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (pcop): chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, 14:139–156, 2018.

[56] Samsung. Real-time kernel protection (rkp). https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp, 2016.

[57] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
[58] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari, and Stephanie M Schwartz. A generic technique for automatically finding defense-aware code reuse attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1801, 2020.
[59] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE symposium on security and privacy*, pages 574–588. IEEE, 2013.
[60] Sami Tolvanen. Kcfi support [lwn.net]. https://lwn.net/Articles/893164/, 2022.
[61] Linus Torvalds. Kernel stack documentation. https://docs.kernel.org/x86/kernel-stacks.html.
[62] Linus Torvalds. Linux kernel calling convention source code. https://elixir.bootlin.com/linux/v5.17/source/arch/x86/entry/calling.h#L18.
[63] Linus Torvalds. native_write_cr4. https://elixir.bootlin.com/linux/v6.0/source/arch/x86/kernel/cpu/common.c#L447.
[64] Linus Torvalds. Page table isolation (pti) documentation. https://www.kernel.org/doc/html/latest/x86/pti.html.
[65] Linux Torvalds. Linux kernel source: arch/x86/include/asm/ptrace.h. https://elixir.bootlin.com/linux/v5.17/source/arch/x86/include/asm/ptrace.h#L59.
[66] Linux Torvalds. Linux kernel source code: fs/select.c. https://elixir.bootlin.com/linux/v5.17/source/fs/select.c#L982.
[67] Linux Torvalds. Linux kernel source: fs/read_write.c. https://elixir.bootlin.com/linux/v5.17/source/fs/read_write.c#L899.
[68] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. {PrivGuard}: Privacy regulation compliance made easier. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3753–3770, 2022.
[69] Yuan Wei, Senlin Luo, Jianwei Zhuge, Jing Gao, Ennan Zheng, Bo Li, and Limin Pan. Arg: Automatic rop chains generation. *IEEE Access*, 7:120152–120163, 2019.
[70] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, 2019.
[71] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.
[72] Xi Yang, Stephen M Blackburn, Daniel Frampton, Jennifer B Sartor, and Kathryn S McKinley. Why nothing matters: The impact of zeroing. *Acm Sigplan Notices*, 46(10):307–324, 2011.
[73] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. {In-Kernel}{Control-Flow} integrity on commodity {OSes} using {ARM} pointer authentication. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 89–106, 2022.
[74] Kyle Zeng. [cve-2022-1786] a journey to the dawn. https://blog.kylebot.net/2022/10/16/CVE-2022-1786/.
[75] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for {K (H) eaps}: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, 2022.
[76] Peter Zijlstra. [x86: Kernel ibt beginnings. https://lwn.net/ml/linux-kernel/20211122170301.764232470@infradead.org/, 2021.
[77] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3201–3217, 2022.

13 APPENDIX

CVE	Trigger System Call	Original Technique	A.P.	S.A.
2010-2959	ioctl	ret2usr	-	✗
2016-0728	keyctl	ret2usr	-	✗
2016-4557	close	N/A	-	-
2016-6187	read	KEPLER	②③	✓
2017-2636	recvfrom	change CR4	①	✗
2017-6074	recvfrom	change CR4	①	✗
2017-7184	read	pivot to heap	②④	✓
2017-7308	lseek	change CR4	①	✗
2017-7533	write	N/A	-	-
2017-8824	getsockopt	change CR4	①	✗
2017-10661	clock_adjtime	change CR4	①	✗
2017-11176	setsockopt	pivot to user	-	✗
2018-6555	getsockopt	ret2usr	-	✗
2021-3490	prctl	N/A	-	-
2021-3492	read	set_memory_x	①④	✓
2021-4154	read	pivot to heap	②④	✓
2021-27365	sendmsg	run_cmd	①④	✗
2021-43267	ioctl	N/A	-	-
2022-0185	read	pivot to heap	②④	✓
2022-1786	execve	N/A	-	-
2022-25636	read	pivot to heap	②④	✓
2022-29581	ioctl	pivot to heap	②④	✓

Table 7: All of the old techniques that do not require additional primitives are no longer applicable in modern systems. And all of the working techniques require additional primitives besides CFHP. Additional Primitives (A.P.) needed by the original technique compared with RetSpill, original technique Still Applicable (S.A.) against modern Linux kernel deployed with protections. ① rdi control, ② any register control, ③ physmap leak, ④ heap info leak, N/A: original exploit does not provide CFHP or there is no public end-to-end exploit.