ExpRace: Exploiting Kernel Races through Raising Interrupts

Yoochan Lee
Seoul National University
yoochan10@snu.ac.kr

Changwoo Min
Virginia Tech
changwoo@vt.edu

Byoungyoung Lee [*]
Seoul National University
byoungyoung@snu.ac.kr

Abstract

A kernel data race is notoriously challenging to detect, reproduce, and diagnose, mainly caused by nondeterministic thread interleaving. The kernel data race has a critical security implication since it often leads to memory corruption, which can be abused to launch privilege escalation attacks. Interestingly, due to the challenges above, the exploitation of the kernel data race is also challenging. Specifically, we find that some kernel races are nearly impossible to exploit due to their unique requirement on execution orders, which are almost impossible to happen without manual intervention.

This paper develops a generic exploitation technique for kernel data races. To this end, we first analyze kernel data races, which finds an intrinsic condition classifying easy-to-exploit and hard-to-exploit races. Then we develop ExpRace, a generic race exploitation technique for modern kernels, including Linux, Microsoft Windows, and MAC OS X. ExpRace turns hard-to-exploit races into easy-to-exploit races by manipulating an interrupt mechanism during the exploitation. According to our evaluation with 10 real-world hard-to-exploit races, ExpRace was able to exploit all of those within 10 to 118 seconds, while an exploitation without ExpRace failed for all given 24 hours.

1 Introduction

Data races are concurrency bugs, which occur when multiple threads access the same memory location while at least one access modifies the location. Without employing a proper synchronization mechanism (such as spinlocks, mutexes, etc.), the data race ends up with inconsistent results, severely harming the security and reliability of underlying systems.

Data races are notoriously difficult to detect, reproduce, and diagnose because they are inherently non-deterministic, caused by thread interleaving or scheduling. This unique characteristic and challenge of data races motivate many studies [3, 7, 9, 18, 27, 30, 40, 50, 57, 65, 66] to focus on assisting software developers in detecting, reproducing, and diagnosing race issues.

[*]Corresponding author

CVE	Kernel Ver.	Race Type	Crash Type	PoC
CVE-2016-8655	< Linux 4.8.12	Single-var.	Use-after-free	✔
CVE-2017-2636	< Linux 4.10.1	Single-var.	Double-free	✔
CVE-2017-7533	< Linux 4.12.3	Single-var.	Heap overflow	✔
CVE-2017-17712	< Linux 4.14.6	Single-var.	Uninitialized use	✗
CVE-2019-11486	< Linux 5.0.8	Single-var.	Use-after-free	✔
CVE-2017-15265	< Linux 4.13.8	Multi-var.	Use-after-free	✗
CVE-2019-1999	< Linux 4.19.37	Multi-var.	Use-after-free	▲
CVE-2019-2025	< Linux 4.19.6	Multi-var.	Use-after-free	▲
CVE-2019-6974	< Linux 4.20.8	Multi-var.	Use-after-free	✗
11eb85ec...	< Linux 5.6	Multi-var.	Use-after-free	✗
1a6084f8...	< Linux 5.6	Multi-var.	Use-after-free	✗
20f2e4c2...	< Linux4.19.97	Multi-var.	Use-after-free	✗
4842e98f...	< Linux 4.4	Multi-var.	Use-after-free	✗
da1b9564...	< Linux 4.14	Multi-var.	NULL deref.	✗
e20a2e9c...	< Linux 4.19.32	Multi-var.	Double-Free	✗

Table 1: Real-world kernel data races in Linux. ✔ denotes an exploit is publicly available; ▲ denotes an exploit is publicly available but requires kernel modification; ✗ denotes no publicly available exploits.

Notably, data races in the kernel can be abused to launch privilege escalation attacks. Data races often lead to traditional memory corruptions, including buffer overflows, double-free, use-after-free, etc. Hence, unprivileged users can exploit the memory corruption issue caused by the kernel data race to gain its privilege illegally.

Interestingly, due to the aforementioned challenges related to races, the exploitation of kernel data races is also challenging. Specifically, race exploitation requires precisely controlling the thread interleaving, but the kernel does not offer such a feature for users. Hence race exploitation in practice relies on a brute-force attack – i.e., simply keeping trying to trigger the race until success. Such a brute-force attack works for some kernel races and has been leveraged by most race-based privilege escalation attacks. For instance, as shown in Table 1, exploits known to the public annotated with a checkmark (✔) do so with the brute-force attack. However, we observe that the same brute-force attack is not effective at all for some types of kernel races (i.e., annotated with a triangle ▲).

In particular, we tested those hard-to-exploit cases, CVE-2019-1999 [23], and CVE-2019-2025 [24]. We confirmed that the brute-force exploitation fails, which tried 5 billion times and 15 million times of exploitation for 24 hours (more details in §7.1). It is worth noting that these two kernel races were confirmed to be vulnerabilities by kernel developers, but such confirmation is done by manually modifying the kernel – i.e., manually inserting a sleep function between racing memory accesses in hopes that the success chance of a brute-force attack increases. In other words, when those were confirmed, the testing environment was contrived, which cannot clearly state its exploitability in real-world.[1]

This paper proposes EXPRACE, a generic exploitation technique for kernel data races. To this end, we attempt to answer the following two research questions regarding kernel races; Q1: Why some kernel races are exploitable through a brute-force attack, while others are nearly impossible to exploit?; Q2: Would it be possible to develop a new exploitation technique that augments the exploitability of hard-to-exploit kernel races?

To answer the first question, we dissect the kernel data races into two categories – 1) a single-variable race and 2) a multi-variable race – and study the exploitability of each category. Our study found a specific set of multi-variable races (named *non-inclusive multi-variable races*), where its probability of successful brute-force exploitation is near zero. Specifically, exploitation of non-inclusive multi-variable races imposes a unique execution order to trigger, which is nearly impossible to occur without using extra debugging features (e.g., inserting a sleep or installing a breakpoint).

To answer the second question, we develop EXPRACE, a generic exploitation technique for non-inclusive multi-variable races. The key idea of EXPRACE is to keep raising interrupts to alter kernel thread's interleaving indirectly. This allows EXPRACE to transform hard-to-exploit multi-variable races into easy-to-exploit multi-variable races. Executing this idea involves several challenges. First, how to raise an interrupt from userspace? An interrupt mechanism is only controllable from the kernel, and clearly, it is not directly accessible from userspace. Second, even if EXPRACE is somehow able to raise an interrupt, how can it impact the thread interleaving in a controlled way? Races occur by multiple threads running on multiple CPU cores. It is unclear how to deliver such an interrupt to a specific thread to alter the thread interleaving for exploitation. Hence, we systematically analyze interrupt mechanisms in modern kernels, including Linux, Microsoft Windows, and Mac OS X. Then EXPRACE proposes four new race exploitation methods, where each leverages a different interrupt mechanism (i.e., rescheduling IPI, TLB shootdown IPI, membarrier IPI, and hardware interrupts).

In order to demonstrate the exploitation effectiveness, we evaluated EXPRACE with 10 real-world multi-variable races in Linux. Our evaluation results confirm that EXPRACE truly transformed hard-to-exploit races into easy-to-exploit races. While a brute-force attack without EXPRACE failed to exploit all of those for 24 hours, a brute-force attack with EXPRACE has successfully exploited all those 10. The time taken to succeed the exploitation varies depending on each exploitation method and vulnerability, but it takes from 10 seconds to 118 seconds.

We note that a clear understanding of the exploitability is the key for security risk assessment and management. In this regard, many works proposing new exploitation techniques (such as heap sprays [14, 51], ASLR breaking attacks [26], return-oriented programming [48], data-oriented programming [25], etc.) significantly impact and help to design secure systems. We believe EXPRACE also sheds a light on kernel race exploitation, which is a relatively under-explored vulnerability type but emerging threats.

To summarize, this paper makes the following contributions:

- **Analysis of Race Exploitability**. We analyzed kernel data races and found an intrinsic condition inherent to each race bug, classifying easy-to-exploit races and hard-to-exploit races.
- **Race Exploitation Methods**. EXPRACE presents several new race exploitation methods for modern OSes, including Linux, Microsoft Windows, and Mac OS X, against those hard-to-exploit races. EXPRACE indirectly induces the kernel to raise various interrupts, which transforms the hard-to-exploit races into easy-to-exploit races.
- **Evaluation with Real-World Races**. We used EXPRACE to exploit ten hard-to-exploit real-world kernel races. Our evaluation show that EXPRACE can exploit all of those within 10 to 118 seconds, while a simple brute-force attack without EXPRACE failed for all in given 24 hours.

The organization of this paper is as follows. §2 analyzes the race exploitability, and §3 describes the problem scope and research approaches of EXPRACE. Then §4 provides a background to understand EXPRACE. §5 presents race exploitation techniques for the Linux kernel, and §6 presents for Microsoft Windows and Mac OS X. §8 presents the discussion on EXPRACE. §7 evaluates EXPRACE, and §9 discusses the related work of this paper. §10 concludes the paper.

2 Exploitability of Kernel Data Races

A kernel data race is a concurrency bug in the kernel, which happens due to improper synchronization of data in its concurrent accesses. Data races in the kernel are notoriously challenging to exploit because its runtime behavior is inherently

[1] The term 'exploitability' in this paper means the reproducibility of the race, but we use 'exploitability' throughout the paper as precisely triggering a race is an essential step to exploit race condition vulnerabilities.

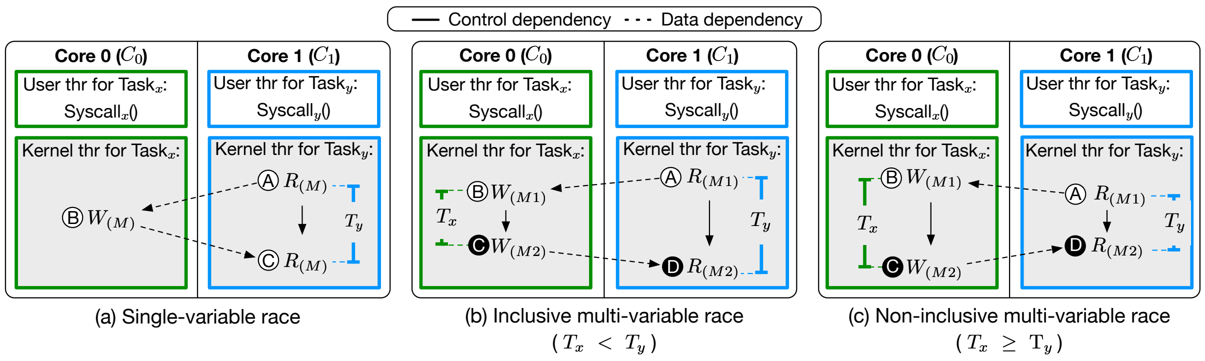


Figure 1: Categorization of kernel data races according to its execution order requirement. T_x and T_y denote the time taken between two instructions in $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$, respectively.

non-deterministic (e.g., impacted by core/thread scheduling orders). Since it involves complex thread interleaving, it is difficult to understand the root cause and reproduce for debugging. This, in fact, becomes a critical hurdle for adversaries who want to exploit data races, which is the key motivation of this paper

In the following, we dive into details of data races from an exploitation perspective. To this end, we first categorize data races into two common types [30, 38], a single-variable race, and a multi-variable race. Then we analyze the exploitability of each race type to motivate this paper.

2.1 Single-Variable Race

A single variable race is a concurrency bug pattern that another task violates atomicity over a single variable (but not correctly enforced by the code) in one task.

An example of a single variable race is illustrated in Figure 1-(a), where two tasks ($\texttt{Task}_x$ and $\texttt{Task}_y$[2]) are running on its own CPU core. These two tasks invoke $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$, respectively. $\texttt{Syscall}_x$'s handler executes an instruction Ⓑ, and $\texttt{Syscall}_y$'s handler executes two instructions Ⓐ and Ⓒ. All these three instructions access the same, single memory variable M. Under this setting, if Ⓑ overwrites the variable M in the middle of Ⓐ and Ⓒ (i.e., T_y, which denotes the time taken between two instructions in $\texttt{Syscall}_y$), the read operation in Ⓒ would get a different value of M compared to the value read in Ⓐ. In other words, the correct behavior may require the atomicity of M over T_y (i.e., the value of M should not change over T_y), but such atomicity is violated due to Ⓑ.

Exploiting Single-Variable Race. To exploit the single-variable race, one needs to precisely control the execution timing involving two kernel threads, where each kernel thread corresponds to $\texttt{Task}_x$ and $\texttt{Task}_y$, respectively, so that the execution order is in Ⓐ ≫ Ⓑ ≫ Ⓒ ($p \gg q$ denotes p happens before q). Since there is no way to precisely control kernel threads' execution order due to the non-deterministic scheduling behavior, brute-force attacks are typically the only exploitation option in practice. In other words, the attacker keeps invoking $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$ from user threads of $\texttt{Task}_x$ and $\texttt{Task}_y$, respectively, in hopes that Ⓑ is executed within T_y at some point.

Hence, the probability of successful exploitation (i.e., P_{single}) is roughly $\frac{T_y}{T_{\texttt{Syscall}_x}}$, where $T_{\texttt{Syscall}_x}$ denotes the time taken to handle each $\texttt{Syscall}_x$. Here, we focus on capturing the exploitation probability when invoking a single $\texttt{Syscall}_y$. Thus, we assume that if $\texttt{Syscall}_x$ terminates earlier than $\texttt{Syscall}_y$, the same $\texttt{Syscall}_x$ invocation keeps being performed until $\texttt{Syscall}_y$ terminates, which is still a typical brute-force exploitation strategy. Please note that we do not consider the case $T_{\texttt{Syscall}_x} \leq T_y$, because in most of single variable race condition, $T_{\texttt{Syscall}_x}$ is bigger than T_y. Although P_{single} may seem low, it clearly implicates that the exploitation is feasible with many trials. In fact, the goal of most adversaries is completed if taking over the system once. Thus the brute-force exploitation is effective and sufficient to exploit a single-variable race. Roughly interpreting the practical implication of P_{single}, the maximum number of $T_{\texttt{Syscall}_x}$ is about 1 M cycles, and T_y is about 10 cycles according to our evaluation. Then the brute-force would certainly succeed if $\texttt{Syscall}_y$ can be invoked more than 100 K times, which can be mostly completed within one minute. Taking the real-world exploits as another example, available privilege escalation exploits (including CVE-2017-7533 [45], CVE-2017-2636 [44], and CVE-2016-8655 [43]) succeed the brute-force exploitation ranging from 5 to 30 seconds.

2.2 Multi-Variable Race

A multi- variable race violates the atomicity involving multiple variables. Taking the example in Figure 1-(b), suppose instruction Ⓐ and Ⓑ access the variable M_1, and instruction

[2]In this paper, Task can refer to both a user process (or a heavyweight user process) or a user thread (or a lightweight user process) if not explicitly stated. The reason why we use this neutral term Task is that many race cases are working the same for both the process and thread. For those race cases showing different restrictions, we will specifically mention which one we refer to.

🅒 and 🅓 access the variable M_2. In this case, if 🅒 overwrites M_2 in the middle of Ⓐ and 🅓 (i.e., T_y), 🅓 would get different M_2 compared to the case not overwritten by 🅒, leading to the atomicity violation. Here, the key difference from a single variable race is that such an atomicity violation involves multiple variables. Thus, in turn, the atomicity violation condition imposes a more strict execution order: Ⓐ $\gg$ Ⓑ $\gg$ 🅒 $\gg$ 🅓.

We note that it is reported that 34% of data races are multi-variable races [38]. Particularly focusing on races in the kernel, we suspect this is related to the fact that the kernel often accesses the data in the following two steps, where each step involves its own race variable: (i) the kernel first searches for a location (i.e., a virtual address) holding data of interest. This search process typically involves enumerating over a well-defined data structure, such as a list, tree, etc.; (ii) once identifying the location, the kernel either fetches the data (i.e., a read operation) or updates the data (i.e., a write operation).

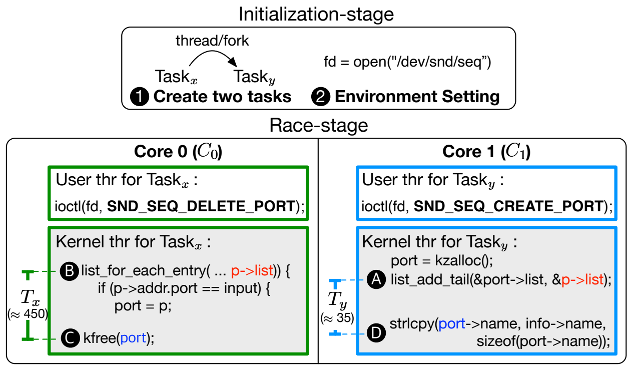
Challenge: Exploiting Multi-Variable Race. Similar to exploiting the single-variable race, the best exploitation strategy would be brute-forcing the multi-variable race. In other words, the attacker keeps invoking $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$ in hopes that the timeline of T_x (in $\texttt{Task}_x$) is placed within T_y (in $\texttt{Task}_y$). Here, T_x denotes the time taken between two instructions in $\texttt{Syscall}_x$. As such, the probability of successful exploitation (denoted as P_{multi}) would roughly be like below:

$$P_{\text{multi}} = \begin{cases} \frac{T_y - T_x}{T_{\texttt{Syscall}_x}} & \text{if } T_x < T_y \\ 0 & \text{if } T_x \geq T_y. \end{cases}$$

This probability model assumes that if $\texttt{Syscall}_x$ terminates earlier than $\texttt{Syscall}_y$, the same $\texttt{Syscall}_x$ keeps invoked to complete the brute-force attack strategy (the opposite case is also possible, but the probabilistic model is roughly the same). If $T_x < T_y$ (depicted in Figure 1-b), P_{multi} is that T_x's timeline is completely overlapped by that of T_y while running each $\texttt{Syscall}_y$. Compared to P_{single}, this success probability would not be much different from the attacker's perspective – as we mentioned before, a typical goal of an adversary is to take over the system just once.

On the contrary, if $T_x \geq T_y$ (depicted in Figure 1-c), P_{multi} is near zero with brute force. That is because it is virtually impossible to satisfy both execution orders when $T_x \geq T_y$: Ⓐ $\gg$ Ⓑ and 🅒 $\gg$ 🅓.

As a result, security researchers who report a multi-variable race vulnerability often use an extra debugging feature to confirm the exploitability (i.e., making the proof-of-concept exploit similar to when $T_x < T_y$). However, such an extra debugging feature cannot be used from userspace, so such a proof-of-concept exploit is far from confirming the real-world exploitability. For instance, to confirm CVE-2019-1999 [23] and CVE-2019-2025 [24], researchers have manually inserted a sleep function between Ⓐ and 🅓 (i.e., modified the kernel code) to increase T_y intentionally. Another example is to intentionally install a breakpoint between Ⓐ and 🅓, artificially



```
fd = open("/dev/snd/seq")
```

User thr for Task_x :

```
ioctl(fd, SND_SEQ_DELETE_PORT);
```

Kernel thr for Task_x :

```
🅑 list_for_each_entry( ... p->list)) {
      if (p->addr.port == input) {
         port = p;

🅒 kfree(port);
```

User thr for Task_y :

```
ioctl(fd, SND_SEQ_CREATE_PORT);
```

Kernel thr for Task_y :

```
   port = kzalloc();
🅐 list_add_tail(&port->list, &p->list);

🅓 strlcpy(port->name, info->name,
                 sizeof(port->name));
```

Figure 2: A simplified multi-variable race: CVE-2017-15265

enforcing the execution orders. Note that above mentioned debugging methods are also used in automated race detection or fuzzing systems such as [19, 27, 29].

In the following, we use a real-world multi-variable race vulnerability, CVE-2017-15265, to clearly describe why this case is nearly impossible to exploit in practice.

Real-world Multi-Variable Race: CVE-2017-15265. CVE-2017-15265 is a multi-variable race condition in the Linux kernel's sound driver (illustrated in Figure 2). In this race condition, we assume that there are two tasks, $\texttt{Task}_x$ and $\texttt{Task}_y$ (as shown in ❶), where $\texttt{Task}_x$ and $\texttt{Task}_y$ can be either the same process (i.e., one is created by `pthread_create()` from another) or different processes (i.e., created through `fork()`).

Then either $\texttt{Task}_x$ or $\texttt{Task}_y$ (or both if $\texttt{Task}_x$ and $\texttt{Task}_y$ are different processes) opens the sound driver to get its file descriptor (i.e., ❷). Using this file descriptor, $\texttt{Task}_y$ requests to allocate a new port by invoking `ioctl()` with a create command (i.e., `SND_SEQ_CREATE_PORT`). In response to this allocation request, the kernel thread for $\texttt{Task}_y$ allocates a new buffer (i.e., `port`) and then inserts that new buffer to the list (i.e., `p->list`), which shown in 🅐. Then the buffer (i.e., `port->name`) is initialized as user input in 🅓.

Simultaneously, $\texttt{Task}_x$ attempts to free the port, which is allocated by $\texttt{Task}_y$ through `ioctl()` with a free command (i.e., `SND_SEQ_DELETE_PORT`). In response to the free request, the kernel thread for $\texttt{Task}_x$ finds the corresponding buffer (i.e., `port`) from the list (i.e., `p->list`), which was also referenced by $\texttt{Task}_y$ (🅑). Then it frees the buffer (🅒).

Suppose these two tasks perform the behaviors above. In that case, it may result in a multi-variable race, which involves the following two variables: i) `p->list`, which is accessed by 🅐 and 🅑, and ii) `port`, which is accessed by 🅒 and 🅓. More specifically, the atomicity is violated if the execution order follows 🅐 $\gg$ 🅑 $\gg$ 🅒 $\gg$ 🅓. Under this execution order, $\texttt{Task}_y$ assumes that when it invokes 🅓, `port` is active. However, `port` is already freed by $\texttt{Task}_x$ since no synchronization method, such as `spinlock` is used to retain `port`. Therefore, $\texttt{Task}_y$ uses `port` after being freed, resulting in a well-known memory corruption issue, use-after-free.

To fully exploit this vulnerability, we need to trigger the use-after-free vulnerability three times. We first spray the `file` pointers through `msgsnd()`. Next, we trigger the vulnerability to partially overwrite the `struct snd_seq_prioq *tickq` within `struct snd_seq_queue` to leak the sprayed `struct file` pointer. Then, we trigger the vulnerability to overwrite `struct iovec` [52] to perform the arbitrarily address read attack, which reads `struct *f_cred` within `struct file`. Finally, we trigger the vulnerability to overwrite `struct iovec` once more to perform the arbitrary address write with the value zero, which eventually overwrites the root privilege to the credential structure (i.e., `struct cred`). This completes the privilege escalation attack.

However, exploiting CVE-2017-15265 through bruteforcing is virtually infeasible because P_{multi} is zero when $T_x > T_y$. More specifically, we observed that T_x is about 12 times longer than T_y, according to our evaluation (§7.1), because there are many instructions executed in between Ⓑ and Ⓒ. Our evaluation also has shown that the brute-force exploitation fails even after trying for 24 hours, confirming that it is nearly impossible to meet the violation execution order.

3 Problem Scope and Research Approaches

3.1 Problem Scope

This paper proposes ExpRace, which aims at developing a practical exploitation method for a non-inclusive multi-variable race (i.e., when $T_x > T_y$ as shown in §2). ExpRace assumes a typical privilege escalation attack scenario—escalating its privilege from the user to the kernel privilege, where an adversary already has access to the user privilege so that she/he can invoke system calls that an underlying kernel provides. As such, ExpRace does not assume that the adversary has the kernel privilege, meaning that the adversary cannot leverage any kernel debugging features. Under this assumption, ExpRace develops user-level applications which are designated to attack race issues. In particular, ExpRace presents exploitation methods for such race issues in the modern kernel, including Linux (§5), Microsoft Windows (§6.1), and Mac OS X (§6.2).

A privilege escalation attack exploiting a race vulnerability mostly takes the following two steps: 1) triggering a race, which leads to memory corruption (such as buffer overflows, double-free, use-after-free, etc.); 2) exploiting a memory corruption, which accordingly hijacks the control-flow (such as ROP attacks [48]) or data-flow (such as DOP attacks [25]) to escalate the privilege eventually. We focus on the first step, triggering the race, and we do not cover the second step, exploiting memory corruption. This is because the second step is not related to generic race issues but related to an exploitation technique for a specific memory corruption issue studied by many previous works [14, 25, 48, 51, 64].

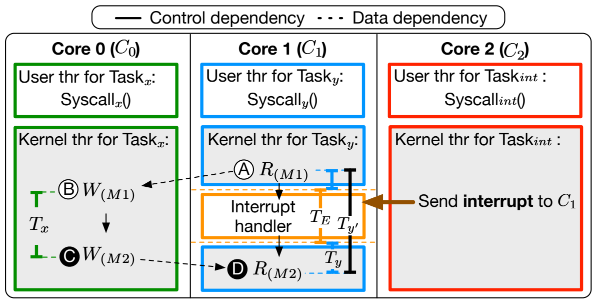


Figure 3: A research approach of ExpRace to exploit a non-inclusive multi-variable race. Using `Task`$_{int}$, ExpRace indirectly causes the kernel to raise an interrupt, which in turn enlarges the original T_y and thus transformed into exploiting an inclusive multi-variable race.

3.2 Research Approaches

The key insight behind ExpRace is in intentionally enlarging T_y in order to transform the hard-to-exploit non-inclusive multi-variable race into the easy-to-exploit inclusive multi-variable race (illustrated in Figure 3). To this end, ExpRace attempts to enlarge T_y by raising an interrupt. Specifically, T_y can be enlarged if following two conditions meet: 1) correct interrupt destination: an interrupt should be delivered to the CPU core running the kernel thread of `Task`$_y$; and 2) precise interrupt timing: an interrupt should be delivered when the kernel thread of `Task`$_y$ executes an instruction between Ⓐ and Ⓓ; If these two conditions were met, the core received the interrupt will switch to the interrupt handler (so as to immediately handle the interrupt), and after completing the interrupt handling, that core will switch back to the kernel thread of `Task`$_y$. As a result, due to the time handling the interrupt (annotated as T_E), the original T_y will be enlarged. We denote such an enlarged time window as $T_{y'}$ such that $T_{y'} = T_y + T_E$, and we call $T_{y'}$ as *a race window*.

To clearly understand the theoretical aspect of this exploitation method, we model the probability of successful exploitation as $P_{\text{multi}}^{\text{ExpRace}}$. This probability is modeled under the assumption that ExpRace can control that the interrupt can be delivered to the destined core. We further assume that for each `Syscall`$_y$ invocation, both `Syscall`$_x$ and `Syscall`$_{int}$ kept being executed without any noise.

$$P_{\text{multi}}^{\text{ExpRace}} = \begin{cases} \dfrac{T_y}{T_{\texttt{Syscall}_{int}}} & \text{if} \quad T_{\texttt{Syscall}_x} \leq T_{y'} \\[2ex] \dfrac{T_y}{T_{\texttt{Syscall}_{int}}} * \dfrac{T_{y'} - T_x}{T_{\texttt{Syscall}_x}} & \text{if } T_{\texttt{Syscall}_x} > T_{y'} \textit{ and } T_x < T_{y'} \\[2ex] 0 & \text{if } T_{\texttt{Syscall}_x} > T_{y'} \textit{ and } T_x \geq T_{y'}. \end{cases}$$

For the first case (i.e., $T_{\texttt{Syscall}_x} \leq T_{y'}$), the exploitation would be successful if an interrupt is raised within T_y, because T_x is always overlapped within the race window $T_{y'}$. For the second case (i.e., $T_{\texttt{Syscall}_x} > T_{y'}$ and $T_x < T_{y'}$), following two events should occur together to be a successful exploit: i) an interrupt is raised within T_y and ii) T_x is overlapped with the enlarged race window $T_{y'}$ during each `Syscall`$_x$ execution. For the last case (i.e., $T_{\texttt{Syscall}_x} > T_{y'}$ and $T_x < T_{y'}$), the probability is zero because T_x is too large to be overlapped within the extended race window, so the race would never occur.

Research Challenges. Since an interrupt mechanism is only controllable from the kernel and thus non-controllable from the user, following research challenges are arising to accomplish EXPRACE. First, how to raise an interrupt which can be used with user privileges and affect kernel mode execution? There is no direct way to raise an interrupt since modern kernels limit user's control over interrupts. Moreover, there are many different types of interrupts (from IPI to hardware interrupts), and we do not know if any of those are controllable at some extent by users. Second, suppose EXPRACE is somehow able to raise an interrupt, but how does EXPRACE deliver the interrupt to the destined core? Modern kernels are heavily optimizing its interrupt handling mechanism, as it is very critical to its runtime responsiveness. As such, its core affinity with respect to interrupt handling can be very different for each type of interrupt, challenging EXPRACE for exploiting non-inclusive multi-variable races.

4 Interrupt Handling in Linux

The exploitation mechanism of EXPRACE highly depends on how it triggers an interrupt. Hence, this section provides the necessary background on the interrupt handling mechanism before presenting EXPRACE's exploitation methods using an indirect interrupt raising mechanism (§5). Specifically, we describe the basic mechanism of interrupts and its different types in this section. Note that most of the descriptions in this section are Linux specific. Since its general working mechanism is similar in other OSes, we will clearly state its differences when describing the exploitation methods for non-Linux systems in §6.

An interrupt is an input signal delivered to the processor, notifying an event that requires immediate attention. As such, an interrupt diverts the normal execution flow since a CPU core, which received the interrupt, first handles the interrupt after temporarily stopping the execution. While there are many different types of interrupts, we focus on hardware interrupts and inter processor interrupts (IPIs), which are the most relevant to EXPRACE's exploitation techniques. Hardware interrupt request (IRQ) is an electric signal sent from an external hardware device to a processor through IO-APIC. This facilitates communication with operating systems. Inter Processor Interrupt (IPI) is a special type of an interrupt in multi-processor systems, which delivers the command from a CPU core to another. In Linux, there are several different types of IPIs, rescheduling IPI, wake-up IPI, stop IPI, function call IPI, etc., and each IPI transfers its own command across CPU cores. Similar to the hardware interrupt, upon receiving the IPI the respected CPU core immediately starts processing the IPI.

It is worth noting that both hardware interrupts and IPIs cannot be raised from user-level code, which we attempt to address in the next section (§5).

Method	**Relation b/w** `Task`$_x$ & `Task`$_y$	**User functions** to send an interrupt	**Metadata** determining a core to receive interrupts
Resched IPI	thread or process	`sched_setaffinity()`, `read()` – `write()`	`cpu_set_t *mask`, Wait process's core affinity
Func Call IPI (TLB shootdown)	process	`mprotect()`, `munmap()`	`struct mm_struct`'s `cpu_bitmap`
Func Call IPI (`membarrier`)	process	`membarrier()`	`struct mm_struct`'s `membarrier_state`
HW interrupt	thread or process	Send request to a device	HW interrupt's affinity

Table 2: A list of EXPRACE's exploitation methods in Linux

5 Exploiting Kernel Races in Linux

This section proposes EXPRACE, a new race exploitation technique, which extends the race window by indirectly raising interrupts. The rest of this section presents various exploitation methods, particularly focusing on Linux systems: (i) using reschedule IPI (§5.1); (ii) using non-reschedule IPI (TLB Shootdown IPI (§5.2.1) and `membarrier` IPI (§5.2.2)); and (iii) using hardware interrupts (§5.3).

For each IPI method, we first briefly introduce its basic working mechanism. Then we present our technique to send a corresponding IPI from the user-space. Lastly, we describe step-by-step guides to extend the race window for race exploitation.

5.1 Reschedule IPI

Reschedule IPI sends a rescheduling request from one core to another. Depending on which preemption mode the kernel has been configured, the responsive behavior is different. If `CONFIG_PREEMPT` [16] option is enabled, the core received an IPI immediately performs the rescheduling unless preemption is not explicitly forbidden through `preempt_disable()`, which in turn raises the context switch to another process. Otherwise, the rescheduling will be deferred until the task (running on the core which received the IPI) either yields the schedule voluntarily or reaches a pre-configured preemption point. However, this option is not affecting the result of our attack.

Sending Reschedule IPI from Userspace. Reschedule IPI is the IPI sent by the internal kernel function, `smp_send_reschedule()`. `smp_send_reschedule()` takes an

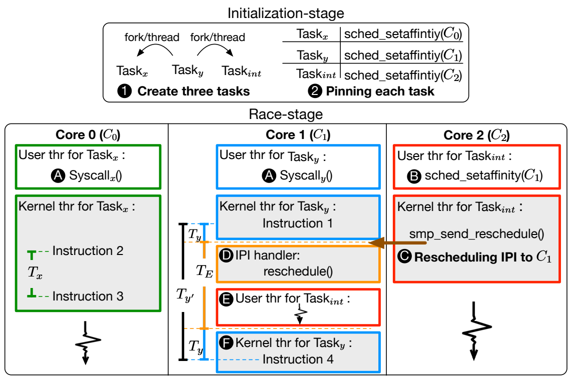


Figure 4: EXPRACE's exploitation method using Reschedule IPI

argument cpu, which specifies which core should receive the IPI.

We found two different methods to trigger `smp_send_reschedule()` from the user space in a controlled way (i.e., specifying a specific core or task). One method is to invoke the `sched_setaffinity()` syscall. This syscall takes an argument `pid` and `mask`, which eventually sets `cpu` of `smp_send_reschedule()` as a core running a process with the specified `pid`.

Another method is through waking up a waiting thread, which can be done as follows: 1) assign a specific core affinity to task A and change the thread's process state to waiting through a syscall such as `read()`, and; 2) wakes up the waiting thread from task B through a syscall such as `write()`. The kernel then changes task A's process state waiting to running and sends a reschedule IPI to the core, which thread A has an affinity.

Although both methods have the same result that sending rescheduling IPI to a specific core, the first method yields better performance than the second method for the two reasons. First, the wait-wake method should use two processes for sending IPI, but the method using `sched_setaffintiy()` uses one process. Second, the `sched_setaffintiy()` method can send more IPIs during the same time period than the wait-wake method. More specifically, because the wake process must wake after the wait process is completely waiting, however, the wake process doesn't have knowledge about wait process's process state immediately; it needs a time for synchronization.

Extending Race Window with Reschedule IPI. Using reschedule IPI, a race window can be extended with the following steps, as shown in Figure 4 (simplified implementation code is shown in Figure A.1). First, we create three tasks ($\texttt{Task}_x$, $\texttt{Task}_y$, $\texttt{Task}_{int}$), where $\texttt{Task}_x$ and $\texttt{Task}_{int}$ can be either a child process or concurrent thread of $\texttt{Task}_y$, respectively (shown in ❶). $\texttt{Task}_x$ and $\texttt{Task}_y$ will be used to invoke two racy syscalls (i.e., $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$), and these two syscalls are assumed to raise a data race. $\texttt{Task}_{int}$ will be used to send the reschedule IPI to C_1. Note that we assume an attacker has full controls over all these user-level three tasks.

Next, each task is pinned to a specific core (i.e., $\texttt{Task}_x$ is pinned to C_0, $\texttt{Task}_y$ to C_1, and $\texttt{Task}_{int}$ to C_2) by invoking `sched_setaffinity()` (❷). Each task is pinned to a different core so as to avoid interference by each other, thereby easily enlarging the race window. After that, $\texttt{Task}_x$ and $\texttt{Task}_y$ invoke a race-raising syscall, i.e., $\texttt{Syscall}_x$ by $\texttt{Task}_x$ and $\texttt{Syscall}_y$ by $\texttt{Task}_y$ (Ⓐ). Then $\texttt{Task}_{int}$ invokes $\texttt{sched\_setaffinity}(C_1)$ (Ⓑ). This makes the kernel to (i) migrate $\texttt{Task}_{int}$ from C_2's run queue to C_1's run queue and (ii) send a reschedule IPI to C_1 (Ⓒ).

At this moment, if C_1 receives the reschedule IPI when it was executing any instruction within the race window (i.e., within T_y), C_1 stops executing $\texttt{Task}_y$ to handle the IPI (Ⓓ). After handling the IPI, it performs a context switch to $\texttt{Task}_{int}$ because this is what is instructed by the IPI (Ⓔ). As a result, the race window, T_y, is extended until $\texttt{Task}_{int}$ is switched out, and $\texttt{Task}_y$ is scheduled in again (Ⓕ).

5.2 Non-Reschedule IPI

Non-reschedule IPI refers to an IPI which is not related to rescheduling. Non-reschedule IPI can be raised to send the following commands: 1) TLB management and 2) memory barriers. We found race-window extending methods using either TLB management (§5.2.1) or memory barriers (§5.2.2), as we describe next.

5.2.1 TLB Shootdown IPI

Translation Lookaside Buffer (TLB) is a cache for translating an address from virtual to physical. Since each CPU core has its own TLB, all TLB entries across different cores should be synchronized in multi-processor systems. Otherwise, incorrect translation may be performed by the core, which refers to outdated TLB entries (i.e., one core updates access permissions or release the memory page, but such an update is not accordingly synchronized with other cores' TLB entries). As such, modern operating systems implement a TLB shootdown mechanism to ensure that TLB entries are synchronized across different cores.

In order to implement the TLB shootdown mechanism, x86-based operating systems rely on TLB shootdown IPI [3]. More specifically, since the kernel code running on one CPU core cannot directly flush the TLB of other CPU cores, it sends TLB shootdown IPI to other CPU cores. Once receiving the IPI, the recipient CPU core immediately stops the currently running task and flushes the TLB such that it does not refer to outdated TLB entries.

More specifically, if any page table entry is to be updated by a CPU core, the kernel has to send IPI to other CPU cores

[3]Not all architectures rely on IPI to implement the TLB shootdown. For instance, ARM supports the `tlbi` instruction, which flushes the TLB of all CPU cores.

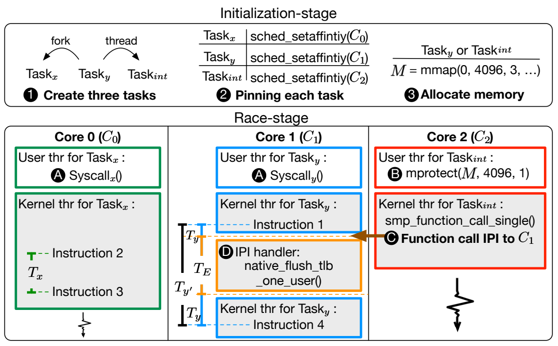

Initialization-stage
fork
thread
Task$_x$
Task$_y$
Task$_{int}$
❶ Create three tasks
Task$_x$ | sched_setaffintiy(C_0)
Task$_y$ | sched_setaffintiy(C_1)
Task$_{int}$ | sched_setaffintiy(C_2)
❷ Pinning each task
Task$_y$ or Task$_{int}$
M = mmap(0, 4096, 3, ...)
❸ Allocate memory
Race-stage
Core 0 (C_0)
Core 1 (C_1)
Core 2 (C_2)
User thr for Task$_x$: Ⓐ Syscall$_x$()
User thr for Task$_y$: Ⓐ Syscall$_y$()
User thr for Task$_{int}$: Ⓑ mprotect(M, 4096, 1)
Kernel thr for Task$_x$:
Instruction 2
T_x
Instruction 3
Kernel thr for Task$_y$: Instruction 1
Ⓓ IPI handler: native_flush_tlb _one_user()
Kernel thr for Task$_y$: Instruction 4
Kernel thr for Task$_{int}$: smp_function_call_single()
Ⓒ Function call IPI to C_1
T_y
T_E
$T_{y'}$
T_y

Figure 5: EXPRACE's exploitation method using TLB shootdown IPI

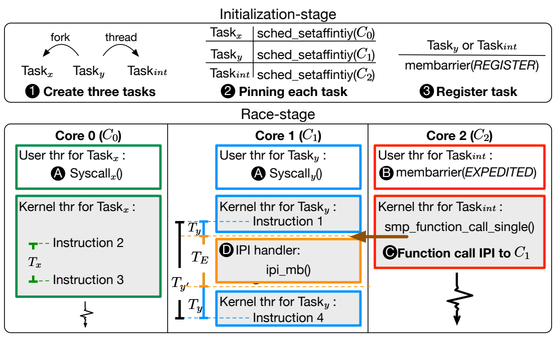

Initialization-stage
fork
thread
Task$_x$
Task$_y$
Task$_{int}$
❶ Create three tasks
Task$_x$ | sched_setaffintiy(C_0)
Task$_y$ | sched_setaffintiy(C_1)
Task$_{int}$ | sched_setaffintiy(C_2)
❷ Pinning each task
Task$_y$ or Task$_{int}$
membarrier(REGISTER)
❸ Register task
Race-stage
Core 0 (C_0)
Core 1 (C_1)
Core 2 (C_2)
User thr for Task$_x$: Ⓐ Syscall$_x$()
User thr for Task$_y$: Ⓐ Syscall$_y$()
User thr for Task$_{int}$: Ⓑ membarrier(EXPEDITED)
Kernel thr for Task$_x$:
Instruction 2
T_x
Instruction 3
Kernel thr for Task$_y$: Instruction 1
Ⓓ IPI handler: ipi_mb()
Kernel thr for Task$_y$: Instruction 4
Kernel thr for Task$_{int}$: smp_function_call_single()
Ⓒ Function call IPI to C_1
T_y
T_E
$T_{y'}$
T_y

Figure 6: EXPRACE's exploitation method using `membarrier` IPI

having the same entry. Thus, the kernel refers to `cpu_bitmap` in `mm_struct`, which has the list of cores that may have the same page table entry [2].

Sending TLB Shootdown IPI from Userspace. From userspace, TLB shootdown can be triggered through syscalls that update the page table, such as `mprotect()` or `munmap()`. These syscalls first flush the TLB of the currently running CPU core, and then send TLB shootdown IPI to other CPU cores. Note that the IPI will be sent to the CPU cores, which may have out-dated TLB entries as the kernel maintains the information on which CPU cores may have out-dated TLBs (i.e., `cpu_bitmap` in `mm_struct`).

Extending Race Window with TLB Shootdown IPI. Leveraging TLB shootdown IPI, the race window can be extended through the following steps, as shown in Figure 5 (simplified implementation code is shown in Figure A.2). First, three tasks, Task$_x$, Task$_y$, and Task$_{int}$ are created (shown in ❶), where Task$_x$ should be the child process of Task$_y$ and Task$_{int}$ should be a concurrent thread of Task$_y$. Task$_x$ and Task$_y$ are for invoking race-triggering syscalls, Syscall$_x$ and Syscall$_y$, respectively, and Task$_{int}$ is for sending the TLB shootdown IPI.

Note that Task$_x$ and Task$_y$ must not be the same process (i.e., created through `fork()`, not through `pthread_create()`). This is because Task$_x$ and Task$_y$ should not refer to the same `mm_struct`. Also, using `fork()` ensures that Task$_x$ and Task$_y$ have their own copy of `mm_struct`. If Task$_x$ and Task$_y$ are the same processes (but different threads), they would reference the same `mm_struct`. In this case, `cpu_bitmap` is set for both Task$_x$ and Task$_y$, so IPI will be sent to both C_0 (the core running Task$_x$) and C_1 (the core running Task$_y$). For a similar reason, Task$_y$ and Task$_{int}$ should be the same process.

Next, each task is pinned to different cores using `sched_setaffinity()`, i.e., Task$_x$ is pinned to C_0, Task$_y$ to C_1, and Task$_{int}$ to C_2 (❷). Then either Task$_y$ or Task$_{int}$ allocates a memory page (say M) using `mmap()`, which will be used to raise the TLB shootdown (❸).

After that, Task$_x$ and Task$_y$ invoke race-raising syscalls, Syscall$_x$, and Syscall$_y$, respectively (Ⓐ). At this moment, if Task$_{int}$ modifies the permission of the previously allocated memory page (i.e., M) using `mprotect()` (Ⓑ), the kernel first flushes the TLB of C_2. Moreover, the kernel also sends a function call IPI to C_1 since C_1 is set in `cpu_bitmap` in `struct mm_struct` for the M (Ⓒ). If C_1 receives the function call IPI when executing the race window (i.e., within T_y), C_1 immediately stops executing Task$_y$ and starts handling IPI (Ⓓ). As a result, the race window is extended until the end of IPI handling (which is performed through `native_flush_tlb_one_user()`).

5.2.2 Memory Barrier IPI

`membarrier` in Linux is a syscall, controlling the memory access orders in multi-processor systems. Since `membarrier` needs to activate a memory barrier on specific threads, it relies on an IPI mechanism to notify specific cores running those threads.

Sending Memory Barrier IPI from Userspace. Unlike other IPIs that we introduced before, the Linux kernel provides the syscall interface `membarrier`, which sends the memory barrier IPI from the user space. Thus, a user task can invoke the syscall `membarrier` to deliver the memory barrier IPI.

Extending Race Window with Memory Barrier IPI. In order to extend the race window, we utilize membarrier syscalls in the following steps as shown in Figure 6 (implementation code is shown in Figure A.3). First, two tasks, Task$_x$ and Task$_y$, are created (which will execute race-raising syscalls) as well as Task$_{int}$ to send Memory Barrier IPI (shown in ❶). Here, since Task$_x$ and Task$_y$ must have different `mm`, Task$_x$ is created through `fork()` from Task$_y$. On the contrary, since Task$_y$ and Task$_{int}$ must have the same `mm`, Task$_{int}$ is created through `pthread_create()` from Task$_y$. Next, each task is pinned to its own core using `sched_setaffinity()` (❷). Then Task$_y$ or Task$_{int}$ invokes the membarrier syscall to register the process to use memory barrier (❸).

Task$_x$ and Task$_y$ invoke race-raising syscalls, Syscall$_x$, and Syscall$_y$, respectively (Ⓐ). After that, as Task$_{int}$ invoke the membarrier syscall with expedited option (Ⓑ), the

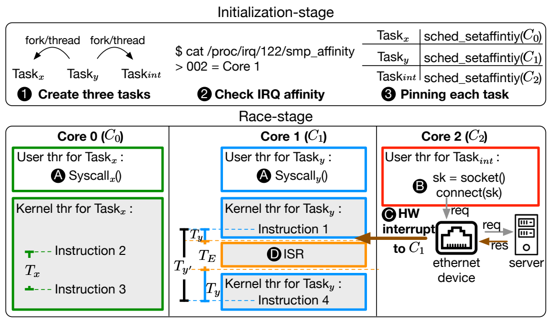


Figure 7: ExpRace's exploitation method using HW interrupts

kernel sends the membarrier IPI to C_1 (Ⓒ). This is because `Task`$_y$ (which is running on C_1) reference the same `struct mm_struct` as `Task`$_{int}$. Once receiving the membarrier IPI, `Task`$_y$ is switched out from C_1 to handle the IPI through `ipi_mb()` (Ⓓ). If this IPI is delivered when executing T_y, the race window is extended.

5.3 Hardware Interrupts

Hardware interrupt request (IRQ) is an electric signal sent from an external hardware device to a processor through IO-APIC. This facilitates communication with operating systems.

Sending Hardware Interrupts from Userspace. If the IRQ is issued, an interrupt controller delivers the interrupt to a certain CPU core to handle the interrupt, which in turn executes an interrupt service routine (ISR). The interrupt controller allows the kernel to specify which CPU core is responsible for which interrupt through bit masking. This allows the kernel to optimize the performance as it directly delivers an interrupt to a dedicated core instead of selecting the core using some other algorithm (e.g., a round-robin).

In Linux, such a specification can be checked by reading the file in `procfs`, `/proc/irq/#/smp_affinity`, where # denotes an IRQ number. Taking an example in our experimental environment, the default kernel configuration is that the `enp2s0` device is assigned to IRQ 122, which is destined to be served by CPU core 11.

This IRQ cannot be sent from userspace directly because it requires the kernel privilege. Therefore, we devise an indirect way to send the IRQ from userspace: i) send a request from userspace to a device; ii) in response to the request, the device issues the IRQ to the kernel. We found several different indirect ways: 1) send TCP request to the ethernet device and the device issues IRQ to the kernel to process the packet; 2) send disk request using file read or write, disk controller device(e.g., AHCI device) issues IRQ to the kernel to signal that a disk request has been fulfilled.

Extending Race Window with Hardware Interrupts. The race window can be extended with the following steps, as shown in Figure 7 (implementation code is shown in Figure A.5). First, two tasks, `Task`$_x$ and `Task`$_y$, are created (which will execute race-raising syscalls) as well as `Task`$_{int}$ to send the TCP request to the ethernet device (shown in ❶). Note that `Task`$_x$ and `Task`$_{int}$ can be either a child process or concurrent thread of `Task`$_y$, respectively, because HW interrupt delivery mechanism is irrespective of its process/thread relationship. Next, by checking `/proc/irq/#/smp_affinity`, we retrieve the CPU core number, which has an affinity to the subjected IRQ (❷). To simplify the description, we assume that the ethernet device has an affinity for cpu C_1.

Then, each task is pinned to a specific core (❸). After that, `Task`$_x$ and `Task`$_y$ invoke a race-raising syscall, i.e., `Syscall`$_x$ by `Task`$_x$ and `Syscall`$_y$ by `Task`$_y$ (Ⓐ). `Task`$_{int}$ sends the TCP request to itself (i.e., an external IP address of a local machine) (Ⓑ). Then in order to process the request packet, the ethernet device issues an IRQ to C_1 (Ⓒ).

If C_1 receives the IRQ within the time frame of T_y, the kernel thread for `Task`$_y$ switches to the corresponding interrupt service routine (ISR) (Ⓓ). After completing the ISR, C_1 returns back to the kernel thread for `Task`$_y$. As a result, the race window (T_y) is extended as much as the execution time of the ISR.

OS	Reschedule IPI	Function Call IPI (TLB shootdown)	HW interrupt
Windows	✓	✓	✓
OS X	✗	✓	✗

Table 3: ExpRace's exploitation summary on other OSes

6 Exploiting Kernel Races in Other OSes

In order to understand if the race window extension mechanism proposed in §5 works for other operating systems, this section studies if it also works for two other popular operating systems: Microsoft Windows (§6.1) and MAC OS X (§6.2). The research challenge here is that these are proprietary kernels, so their detailed internal mechanism is difficult to understand. Note that we have studied all the methods except `membarrier` (§5.2.2), as `membarrier` is a unique feature only available in Linux.

To summarize (shown in Table 3), in the case of Windows we confirmed that all the race enlarging methods (except `membarrier`) presented in §5 also work. In the case of MAC OS X, we confirmed that the TLB shootdown IPI works but reschedule IPI would not work. We were not able to do a meaningful study on HW interrupts on OS X due to the lack of internal information.

6.1 Microsoft Windows

Reschedule IPI. We found that Windows's preemption mode is mostly similar to that of Linux's `CONFIG_PREEMPT` mode. The key difference between Windows and Linux is that Windows takes account of thread's priority [42]. More precisely, if a new thread is enqueued for rescheduling in Windows, that new thread is only rescheduled if it has a higher priority than a currently running thread. As a result, Windows keeps elevating the priority of that new thread so that it can take a chance to be rescheduled.

Therefore, compared to the reschedule IPI method for Linux (§5.1), we modified two things to work for Windows: (i) use a different syscall for setting up a thread's CPU affinity (i.e., `SetThreadAffinityMask()` in Windows) and (ii) additionally invoke a syscall to set the high priority using `SetThreadPriority()`.

TLB Shootdown IPI. Similar to Linux, the TLB shootdown mechanism is carried out by sending IPI. Thus, we confirmed that the race window could be mainly extended by the same method introduced in §5.2.1. The difference is platform-dependent syscall uses (more precisely, WinAPI in the Windows terminology), i.e., we used `VirtualAlloc()`, `VirtualProtect()`, and `VirtualFree()` to allocate, modify, and free the memory page, respectively.

HW Interrupt. Windows also offers a kernel feature that each device driver can configure an affinity policy, such that each can declare a set of CPU cores to serve relevant hardware interrupts. Specifically, the affinity policy can be configured in the device's INF file or registry settings (e.g., `#Device ParametersInterrupt registry` [41]).

However, in our experimental setup (§7), all device drivers installed on the machine are configured to have no core affinity. In other words, all device drivers are handled in a round-robin order. Although this may imply that the extension method with HW interrupt (§5.3) cannot be used for Windows, still this brings a significant benefit in extending the race window. Theoretically, if there are k different cores in the machine, the HW interrupt-based method in Windows would have k times less efficient than Linux. This is because in Windows it is possible that the IPI can be served by $k-1$ irrelevant cores due to the round-robin, while in Linux it is always served by the dedicated core.

6.2 Mac OS X

Reschedule IPI. Although Mac OS X kernel is designed to support a preemption mode, its default configuration is a non-preemption mode, and the configuration cannot be changed [20]. We developed the similar exploitation attack as shown in §5.1, but the exploitation failed and it was challenging for us to simply understand why it fails due to the limited internal information.

TLB Shootdown IPI. Similar to Linux, Mac OS X sends an IPI to the CPU core which has a TLB entry to be flushed. Since Mac OS X is UNIX-based operating systems, we confirmed that the race window extension method is shown in §5.2.1 also works on Mac OS X as well—i.e., using similar OS X system calls including `mmap()`, `mprotect()`, and `munmap()`, we were able to raise TLB shootdown IPI, thereby extending the race window (i.e., T_y).

HW Interrupt. OS X does not provide internal information on hardware interrupts, so we could not understand whether IRQ handling mechanisms in OS X involve the core affinity. We developed a similar exploitation attack, as shown in §5.3, but it does not work and we were not able to understand the reason due to lack of information.

7 Evaluation

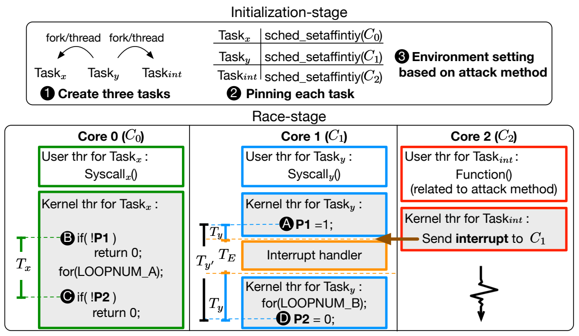
This section aims at evaluating the exploitation effectiveness of ExpRace. First, we used ExpRace to exploit 10 real-world multi-variable races in Linux (§7.1). Then to understand more detailed aspects of real-world exploitation, we developed and exploited synthetic multi-variable races in Linux (§7.2). Lastly, we launched the synthetic multi-variable races on Windows and Mac OS X, testing if ExpRace also works for OSes other than Linux (§7.3).

Experimental Setup. For the experiments on Linux, we ran Ubuntu 18.04.3 LTS on Intel i7-8700 (3.20GHZ) with 32 GB of memory, which enabled the `CONFIG_PREEMPT_VOLUNTARY` option (which is a default configuration for desktop machines). For the experiments on Microsoft Windows, we ran Windows 10 version 1909 (OS build 18363.592) on Intel i7-8700 (3.20GHZ) with 32 GB of memory. For the experiments on Mac OS X, we ran macOS 10.14 (19A583) on Mac mini (2018) on Intel i5-8500B (3.00GHz) with 8 GB of memory.

Evaluation Methods. Throughout this evaluation section, we varied the exploitation method as follows: `Baseline` refers to the brute-force attack without ExpRace. `Reschedule` refers to the brute-force attack with ExpRace's reschedule IPI method (§5.1). `membarrier` refers to the brute-force attack with ExpRace's membarrier IPI method (§5.2.2). `TLB shootdown` refers to the brute-force attack with ExpRace's TLB shootdown IPI method (§5.2.1). `HW interrupt` refers to the brute-force attack with ExpRace's hardware interrupt method (§5.3).

7.1 Exploiting Real-World Races in Linux

Real-World Exploitation Setup. In order to demonstrate that ExpRace is truly effective in exploiting race vulnerabilities, we used ExpRace to exploit 10 real-world multi-variable races in Linux listed in Table 1. We utilized publicly available exploits for CVE-2019-1999 and CVE-2019-2025. Since the rest eight do not have publicly available exploits, we developed an exploit for the rest eight. We run the vulnerable kernel version of each vulnerability to launch the exploitation: CVE-2017-15265 on v4.13.5; da1b9564 on v4.18-rc3; 4842e98f on v4.4.19; and all the other vulnerabilities on v4.19.0.



Initialization-stage
fork/thread
fork/thread
Task_x
Task_y
Task_{int}
❶ Create three tasks
Task_x | sched_setaffintiy(C_0)
Task_y | sched_setaffintiy(C_1)
Task_{int} | sched_setaffintiy(C_2)
❷ Pinning each task
❸ Environment setting based on attack method
Race-stage
Core 0 (C_0)
User thr for Task_x : $\text{Syscall}_x()$
Kernel thr for Task_x :
Ⓑ if(!P1) return 0;
for(LOOPNUM_A);
Ⓒ if(!P2) return 0;
T_x
Core 1 (C_1)
User thr for Task_y : $\text{Syscall}_y()$
Kernel thr for Task_y :
Ⓐ P1 =1;
Interrupt handler
Kernel thr for Task_y :
for(LOOPNUM_B);
Ⓓ P2 = 0;
T_y
$T_{y'}$
T_E
T_y
Core 2 (C_2)
User thr for Task_{int} : Function() (related to attack method)
Kernel thr for Task_{int} :
Send interrupt to C_1

Figure 8: A workflow of synthetic race exploitation evaluation

These 10 real-world vulnerabilities are non-inclusive multi-variable races, and its T_x and T_y are measured, as shown in Table 4. For each vulnerability, we attempted to exploit for 24 hours at maximum (i.e., simply taking an infinite loop to trigger the race) while varying an exploitation method to enlarge T_y. Note that once the exploitation is successful, we stopped the experiment due to the following two reasons: i) kernel crashes due to memory corruption; (ii) even if not crashing, the kernel is in an abnormal state that the race cannot be triggered again. The only reliable way is to reboot the system, which requires non-trivial evaluation efforts.

Real-World Exploitation Results. The overall results are shown in Table 4. Without ExpRace, the 24-hours long exploitation attempts were failed for all 10 real-world vulnerabilities as expected (shown in `Baseline` column). Using `Reschedule`, three vulnerabilities were successfully exploited within 66 seconds, while the rest seven cases were failed. These failed cases were related to the fact that the length of an enlarged race window $T_{y'}$ is smaller than $T_x - T_y$, which we further study shortly. Using membarrier IPI, three cases were successfully exploited. It failed to exploit five cases (i.e., CVE-2019-6974, CVE-2019-1999, 11eb85ec, 1a6084f8, and e20a2e9c) due to the small $T_{y'}$. With TLB shootdown, seven cases were successfully exploited. With hardware interrupts, all 10 cases were successfully exploited. Note that membarrier and TLB shootdown cannot be applied to exploit CVE-2019-6974 and da1b9564, as these require that the two racy syscalls should be invoked from the same process, which cannot be supported by these methods.

Accuracy of Probability Model. To interpret these results using the exploitation probability model (i.e., $P^{\text{ExpRace}}_{\text{multi}}$), we also collected the number of relevant events during the exploitation (shown in Table A.1). This confirms that ExpRace's exploitation is feasible within a reasonable time (i.e., at most 118 seconds).

7.2 Exploiting Synthetic Races in Linux

7.2.1 Design Synthetic Races

To perform an in-depth study on the effectiveness of ExpRace, we created a synthetic race vulnerability for Linux. This vulnerability is implemented as a device driver, which takes two syscalls from userspace, $Syscall_x$ and $Syscall_y$, where two syscalls have multi-variable race vulnerability on two global variables `P1` and `P2`. In order to check if the race exploitation was successful, we designed the vulnerability such that $Syscall_x$ returns 0x1337 if successful. Otherwise, it returns zero. We also inserted two `for` loops, one in between Ⓐ and Ⓓ and the other in between Ⓑ and Ⓒ, so that we can control T_x and T_y by modifying the number of loop iterations (i.e., `LOOPNUM_A` for T_x and `LOOPNUM_B` for T_y). Note that we cannot precisely control CPU cycles of T_x and T_y as it is indirectly impacted by executed instructions.

To successfully exploit this race, the initialization stage is similar to the attack shown in §5. The important thing is that the execution should occur in the following order: Ⓐ $\gg$ Ⓑ $\gg$ Ⓒ $\gg$ Ⓓ. When trying each race window enlarging method, we performed the necessary steps to trigger IPI or interrupts as described in §5 (i.e., creating $\texttt{Task}_{int}$ and invoking a set of syscalls from $\texttt{Task}_{int}$). Similar to the real-world exploitation case, we kept invoking $\texttt{Syscall}_x$ and $\texttt{Syscall}_y$ while varying an exploitation method. We repeated the above mentioned exploitation for one minute since one minute was enough to collect a meaningful number of data as we show next.

7.2.2 Synthetic Race Exploitation Results

We launched an exploitation as described in §7.2.1 so as to clearly interpret the exploitation result against real-world races (shown in Table 4). In order to simulate T_x and T_y of real-world cases, we picked four different T_y, and launched the exploitation for each T_y while varying T_x. For each T_x and T_y pair, we launched an exploitation for one minute and measured the following information: the number of total trials, the number of successful exploitation, and the number of issued interrupts.

The results of the synthetic exploitation are shown in Figure 9. Each subfigure is the result of fixing T_y at around 17, 41, 130, and 1135 cycles, and X-axis represents T_x and Y-axis represents the number of successful exploitation after trying one minute.

Overall, the `baseline` method only shows the success case if $T_x < T_y$ (implicating the inclusive multi-variable race), and it does not show any success case for if $T_x > T_y$ (implicating the non-inclusive multi-variable race). Moreover, TLB shootdown is the most effective when T_x is less than 1,500 cycles. This is because it takes less time to invoke TLB shootdown by $\texttt{Task}_{int}$, so it issues a large number of IPIs compared to other exploitation methods. Hardware interrupts show the stable success number over T_x, because T_E (i.e., an enlarged

Vulnerability	Success (Time taken until the first success)					Average Cycles			
	`Baseline`	`Reschedule`	`membarrier`	`TLB shootdown`	`HW interrupt`	T_x	T_y	$T_{\texttt{Syscall}_x}$	$T_{\texttt{Syscall}_y}$
CVE-2019-6974	✗ (> 24 hours)	✗ (> 24 hours)	✗ (Cannot apply)	✗ (Cannot apply)	✓ (< 30 sec)	1,210	18	3,818	7,102
CVE-2019-2025	✗ (> 24 hours)	✓ (< 34 sec)	✓ (< 10 sec)	✓ (< 10 sec)	✓ (< 25 sec)	600	50	8,131	227,538
CVE-2019-1999	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	✓ (< 60 sec)	✓ (< 70 sec)	1,800	150	52,285	623,597
CVE-2017-15265	✗ (> 24 hours)	✓ (< 66 sec)	✓ (< 60 sec)	✓ (< 60 sec)	✓ (< 80 sec)	450	35	9,893	17,893
11eb85ec... [35]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	▲ (< 30 sec)	✓ (< 70 sec)	2,515	113	54,389	18,296
1a6084f8... [53]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	▲ (< 40 sec)	✓ (< 60 sec)	2,363	158	56,275	13,499
20f2e4c2... [36]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	✓ (< 20 sec)	✓ (< 45 sec)	1,580	122	50,755	6,392
4842e98f... [32]	✗ (> 24 hours)	✓ (< 31 sec)	✓ (< 15 sec)	✓ (< 10 sec)	✓ (< 25 sec)	730	120	11,704	28,363
da1b9564... [33]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (Cannot apply)	✗ (Cannot apply)	✓ (< 118 sec)	2,250	18	349,342	176,165
e20a2e9c... [34]	✗ (> 24 hours)	✗ (> 24 hours)	✗ (> 24 hours)	▲ (< 30 sec)	✓ (< 30 sec)	13,121	1,153	109,873	19,503

Table 4: Exploitation results on real-world race vulnerabilities in Linux. ✓: the exploitation is successful within 24 hours; ▲: The extended cycle by TLB shootdown IPI are vary depends on the number of pages. The exploitation with 1 page isn't successful for given 24 hours but successful with a number of pages; ✗: the exploitation has failed for given 24 hours. The time enclosed in a parentheses denotes the time taken for the first exploitation success.

cycles) is larger than all plotted T_x values (i.e., according to our measurement shown in §7.2.3, T_E is measured to be about 14,103 cycles for hardware interrupts, respectively).

In each subfigure, annotated vertical lines indicate when T_x and T_y are similar as those of real-world vulnerability. When T_y is about 17 cycles (Figure 9-a-1), it can be explained why da1b9564 can be exploited by the hardware interrupt—the hardware interrupt maintains its success number even if T_x is more than 2,250 cycles. All other methods failed to maintain its success number before reaching 2,250 cycles of T_x. Note that membarrier and TLB shootdown also failed to maintain the success number, implying that even if it can be applied to da1b9564 (irrespective of the process/thread issue), it would have failed to exploit. In probability model Figure 9-a-2, hardware interrupt only success when T_x is about 2,250 cycles.

The rest sub-figures, from Figure 9-b-1 to d-1, also show consistent results as Figure 9-a-1. In particular, when T_y is {41, 130, 1135} cycles, {CVE-2017-15265, CVE-2019-1999, CVE-2019-2025, 11eb85ec, 1a6084f8, 20f2e4c2, 4842e98f, e20a2e9c} can be exploited by all EXPRACE's methods, respectively. However, the `baseline` method failed for all the above eight, showing consistent results as the real-world exploitation results. For CVE-2019-1999, only `TLB shootdown` and `HW interrupt` were successful, which is also consistent with the real-world exploitation results.

Figure 10 shows the number of related events — $\texttt{Syscall}_x$, $\texttt{Syscall}_y$, the number of interrupts (including both hardware interrupts and IPIs) — during the exploitation. During this one minute, the number of $\texttt{Syscall}_x$ were similar for all exploitation methods (i.e., about 250M) because the handling mechanism of $\texttt{Syscall}_x$ is not impacted by varying the exploitation method.

On the contrary, the number of $\texttt{Syscall}_y$ varies. The baseline method had the biggest number because $\texttt{Syscall}_y$ handling was not interfered by the interrupt. All other methods have less because execution of $\texttt{Syscall}_y$ stalls for a while once receiving an interrupt.

In terms of the number of interrupts, there were no interrupts while trying `baseline`. `membarrier` had the biggest number of interrupts, which seems to be related to the fact the `membarrier` IPI is the lightest compared to the others, so it can be quickly delivered. `HW interrupt` had the smallest among EXPRACE's exploitation methods, implying that its interrupt issue logic is the slowest.

Accuracy of Probability Model. Overall, both synthetic evaluation results and probability model results decrease sharply at a specific x-axis value. As shown in Figure 9-d-1, the number of successes is sharply decreased after T_x is bigger than 16,000 cycles. Similarly, as shown in Figure 9-d-2, the number of success is sharply decreased after T_x is bigger than 15,000 cycles. We note that the slight difference between these two results seems to be due to the measurement errors. While all the cycles are fixed values in the probability model, measured cycles in the synthetic experiments can have measurement errors due to the noise.

7.2.3 Length of Enlarged Race Windows

To clearly see how much a race window is enlarged, we measured cycles of $T_{y'}$ when exploiting the synthetic race. More specifically, we fixed T_x and T_y as 539 and 25 cycles, respectively. Then we instrumented `rdtsc` at two places to measure $T_{y'}$: (i) right before line 23 in Figure A.4 (i.e., Ⓐ in Figure 8) and (ii) right after line 27 Figure A.4 (i.e., Ⓓ in Figure 8).

Figure 11 shows the average of $T_{y'}$ for each exploitation method. A filled circle denotes the average cycles when the exploitation is successful, and a cross mark denotes when failed. Here, the cycle difference between success and failure for each method indicates T_E, because success means the race window is extended (i.e., $T_{y'} = T_y + T_E$). Overall, all exploitation methods show higher $T_{y'}$ when successful, and each method shows a different enlargement. The hardware interrupt is the largest (i.e., T_E = 14,103 cycles), which explains why hardware interrupts have maintained the number of success while varying T_x in Figure 9—for all T_x cycles, $T_{y'} > T_x$. The contrary example is `Reschedule`, which has

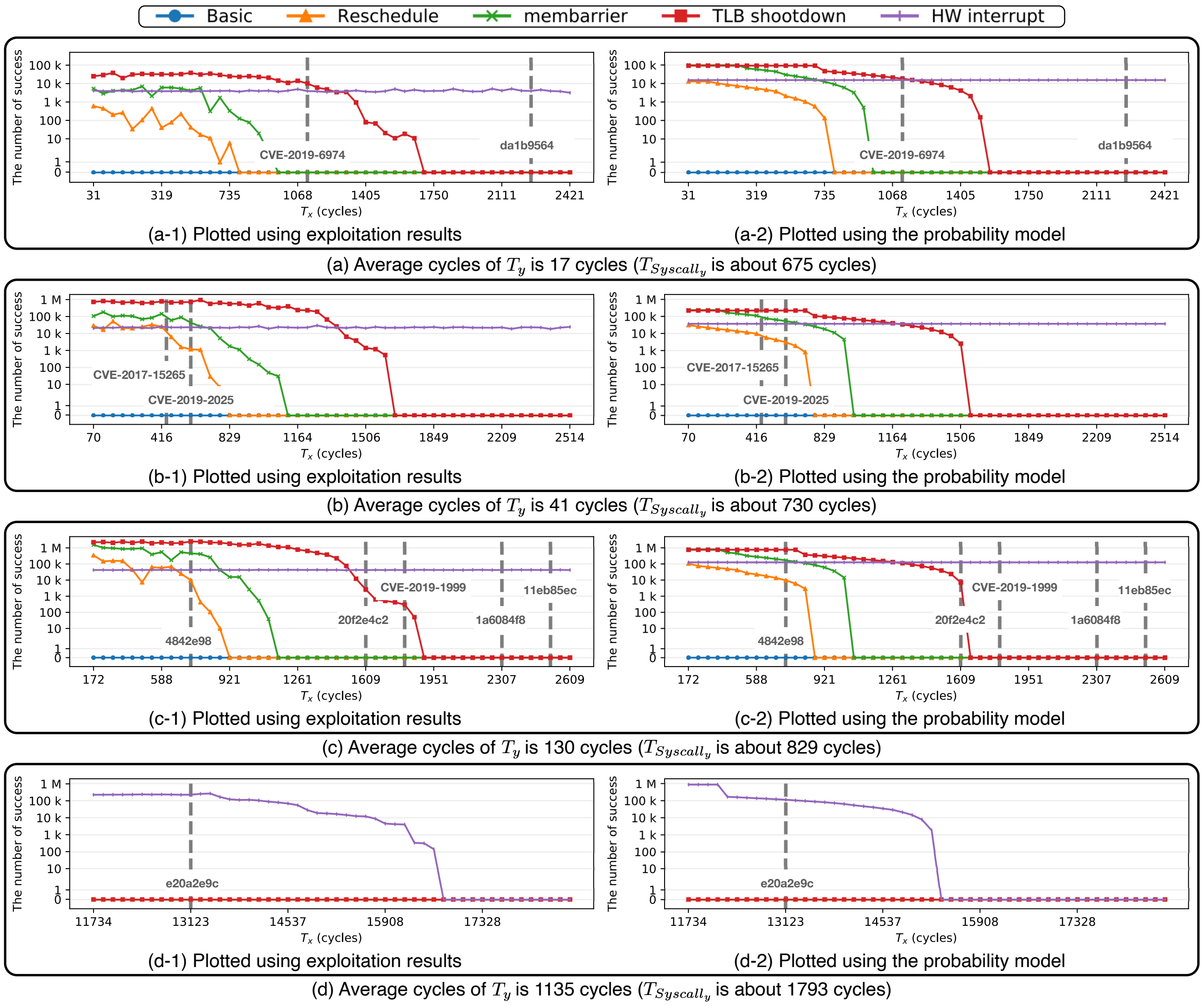


Figure 9: Exploitation results on the synthetic race vulnerability in Linux

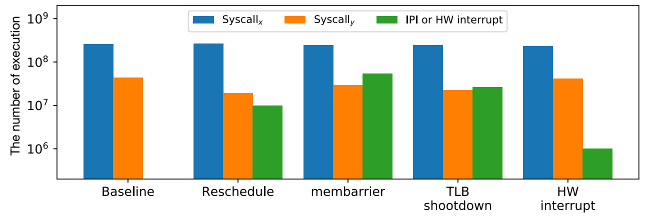


Figure 10: The number of events while exploiting the synthetic race vulnerability

shown 734 cycles of T_E. Therefore, the number of success for `Reschedule` always dropped earlier than other EXPRACE's methods in Figure 9.

7.3 Exploiting Other OSes

In order to check the effectiveness of exploiting other OSes using EXPRACE, we launched the exploitation against the synthetic race vulnerability (as described in §7.2.1) developed as a kernel driver for Windows and OS X, respectively. Figure 12 shows exploitation results, where we fixed T_y as 17 cycles and launched an exploitation for one minute. Overall, `Reschedule` and `TLB shootdown` has shown far more success numbers than `baseline`, demonstrating the exploitation effectiveness of EXPRACE. One thing to n ote is that in Linux the success number quickly drops when $T_x > 1,400$, but in Windows and OS X such a drop occurs when $T_x > 2,000$. We suspect this suggests that the TLB flushing in Linux is processed faster than Windows and OS X.

Unfortunately, we were not able to include the real-world exploitation cases for Windows and Mac OS X, because no

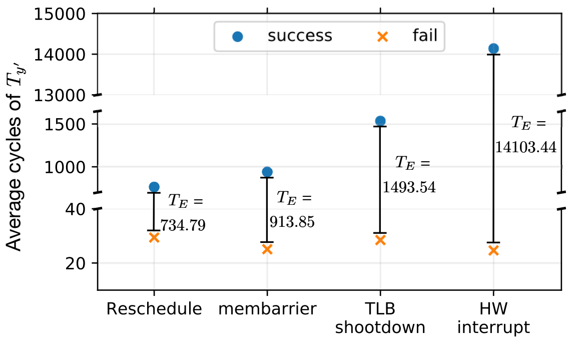
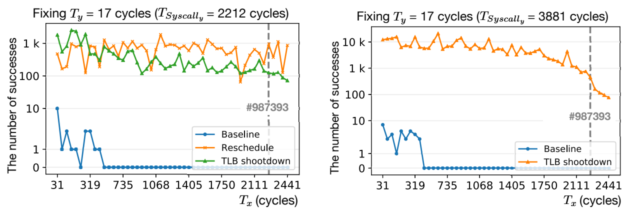


Figure 11: CPU cycles when exploiting the synthetic race vulnerability. A blue filled circle denotes averaged CPU cycles of T_y when the exploitation is successful; A orange cross denotes averaged CPU cycles of T_y when failed. The cycle difference between success and fail approximately shows an enlarged race window (i.e., T_E).



(a) Microsoft Windows (b) Mac OS X

Figure 12: Exploitation results against the synthetic race vulnerability on Windows and OS X

public descriptions on non-inclusive race vulnerabilities on these platforms were available. We further discuss this limitation of ExpRace on other OSes in §8.1.

8 Discussion

In this section, we first discuss the possibility of exploiting other OSes using ExpRace (§8.1). Then we discuss how the new threats introduced by ExpRace can be mitigated (§8.2).

8.1 Possibility of Exploiting Other OSes

Our evaluation in §7.3 suggests that it is possible to launch exploitation with ExpRace on synthetic race issues. However, as we were not able to perform the real-world case study, it is very premature to confirm the exploitation possibility using ExpRace on other OSes. Thus it requires more studies in the future. Nevertheless, because there are no significant differences between Linux and Windows/Mac OS X from the perspective of race issues, we believe the attacking method introduced by ExpRace may still be valid for these platforms as well.

8.2 Mitigation

In order to mitigate new exploitation threats introduced by ExpRace, there can be two potential mitigation approaches: 1) identifying an abnormal frequency of interrupts and 2) avoiding preemption by userspace-originated interrupts.

The first approach is based on the fact that ExpRace imposes very frequent interrupts. Thus, exploitation attempts with ExpRace can be captured with the kernel-level monitor, which checks if too many interrupts were raised within a short period of time. While this approach would be simple to implement and deploy, it may have false positives (i.e., some benign behaviors may entail frequent interrupts), which needs further investigation.

The second approach is to avoid any preemption of kernel execution context if an interrupt is indirectly raised by a user. With this avoidance, the user will not be able to enforce the preemption within the race window, thereby mitigating ExpRace's exploitation technique. While this approach would have more precise detection capability than the first one, it requires heavy kernel modification to keep track of the origins of all interrupt events, which may hinder its practical adoption as well as imposing runtime tracking overheads.

To summarize, we expect adopting these two approaches needs careful investigation so as not to break the original semantics and backward compatibility of existing kernel services. We further hope this paper provokes interesting discussions on the fundamental design of interrupt handling in operating systems, particularly from security perspectives—asking if the interrupt timing controls by unprivileged users should be allowed or not.

9 Related work

Detecting Races. For the sake of detecting race condition, many works attempted to use either a static analysis approach [5, 15, 17, 39, 58–60, 63] or a dynamic analysis approach [7, 9, 28, 30, 31, 40, 47, 49, 50], or both [27]. Most race detectors using static analysis are based on lockset analysis [5, 15, 17, 58, 59]. WHOOP [15] uses symbolic pairwise lockset analysis for detecting race condition in the Linux kernel driver. Deadline [63] uses static analysis to find multi-reads in the kernel and employs symbolic checking to check each multi-read satisfies the constraints to be a double-fetch bug. Memory sampling techniques [7, 9, 18, 28, 40] selectively monitor memory accesses to detect race conditions. SNORLAX [30] utilizes a coarse interleaving hypothesis, which relies on a dynamic-static interprocedural pointer and type analysis, to diagnose the root causes of concurrency bugs. Razzer [27] first extracts a set of race candidates through the static analysis and then starts fuzzing while setting up the breakpoints (fuzzing) to discover races in an efficient way. Compared to ExpRace, these studies were focusing on automating the race condition detection, while ExpRace focuses on how to exploit the real-world race condition issues.

Avoiding Races. Previous works employ deterministic execution to avoid concurrency bugs [4, 6, 12, 13, 37]. Grace [6] turns the multi-threaded program into a sequential program using fork-join parallelism. DThread [37] keeps track of memory modification sites using virtual memory protection and ensures deterministic update orders by each thread. PEREGRINE [12] proposed a hybrid scheduling mechanism, which uses mem-schedule for the racy part and sync-schedule for the non-racy part, thereby guaranteeing a deterministic multi-threading system. Parrot [13] orders thread synchronization in the well-defined round-robin order.

Automating Exploitation of Memory Corruptions. APEG [8] identifies missing sanitization checks by compare patched and unpatched binary using binary analysis and generate an input to trigger the difference. AEG [54] and mayhem [10] use symbolic execution (or hybrid symbolic execution) to generate shell spawning exploit. FUZE [62] and Revery [61] identify and further analyze the root cause of vulnerabilities, and they automatically generate an exploit. KOOBE [11] evaluates the exploitability of kernel's out-of-bound write vulnerabilities using capability-guided fuzzing for automated exploit generation.

Performance Degradation. Many researchers leveraged performance degradation factors (e.g., interrupt [22, 46, 55, 56] or cache eviction [1, 21]) to launch or assist side-channel attacks. While these and EXPRACE both degrade the performance to launch attacks, the difference is that EXPRACE focuses on attacking race issues where previous works cannot be applied.

Nemesis [56] used interrupts to leak instruction timings against Intel SGX. SGX-Step [55] used APIC timer interrupts to track page table entries directly from user space. Hahnel *et al*. [22] use timer interrupt to achieve higher temporal and spatial resolution. Cachezoom [46] consecutively sends interrupts to amplify the cache side channel. However, they assume the attacker already had full control over the kernel. This allows the attacker to generate interrupts as desired. On the contrary, EXPRACE assumes that the attacker only has user-level privileges, so the interrupt generation cannot directly be performed.

Thomas *et al*. [1] amplified the result of side-channel attacks using cache eviction. Compared to EXPRACE, this method does not slow down a specific target core but slows down entire cores. Thus, this attack would increase both T_x and T_y, which cannot be applied for EXPRACE.

10 Conclusion

This paper studies the exploitability of kernel data races. We analyzed real-world kernel races and found an intrinsic condition separating easy-to-exploit and hard-to-exploit races. Then we developed EXPRACE, a generic race exploitation technique for Linux, Windows, OS X. Through evaluating with real-world race vulnerabilities, EXPRACE demonstrated that it truly augments the exploitability of kernel races.

11 Acknowledgment

We sincerely thank anonymous reviewers and our shepherd, Vasileios P. Kemerlis, for their insightful comments, which significantly improved the final version of this paper. The authors would like to thank Dae R. Jeong who provided the initial idea and insights of this work. This work was supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone), and US Office of Naval Research under grants N00014-18-1-2022. The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work. The Institute of Engineering Research at Seoul National University provided research facilities for this work.

References

[1] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom. Amplifying side channels through performance degradation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2016.

[2] N. Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[3] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. Sharc: Checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, Arizona, June 2008.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012.

[5] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, RENTON, WA, July 2019.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *Proceedings of the 24th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando Florida, Oct. 2009.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010.

[8] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[9] Y. Cai, J. Zhang, L. Cao, and J. Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 24th ACM SIGSOFT*

Symposium on the Foundations of Software Engineering (FSE), Seattle, WA, Nov. 2016.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[11] W. Chen, X. Zou, G. Li, and Z. Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security)*, BOSTON, MA, Aug. 2020.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[13] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[14] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with javascript. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies (WOOT)*, SAN JOSE, CA, July 2008.

[15] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *Proceedings of the 30rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, Nebraska, Sept. 2015.

[16] F. Electrons. Realtime in embedded linux systems. 2004.

[17] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[18] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[19] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[20] L. G. Gerbarg. Advanced synchronization in mac os x: Extending unix to smp and real-time. In *BSDCon*, pages 37–45, 2002.

[21] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[22] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[23] J. Horn. Android: binder use-after-free of vma via race between reclaim and munmap, 2018. https://bugs.chromium.org/p/project-zero/issues/detail?id=1720.

[24] J. Horn. Android: binder use-after-free via racy initialization of ->allow_user_free, 2018. https://bugs.chromium.org/p/project-zero/issues/detail?id=1721&q=cve-2019-1999.

[25] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[26] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[27] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2019.

[28] Y. Jiang, Y. Yang, T. Xiao, T. Sheng, and W. Chen. Drddr: a lightweight method to detect data races in linux kernel. *The Journal of Supercomputing*, 72(4):1645–1659, 2016.

[29] G. Jin, W. Zhang, and D. Deng. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[30] B. Kasikci, W. Cui, X. Ge, and B. Niu. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[31] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[32] Linux. Linux commit log 4842e98f26dd80be3623c4714a244ba52ea096a8., 2017. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4842e98f26dd80be3623c4714a244ba52ea096a8.

[33] Linux. Linux commit log da1b9564e85b1d7baf66cbfabcab27e183a1db63., 2018. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da1b9564e85b1d7baf66cbfabcab27e183a1db63.

[34] Linux. Linux commit log e20a2e9c42c9e4002d9e338d74e7819e88d77162., 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e20a2e9c42c9e4002d9e338d74e7819e88d77162.

[35] Linux. Linux commit log 11eb85ec42dc8c7a7ec519b90ccf2eeae9409de8., 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=11eb85ec42dc8c7a7ec519b90ccf2eeae9409de8.

[36] Linux. Linux commit log 20f2e4c228c712158113583947f4e16691e951f6., 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=20f2e4c228c712158113583947f4e16691e951f6.

[37] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[38] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[39] Y. Luo, P. Wang, X. Zhou, and K. Lu. Dftinker: Detecting and fixing double-fetch bugs in an automated way. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 780–785. Springer, 2018.

[40] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[41] Microsoft. Interrupt affinity, 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/interrupt-affinity-and-priority.

[42] Microsoft. Scheduling priorities., 2018. https://docs.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities.

[43] MITRE. CVE-2016-8655., 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655.

[44] MITRE. CVE-2017-2636., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2636.

[45] MITRE. CVE-2017-7533., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533.

[46] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Proceedings of the 2017 Cryptographic Hardware and Embedded Systems (CHES)*, Taipei, Taiwan, Sept. 2017.

[47] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Imple-*

mentation (PLDI), San Diego, CA, June 2007.

[48] M. Prandini and M. Ramilli. Return-oriented programming. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[49] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Incheon, Korea, May–June 2018.

[50] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[51] A. Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007, 2007.

[52] M. Stone. Bad binder: Android in-the-wild exploit, 2019. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[53] Syzkaller. Syzkaller log 1a6084f827bc586c4361b6256040c593f4c19f5b., 2020. https://syzkaller.appspot.com/bug?id=1a6084f827bc586c4361b6256040c593f4c19f5b.

[54] H. A. Thanassis, C. S. Kil, and B. David. Aeg: Automatic exploit generation. In *ser. Network and Distributed System Security Symposium*, 2011.

[55] J. Van Bulck, F. Piessens, and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[56] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[57] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[58] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: the goblint approach. In *Proceedings of the 31rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, Singapore, Sept. 2016.

[59] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007.

[60] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[61] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[62] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*, BALTIMORE, MD, Aug. 2018.

[63] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2018.

[64] Z. Xu, G. Liu, T. Wang, and H. Xu. Exploitations of uninitialized uses on macos sierra. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, VANCOUVER, BC, Aug. 2017.

[65] T. Zhang, D. Lee, and C. Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[66] T. Zhang, C. Jung, and D. Lee. Prorace: Practical data race detection for production use. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

A Appendix

```
void pin_this_task_to(int cpu) {
  cpu_set_t cset;
  CPU_ZERO(&cset);
  CPU_SET(cpu, &cset);

  // if pid is NULL then calling thread is used
  if(sched_setaffinity(0, sizeof(cpu_set_t), &cset))
    err(1, "affinity");
}

void target_thread(void *arg) {
  // Suppose that a victim thread is running on core 2.
  pin_this_task_to(2);
  while(1) {
    // There is a data race in this thread.
  }
}

int main() {
  pthread_t thr;
  pthread_create(&thr, NULL, target_thread, NULL);
  // Send rescheduling IPI to core 2 to extend the race window.
  pin_this_task_to(2);
}
```

Figure A.1: The simplified code of EXPRACE's Reschedule IPI exploitation method

```
int map_size = 0x1000;

void sendIPI() {
  char buf[8];
  void *addr;

  // Allocate memory for tlb shootdown
  addr = mmap(0, map_size, (PROT_READ | PROT_WRITE),
              MAP_SHARED | MAP_ANON, -1, 0);
  // Access memory to update tlb
  memcpy(buf, addr, 1);
  // Modify memory permission for TLB shootdown
  mprotect(addr, map_size, PROT_READ);
}

void target_thread(void *arg) {
  while(1){
    // There is a data race in this thread
  }
}

int main(void) {
  pthread_t thread;
  pthread_create(&thread, NULL, (void *)target_thread, NULL);
  sendIPI();
}
```

Figure A.2: The simplified code of EXPRACE's TLB shootdown IPI exploitation method

Vulnerability	Baseline				Reschedule				membarrier				TLB shootdown				HW interrupt			
	P_{multi}	Syscall$_x$	Syscall$_y$	interrupt	$P_{\text{multi}}^{\text{ExpRace}}$	Syscall$_x$	Syscall$_y$	interrupt	$P_{\text{multi}}^{\text{ExpRace}}$	Syscall$_x$	Syscall$_y$	interrupt	$P_{\text{multi}}^{\text{ExpRace}}$	Syscall$_x$	Syscall$_y$	interrupt	$P_{\text{multi}}^{\text{ExpRace}}$	Syscall$_x$	Syscall$_y$	interrupt
CVE-2019-6974	0	198 B	81 B	0	0	190 B	72 B	39 B	✗	✗	✗	✗	✗	✗	✗	✗	3.92e-04	30 K	4 K	14 K
CVE-2019-2025	0	150 B	5 B	0	4.42e-05	59 M	1 M	15 M	4.4e-04	30 M	39 K	9 M	1.38e-03	20 M	30 K	5 M	1.09e-03	43 M	1.3 M	250 K
CVE-2019-1999	0	19 M	19 M	0	0	12 M	12 M	37 B	0	14 M	14 M	78 B	2.95e-05	50 K	50 K	26 M	7.79e-04	140 K	140 K	1.1 M
CVE-2017-15265	0	6 B	130 M	0	4.43e-05	6 M	50 K	29 M	3.47e-04	5 M	60 K	54 M	9.13e-04	5 M	100 K	26 M	7.63e-04	6 M	113 K	1.3 M
11eb85ec...	0	2.7 M	2.7 M	0	0	1.9 M	1.9 M	12 M	0	2.2 M	2.2 M	14 M	0	2 K	2 K	46 K	5.30e-04	5 K	5 K	1 M
1a6084f8...	0	4.1 M	4.1 M	0	0	2.8 M	2.8 M	13 M	0	3.5 M	3.5 M	14 M	0	3 K	3 K	53 K	1.02e-03	5 K	5 K	1 M
20f2e4c2...	0	33 B	66 B	0	0	32 B	43 B	36 B	0	32 B	43 B	80 B	2.01e-05	7 M	15 M	10 M	6.63e-04	9 M	17 M	56 K
4842e98f...	0	7 B	51 M	0	4.99e-05	3 M	10 K	13 M	6.12e-04	1.1 M	7 K	14 M	2.16e-03	810 K	8 K	5 M	2.61e-03	2 M	13 K	417 K
da1b9564...	0	5 M	5 M	0	0	3.3 M	3.3 M	37 B	✗	✗	✗	✗	✗	✗	✗	✗	1.33e-05	9 K	9 K	2 M
e20a2e9c...	0	50 M	50 M	0	0	35 M	35 M	36 B	0	43 M	43 M	80 B	0	1 K	1 K	71 B	4.88-e4	1 K	1 K	44 K

Table A.1: Detailed exploitation results on real-world race vulnerabilities in Linux. ✗ denotes that the exploitation was not performed as the corresponding exploitation method does not work for the subjected race vulnerability.

```c
void sendIPI(void) {
  membarrier(
    MEMBARRIER_CMD_PRIVATE_EXPEDITED, 0);
}

void registerIPI(void) {
  membarrier(
    MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED, 0);
}

void target_thread(void *arg) {
  while(1){
    // There is a data race in this thread
  }
}

int main() {
  pthread_t thread;
  registerIPI();
  pthread_create(&thread, NULL, target_thread, NULL);
  sendIPI();
}
```

Figure A.3: The simplified code of ExpRace's `membarrier` IPI exploitation method

```c
int P1, P2;

// __attribute__((optimize("O0")))
long Syscallx(ulong LOOPNUM_A) {
    if(!P1)
        // Failed to exploit.
        return 0;

    for(int i = 0; i < LOOPNUM_A; i++);

    if(!P2)
        // Failed to exploit.
        return 0;

    // Race exploitation is successful.
    return 0x1337;
}

// __attribute__((optimize("O0")))
long Syscally(ulong LOOPNUM_B) {
    P2 = 1;
    // rdtsc(); // to measure Ty
    P1 = 1;

    for(int i = 0; i < LOOPNUM_B; i++);

    P2 = 0;
    // rdtsc(); // to measure Ty
    P1 = 0;

    return 0;
}
```

Figure A.4: Synthetic race vulnerability code

```c
int map_size = 0x1000;

void pin_task_to(int pid, int cpu) {
  cpu_set_t cset;
  CPU_ZERO(&cset);
  CPU_SET(cpu, &cset);

  // if pid is NULL then calling thread is used
  if(sched_setaffinity(pid, sizeof(cpu_set_t), &cset))
    err(1, "affinity");
}

void sendIRQ() {
  int sk;
  struct sockaddr_in addr;

  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = inet_addr(IP);
  addr.sin_port = htons(PORT);

  // Create socket
  sk = socket(AF_INET, SOCK_STREAM, 0);

  // Connect to server
  // HW interrupt will occurs when reply packet arrive
  connect(sock, (struct sockaddr *)&server_addr, \
          sizeof(struct sockaddr_in));
}

void target_thread(void *arg) {
  // pin process to IRQ's affinity
  pin_task_to(0, 11);
  while(1){
    // There is a data race in this thread
  }
}

int main(void) {
  pthread_t thread;
  pthread_create(&thread, NULL, (void *)target_thread,
                 NULL);
  sendIPI();
}
```

Figure A.5: The simplified code of ExpRace's `HW interrupt` exploitation method