EECS750 Term Project - Implementing and Evaluating Speculative Race Conditions from GhostRace

Landen Doty, Cole Strickler

I. Introduction

GHOSTRACE investigated how synchronization primitives in Linux behave during speculative execution and explored possible security implications as such [2]. In doing so, they discovered that all synchronization primitives can be microarchitecturally bypassed and turn all architecturally race free critical regions into Speculative Race Conditions (SRCs) [2]. For a proof-of-concept, they developed an end-to-end information leakage attack using the Speculative Use after Free (SCUAF) technique [2]. Their results show that SRCs can be used to leak, unboundedly, arbitrary memory from user or kernel space [2].

This novel technique requires several components demonstrated in [2], including the discovery of a SCUAF gadget, extending the UAF window, and turning the architecturally race-free section into an SRC. Additionally, a traditional cache side-channel attack must be used to transfer the data to the architectural state [2].

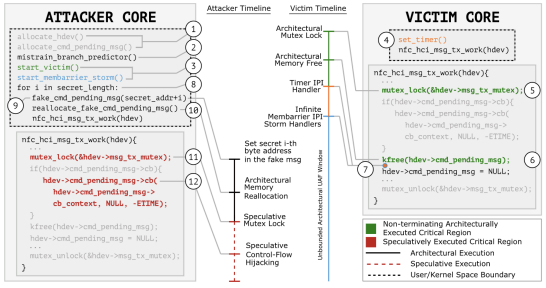

Fig. 1: Overview of GhostRace Attack

For our end of semester project, we proposed a reimplementation of *GhostRace* and an evaluation of its reproducibility, effectiveness, and impact. Our objectives were to (a) reimplement the GhostRace vulnerability on the x86 architecture using (b) a *new* gadget not used in Regab et al., and, time permitting, (c) explore its feasibility on the ARM architecture. Essentially, we were interested in recreating the attack diagramed in Figure 1. In this report, we present our findings from this work, highlighting the various challenges experienced in mounting a GhostRace attack. We first provide a background of *GhostRace* in II and outline our approach and anticipated challenges in III. We then present our findings in IV and provide discussion on the feasibility and impact of *GhostRace* in V.

II. Background

Regab et al. presented the first security analysis on the behavior synchronization primitives under speculative execution [2]. *Speculative execution* is a hardware-level performance optimization technique that, based on previous execution traces, predicts branches to be taken ahead of time. This technique reduces the impact of latency introduced by control dependencies in a branching sequence by allowing the processor to execute instructions within a predicted branch ahead of time. In the case of an incorrectly-predicted branch, instruction results are cleared from the execution trace and are not committed to the architectural state of the program. However, as is well-known, speculative execution leaves evidence of its execution in the *microarchitectural* state and can be used to perform side-channel-assisted information leakages known as *Spectre* attacks [4].

Since their discovery, Spectre attacks have been in the forefront of academic and industry research, discovering and remediating their effects in various settings. Prior work has extensively investigated Spectre attacks on varying architectures and applications, but have focused primarily on memory-safety bug classes like out-of-bounds reading and use-after-free (UAF) [2]. Thus, Regab et al. focuses specifically on exploiting Spectre-like attacks in concurrent code. Their work

```
1 void mutex_lock(struct mutex *lock){
2   ...
3   if (!__mutex_trylock_fast(lock))
4     if (atomic_long_try_cmpxchg_acquire(&lock, ...))
5       return true;
6   ...
7 }

  Call Stack:
atomic_long_try_cmpxchg_acquire(&lock, ...)
 └►arch_atomic_long_try_cmpxchg_acquire(&lock, , ...)
   └►arch_atomic_try_cmpxchg_acquire(&lock, , ...)
     └►arch_atomic_try_cmpxchg(&lock, , ...)
       └►arch_try_cmpxchg((&lock, , ...)
         └►__raw_try_cmpxchg(ptr, ...)({
             asm volatile(
                  "lock cmpxchgq %2, %1"
                  : "=a" (ret), "+m" (*ptr)
                  : "r" (new), "0" (old)
                  : "memory"
                  );
           })
```

Fig. 2: The implementation of *mutex_lock* uses a conditional branch before acquiring the lock and an un-serialized *cmpxchgq*. Credit [2].

contributes a number of findings, discovering that *all* Linux

synchronization primitives are implemented using a conditional branch and, thus, are subject to speculative execution. Figure 2 demonstrates an example of this in *mutex_lock* that, using a conditional branch, checks if the lock has been acquired before allowing execution of the critical region. Under speculative execution, the highlighted branch may be taken, allowing a number of instructions in the critical region to be executed regardless of the result of the *cmpxchgq* instruction.

With no modification or malicious interference, [2] determined that speculative execution of a critical region is significant across microarchitectures and execution configurations. Seen in Figure 3, the execution of speculative loads is consistent in Intel and AMD microarchitectures and is extended when the victim and attacker threads are located on different cores.

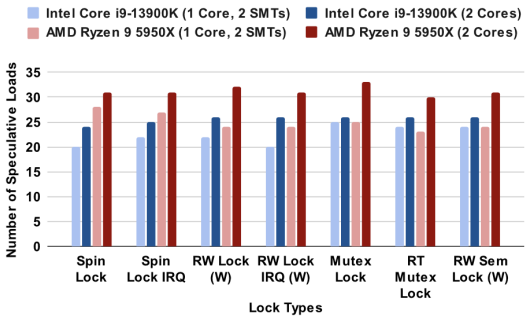

Fig. 3: Speculative window for different synchronization primitives, measured by the number of speculative loads. Credit [2]

A. SRCs and SCUAF

Regab et al. [2] introduced a new class of software bugs focusing on data races occurring in *architecturally* race-free critical regions. *Speculative Race Conditions* (SRCs) occur when two threads execute while accessing the same memory region, with one performing an architectural write and the other, a speculative access. SRCs can, in theory, be used in isolation for minimal information leakage and otherwise affect correctness of the speculatively executed section. [2] specifically investigated *Speculative Concurrent Use after Free* (SCUAF) bugs, a SRC subclass closely resembling architectural use-after-free race conditions. Regab et al. note the critical impact of SCUAFs due to the possibility of control-flow hijacking, making this class a powerful exploitation primitive.

GhostRace SCUAF attacks aim to disclose arbitrary memory using a local unprivileged user account with the ability to issue system calls. Mounting an end-to-end GhostRace attacks depends on the existence of a SCUAF *gadget*, as well as a sequence of carefully-crafted exploitation techniques. SCUAF gadgets (discussed further in III) must include an architectural free and use of a shared structure containing a function pointer field within a guarded critical region.

Visualized in Figure 4, SCUAF exploitation introduces multiple complexities that must each be individually addressed. To mount an arbitrary memory disclosure, an attacker requires the

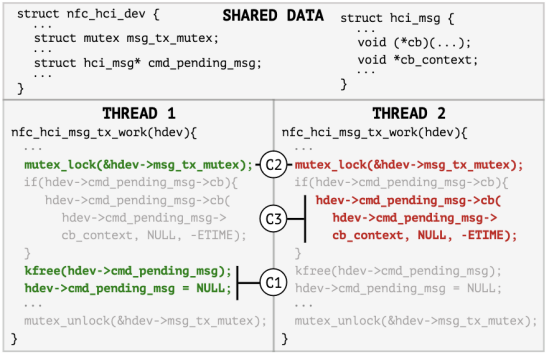

Fig. 4: Example SCUAF Gadget (/net/nfc/hci/core.c:78) Credit [2]

ability to preempt the architecturally executing thread after it has been free'd but *before* reassignment of the shared structure and/or function pointer (C1). Ideally, this window should be unbounded by the attacker, allowing the dangling function pointer in the speculatively executed thread to be redirected to a known Spectre gadget repeatedly. In addition, the SCUAF gadget should contain, as a parameter, a structure that is used in the synchronization primitive (C2) as well as a function pointer within the same structure that takes arguments (C3). Adjacently, an attacker-controlled structure must be allocated in the same memory slot as the free'd structure to successfully invoke the Spectre gadget. This requires careful slab cache collision and type confusion techniques [5] and introduces some variation into the stability of the attack.

III. Approach

Towards our goal of evaluating the reproducibility of *GhostRace*, we fully reimplement the *Inter Processor Interrupt Storm* technique used in [2] to unbound the UAF window (C1). Additionally, we utilize the provided Coccinelle scripts to identify SCUAF gadgets in version 6.8.2 of the Linux Kernel and, ideally, identify a subset that are exploitable without modification to the kernel source [1], [2]. Finally, we combine these results to produce two variations of the GhostRace attack.

A. Extending UAF Window

GhostRace utilizes the combination of two techniques to effectively unbound the UAF exploitation window. The first, based on file descriptor-bound timers, provides a reasonable delay upon expiration of a timer. The attacker first registers a series of `epoll` instances, which are kernel-level data structures that provide file descriptor monitoring services to user space processes. Each epoll instance maintains a list of *interest* and *ready* file descriptors that are monitored and notified for service when ready for I/O, respectively [6]. Then, using the `timerfd` library, the attacker registers a high-precision timer and binds it a file descriptor [6], [9]. By duplicating the timer's file descriptor (potentially thousands of times), the attacker can create long interest lists across multiple epoll instances and, upon the timer's expiration, cause

the interrupt handler to iterate for hundreds of microseconds [9]. As mentioned, the timer must be able to precisely interrupt between the end of kfree and the beginning of reassignment. In kernel mode, kfree disables hardware IRQ until the last 8 instructions of the function. Upon expiration, the timer interrupt will not be processed until IRQ is reenabled. These features help facilitate the surgical interruption needed for the attack.

The second technique focuses on interrupting the timerfd/epoll interrupt handler and causes the victim core to be infinitely preempted. Prior to Linux Kernel version 6.8.9, the `membarrier` system call provided a low-latency mechanism to preempt any adjacent core. Membarrier exposes a command, `MEMBARRIER_CMD_PRIVATE_EXPEDITED_RSEQ`, that, in benign cases, is used as a low-level synchronization mechanism to restart sequences of `rseq` instructions previously preempted during migration, I/O, or other interrupts. It requires that a process register child threads it intends to restart, but does not prevent it from repeatedly issuing the command and thus, infinitely preempting adjacent cores.

We implemented these techniques in isolation and evaluated their effectiveness in precisely and unboundedly preempting a target core's execution.

B. SCUAF Gadget Search

The feasibility of GhostRace attacks depends wholly on the ability to identify and reach SCUAF gadgets in the target codebase. [2] provided a set of Coccinelle scripts that were used to identify the gadgets used in their investigation on version 5.15.83 of the Linux Kernel. We repurpose these scripts to scan over the most recent version available at the onset of our project, 6.8.2. Coccinelle is a common tool amongst Linux kernel developers and is used to apply patches and detect particular patterns in large codebases using structural pattern matching.

Coccinelle does not provide a mechanism for dataflow analysis and thus, cannot provide information on data dependencies between gadgets. The provided scripts from Regab et al. also do not require that the parameters to the `TARGET_FUNC` be the same structures operated on in the critical section of the function. These limitations lead to inherit imprecision and over approximation of true SCUAF gadgets present in the Linux kernel. [2] also notes a large concentration of discovered gadgets in device drivers and other hardware, device-specific, service code that is generally unreachable on arbitrary Linux installations. To address these limitations, we scan the entirety of the version 6.8.2 Linux kernel, but provide statistics on infeasible gadgets.

We identify three subsets of *feasible* gadgets - Non-Device Driver (ND), Colocated (C), Free and Use on the Same Struct (FU). Filtering of ND is mostly straightforward; we filter out those in the `/drivers` top-level source directory as well as those servicing device-specific functionality in `sound`, `bluetooth`, `and networking`. The latter required some manual analysis. Filtering of C considers only gadgets where the *free* and *use* are performed in the same

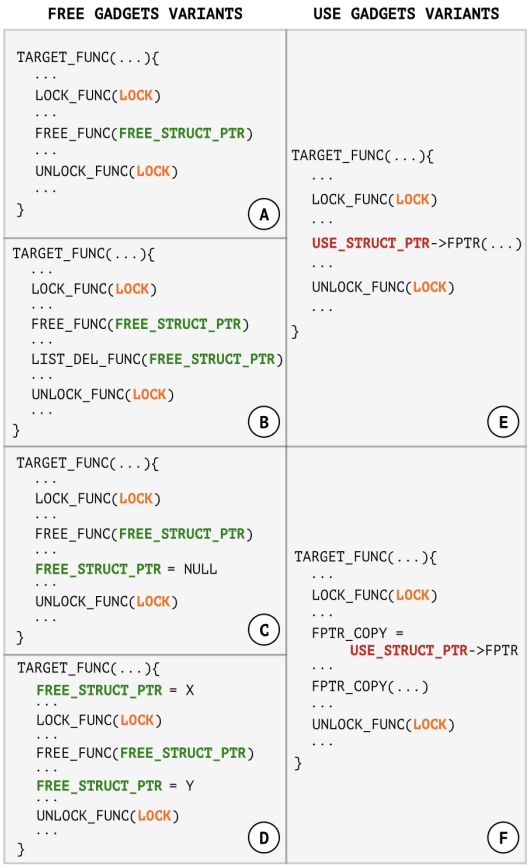

Fig. 5: Coccinelle scripts used to scan the Linux kernel source for SCUAF gadgets. Credit [2]

function. While this does provide an under approximation of useful gadgets, these gadgets guarantee that the attacker can reliably trigger both exploitation conditions in a single function invocation and that the target structure is not otherwise accessed in another execution context. FU filtering provided an obvious filtering of gadgets where free and use are performed on the same structure. Some initial manual analyses of the unfiltered scans showed a significant number of SCUAF gadgets where a SCUAF-like free *or* use could occur but not the counterpart.

We provide the results of these scans in IV, showing that a significantly minimal amount of SCUAF gadgets are truly GhostRace-exploitable.

C. Attack Variants

Reviewing the results from our SCUAF gadget filters discussed previously, we found it necessary to consider other GhostRace attack variants in order to implement an end-to-end information disclosure. As such, we provided *two* derivations - one in user space as a standalone executable and another as a loadable kernel module.

IV. Results

We implemented each component discussed in II and III and provide our results and evaluations below. The source for

our project can be found at https://github.com/ColeStrickler/EECS750-FinalProject/.

A. IPI Storm

The `timerfd/epoll` and `membarrier` IPI technique was evaluated as an isolated mechanism. We configured two threads - one set to spin in a `while` loop (T1) and another to terminate the first, immediately after it's created (T2). We measured the average amount of time T1 executed before it is terminated by T2. T1 and the main process (M) are pinned to `CPU 0` and T2 is pinned to `CPU 1`. The remainder of the logical CPU cores are tenanted by the IPI storming threads. This simulates a victim thread (T2) attempting to perform legitimate, time-sensitive work while an attacker thread (M) prevents it from doing so using the IPI storm technique.

We performed our experiments on a 12th Generation Intel i5-12600K with 10 cores and 16 Simultaneous Multithreads (SMTs). We vary the number of SMTs used by the IPI mechanism, leaving the remaining to unrelated tasks and average our results over 10 experiments per SMT. We find that only 10 SMTs are required to unbound T1's execution time and as few as 8 SMTs to extend T1's execution to at least one minute. This confirms the findings from [2] that, on an Intel i9-12900K, required 15 SMTs to unbound the victim's execution.

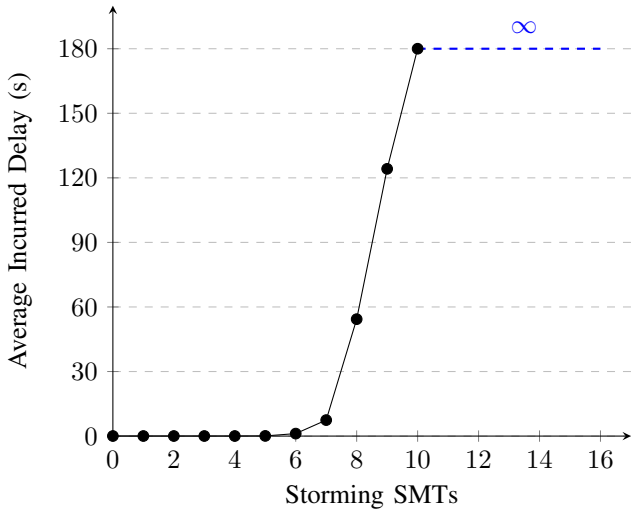

Fig. 6: Extended Exploitation Window via IPI Storm

It should be noted that this technique was assigned CVE-2024-26602 and has since been patched by the Linux Kernel [3], [8].

B. Gadget Availability

Using the formatted results from the Coccinelle scripts, we performed filtering according to the three subsets outlined in III - ND, C, and FU. We assume that each subset represents a distinct feature of SCUAF gadgets, but all three are necessary for a gadget to be considered useful. We provided statistics on the individual frequency of each subset as well as the union of C + FU and C + FU + ND. The first represents gadgets who *may* be useful, should an attacker have access to the necessary device driver; however, we considered these out of scope for our project. The latter represents the single gadget that meets all filtering criteria and required further manual analysis.

TABLE I: Useful Gadget Frequency

Filter	No. of Gadgets	Percent of Total
Total	14754	100.0%
ND	3498	23.7%
FU	1693	11.5%
C	154	1.1%
C + FU	23	0.16%
C + FU + ND	1	0.07%

ND: Non-Device Driver
C: Colocated
FU: Free and Use on the Same Struct

The single gadget is contained in `/block/blk-cgroup.c:blkcg_policy_register` and operates on `struct pol`. This function handles policy registration related to the `blkio` group, so we had hoped to find a system call or programmable interface via `cgroup` that would expose it to user space. Despite our efforts and manual analysis, this gadget is only referenced six times throughout the entirety of the kernel source and is not reachable from user space. It may be invoked as a side effect of user space `cgroup` configuration, but is not programmatically reachable in way that an attacker could invoke it for information disclosure.

Due to this result, we implemented a derivation of the GhostRace attack, demonstrating it's feasibility in an isolated user mode process and as a loadable kernel module.

C. Attack Derivations

Both attacks follow the same general procedure as the original GhostRace attack. We train the branch predictor according the synchronization type used in the gadget, initiate the IPI storm, and set a high precision timer to expire after the vulnerable struct is architecturally free'd. In kernel mode, we were able to perform the surgical interruption facilitated by the IRQ enable/disable sequence, but in user mode this faculty was not available and therefore we padded instructions after the call to free to facilitate timing. Then, we incrementally leak the secret by reallocating a structure containing a function pointer to a Spectre-v1 gadget.

We measure the effectiveness of our implementation using three metrics - Leak Rate, Race Condition Accuracy, and Inference Accuracy. *Leak Rate* corresponds to the rate in which raw data (correctness aside) is transmitted. *Race Condition Accuracy* is the rate of which our attack successfully creates an unbounded exploitation window. *Inference Accuracy* is the rate of which our attack successfully discloses the correct data. In both cases, we did not implement slab cache collision to fully control reallocation into the victim thread's free'd struct location. This technique proved more difficult to reimplement than expected and is left to future work beyond the duration of this course.Instead, we chose to simulate the effects as if we had achieved a successful slab collision.

User Mode Process. In our user mode implementation, we achieved a leak rate of 0.29b per second, race condition accuracy of 100%, and inference accuracy of 56.7%.

Kernel Module. In our kernel module implementation, we achieved a leak rate of 0.43b per second, race condition accuracy of 83.5%, and inference accuracy of 99.7%.
Due to the slow leak rate, we made a design decision to leak 1 byte at a time rather than a single bit. This design decision caused the accuracy of the user mode implementation to suffer.

V. Discussion and Future Work

Feasibility in the Kernel. The results from our project demonstrate a clear limitation in feasibility of the GhostRace attack, as presented in its original paper. While the attack is *possible* under specific circumstances, the overall frequency of SCUAF gadgets available in the vast majority of deployed Linux distributions is sparse. We showed that out of the *entirety* of the kernel source, there exists no generally reachable gadget that doesn't require access to a specific device. Further, the sequence of techniques required to mount a GhostRace attack furthers the complications in wide feasibility and is largely limited without access to an IPI mechanism - something that has already been recently mitigated.

SCUAF in Other Codebases. In our project, we focus specifically on reimplementing the GhostRace attack on the Linux Kernel, following directly from the original paper. While we found it improbable for an attacker to mount such an attack in the kernel, it has, to our knowledge, not been explored in other popular codebases. The Coccinelle scripts, along with our filters, could be easily leveraged to scan codebases commonly installed in popular Linux distributions.

ARM, Heterogenous, and More We had originally planned to present some initial findings on GhostRace applied to an ARM platform; however, [2] performed verification of SRCs impact on other major hardware platforms, confirming that the compare and exchange instruction sequence is universally vulnerable. Thus, at the software level, all platforms utilizing conditional branches in the implementation of synchronization primitives are vulnerable to GhostRace attacks or near derivatives. Further, a small branch of Spectre-related work has investigated the impact of these attacks of heterogenous platforms [7]. Due to the highly parallel, shared data-based paradigm of these systems, a natural extension of GhostRace could investigate similar attack mechanisms affecting GPUs, FPGAs, and other hardware-based accelerators.
Leak Rate The authors of [2] claimed they achieved a 12Kb/s leak rate. We were unable to reproduce these results. To leak a single byte, we must do the following: train the lock branch, create the timer resources, create and start the victim and IPI storm threads,leak the secret by speculative access to the critical region - which will train the lock branch to not taken, recover the leaked secret via the side channel, clean up timer resources. To perform these steps take time, and because the authors did not release their full implementation we were not able to compare theirs to ours.

VI. Conclusion

For our final project, we conducted an investigation of Speculative Race Conditions following *GhostRace* and provided an evaluation of their reproducibility, feasibility, and impact on the security of modern Linux systems. Notably, we showed that, in general, Speculative Race Conditions do not currently affect the security of the newest Linux Kernel. However, we demonstrated that it is possible to reproduce a derivation of a full GhostRace-based information disclosure attack in user space or in a loadable kernel module. Our results confirm that, despite its relative obscurity and difficulty, GhostRace presents a reasonable risk to modern systems and should be investigated for further impact in other codebases, operating systems, and hardware platforms.

References

[1] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, January 2009.
[2] Ragab Hany, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. GhostRace: Exploiting and mitigating speculative race conditions. In *(to appear) 33rd USENIX Security Symposium (USENIX Security 24)*, 2024.
[3] Linux Kernel. sched/membarrier: reduce the ability to hammer on sys_membarrier. Accessed: 2024-05-01.
[4] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.
[5] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. Pspray: Timing Side-Channel based linux kernel heap exploitation technique. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6825–6842, Anaheim, CA, August 2023. USENIX Association.
[6] Linux Programming Manual. epoll - linux manual pages. Accessed: 2024-05-01.
[7] Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, and Nael Abu-Ghazaleh. Microarchitectural attacks in heterogeneous systems: A survey. *ACM Comput. Surv.*, 55(7), dec 2022.
[8] NIST. Cve-2024-26602 detail. Accessed: 2024-05-01.
[9] Google Project Zero. Racing against the clock – hitting a tiny kernel race window. Accessed: 2024-05-01.