The Thundering Herd: Amplifying Kernel Interference to Attack Response Times

Samuel Mergendahl* Samuel Jero* Bryan C. Ward*‡1 Juliana Furgala* Gabriel Parmer† Richard Skowyra*
*MIT Lincoln Laboratory, †The George Washington University ‡Vanderbilt University

***Abstract*—Embedded and real-time systems are increasingly attached to networks. This enables broader coordination beyond the physical system, but also opens the system to attacks. The increasingly complex workloads of these systems include software of varying assurance levels, including that which might be susceptible to compromise by remote attackers. To limit the impact of compromise, μ-kernels focus on maintaining strong memory protection domains between different bodies of software, including system services. They enable limited coordination between processes through Inter-Process Communication (IPC). Real-time systems also require strong temporal guarantees for tasks, and thus need temporal isolation to limit the impact of malicious software. This is challenging as multiple client threads that use IPC to request service from a shared server will impact each other's response times.**

To constrain the temporal interference between threads, modern μ-kernels often build priority and budget awareness into the system. Unfortunately, this paper demonstrates that this is more challenging than previously thought. Adding priority awareness to IPC processing can lead to significant interference due to the kernel's prioritization logic. Adding budget awareness similarly creates opportunities for interference due to the budget tracking and management operations. In both situations, a Thundering Herd of malicious threads can significantly delay the activation of mission-critical tasks. The Thundering Herd effects are evaluated on `seL4` and results demonstrate that high-priority threads can be delayed by over 100,000 cycles per malicious thread. This paper reveals a challenging dilemma: the temporal protections μ-kernels add can, themselves, provide means of threatening temporal isolation. Finally, to defend the system, we identify and empirically evaluate possible mitigations, and propose an admission-control test based upon an interference-aware analysis.

I. INTRODUCTION

Timing predictability and correctness is a paramount design consideration in many embedded systems, cyber-physical systems, and safety-critical systems. Many such systems include hard real-time (HRT) tasks, where unexpected jitter or latency can cause deadline misses that can have catastrophic physical consequences. It is therefore imperative to maintain temporally correct execution for such tasks, even while processing other workloads with less stringent temporal requirements. This fundamental challenge has motivated over four decades of research dating back to seminal results such as the sporadic server [1], and up through current research on mixed-criticality scheduling (MCS) [2], [3].

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8702-15-D-0001 and the National Science Foundation under Grant CPS-1837382. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense or the National Science Foundation.

[1]Work conducted at MIT Lincoln Laboratory.

A guiding philosophy of much work on temporal budgeting and mixed-criticality scheduling is that of *temporal isolation*. Hard real-time safety-critical tasks should be capable of meeting their real-time requirements even in the presence of high demands placed on other aspects of the system. In its original formation, the sporadic server was presented to improve the quality of service (QoS) of aperiodic tasks while ensuring periodic HRT tasks could still safely execute. More recently, much work on MCS is motivated by the fact that determining task worst-case execution times is notoriously difficult, especially on modern complex architectures. In the (hopefully) exceedingly rare case that high-criticality jobs run longer than expected (*e.g.,* due to rare microarchitectural effects), MCS provides a means of maintaining the temporal-correctness guarantees of high-criticality tasks.

While temporal-isolation mechanisms were principally designed for reliability and QoS, they are also important *for security*. Without temporal isolation, malicious actors could trivially perform denial of service (DoS) attacks to consume processing time needed by HRT tasks, delaying or even preventing the execution of safety-critical tasks.

The need for temporal isolation is also evident in the sharing of functional services. Some Operating Systems (OSes) that focus on providing a secure execution environment for embedded and real-time computation are structured as μ-kernels. Services are implemented as user-level servers accessed by multiple clients through Inter-Process Communication (IPC). Fast IPC between processes is often *synchronous* [4] and mimics the control flow of function calls, switching from the clients to server, and back. As synchronous IPC binds potentially untrusting clients and servers, it is particularly security sensitive and has a history of challenges [5]. Early performance-driven IPC mechanisms avoided scheduler interaction on IPC, thus eliding proper client/server accounting [6]. Pairing temporal isolation facilities based around limited thread execution budgets with synchronous IPC opened systems to budget attacks in which clients attempt to expend a server's budget to prevent it servicing other clients [7], [8]. Additional improvements to the temporal properties of synchronous IPC have included priority-order execution of clients [7], adding priority [8], [9] and budget [7], [8] inheritance.

This paper introduces Thundering Herd Attacks on the synchronous-IPC and budget-management mechanisms in OS kernels. In the classical Thundering Herd problem [10], many

threads waiting on some event are woken up but only one is actually able to proceed, causing the other threads to consume resources before blocking again. Similarly, in our Thundering Herd Attacks, a large number of normal application threads methodically use IPC facilities and carefully consume budget in a manner that causes kernel execution commensurate with the number of threads. Many of the mechanisms added to ostensibly improve temporal isolation inadvertently enable this class of attacks. Most kernels employ non-preemptive execution to control concurrency. However, when the kernel execution caused by the Thundering Herd runs non-preemptively, long stretches of non-preemptive execution interfere with and delay the activation of high-priority threads. This can threaten the high-priority thread's ability to meet deadlines.

This leaves us with what we call the *system-coordination dilemma*: if we use simple IPC mechanisms, then the system suffers from inter-client interference, thread accounting is unpredictable, and execution is unconstrained; if we counter inter-client interference by making IPC priority aware, then attacks on the kernel's priority mechanisms cause significant interference; if we counter unpredictable system accounting and lack of execution isolation by using budgets, then the budget-accounting mechanisms cause significant interference. The techniques employed to increase the intelligence of thread coordination also lead to significant attacks. Many of these issues are quite nuanced and not immediately obvious, especially to application engineers instead of systems researchers. This paper therefore sheds light on this fundamental scientific dilemma, and illustrates specific tradeoffs.

This paper first covers μ-kernel implementation background (§II) and introduces the system threat model (§III). Then it introduces a number of known interference issues on the IPC interactions of the `seL4` μ-kernel. These issues (§IV) make relatively restrictive assumptions and focus on the challenges in predictable IPC interactions when an arbitrary number of attacking client threads harness a server. Following this, this paper makes a number of contributions:

- In §V, we introduce Thundering Herd Attacks that cause attacker-controlled lengths of non-preemptive kernel execution that can cause deadline misses and temporal violations.
- We quantify the impact of (§VI-C) three instances of these attacks in `seL4`'s Mixed Criticality Scheduling extensions that target the improved temporal facilities for properly prioritizing IPC and implementing budgets.
- We identify and discuss two mitigations for our Thundering Herd Attacks, implementing and empirically characterizing the performance of one (§VI-D) and qualitatively examining the other (§VI-E).
- We propose an admission-control test (§VII) that determines if a system is schedulable despite the attacks.
- Finally, we discuss design options to mitigate these attacks (§VIII) and how they may be applied in other μ-kernels.

II. Background

To better understand the complexities of μ-kernel construction and the trade-offs they make, this section discusses the historical designs of synchronous IPC and budgets.

A. μ-kernel Background

μ-kernels are focused on the placement of Operating System (OS) functionality in user-level processes called "servers" instead of in the kernel. When applications or servers wish to harness the services provided by a server, they make the request using Inter-Process Communication (IPC). As IPC is on the critical path of all system service requests, a significant focus of μ-kernels is on its optimization. L4 μ-kernel variants [11] implement IPC with *synchronous rendezvous between threads*. Synchronous IPC features control flow that mimics a function call where the client thread blocks until the server thread returns.

B. Synchronous IPC: Optimization and Predictability

There has often been a tension between performance and timing predictability in μ-kernel design. Liedtke's L4 [4] innovated synchronous IPC paths with overheads close to those of the hardware by avoiding software overheads such as scheduling, and data queuing. A client requesting a service from a server would directly switch to the server (and directly back to the client on return) without executing scheduler logic. This had a negative side-effect of introducing inaccuracy into execution time tracking and prioritization. During an IPC, execution is accounted to the client and the server executes at the client priority *unless* a timer interrupt updates the scheduler. In this case, IPC execution from the timer tick onward is accounted to the server, at its priority. The overhead of invoking the scheduler during IPC is non-negligible – imposing more than a 30% slowdown [6] – but it increases system predictability.

The core questions for *predictable* IPC using synchronous rendezvous between threads are: (1) which priority is used for the server, and (2) which thread's budget should be used? These decisions are complicated by the need for predictable resource-sharing protocols. The need for these protocols is due to higher-priority clients that request service from a server while it is already servicing a lower-priority client. This *contention* on the server thread[2] requires kernel queueing of client service requests. A key design of L4 is that the threads themselves are queued, rather than the data. This removes the need for memory allocation on the critical IPC path, but requires a queueing discipline that affects predictable service.

Modern L4 variants resolve the contention issue by either enabling the server to inherit the priority of all client threads, or avoiding inheritance and requiring system designers to carefully choose server priorities as the ceiling of all clients. Priority inheritance can be difficult to implement, and variants that use it often rely on the scheduler to choose the highest-priority thread and walk through its (sometimes transitive) dependencies to find an active thread to execute.

[2]We discuss contention on the server *thread* for simplicity, despite IPC endpoints that act as a level of indirection between client and server threads being a popular current design [8], [9], [12].

C. Rate-limiting Policies and Synchronous IPC

Systems must often constrain the execution of low-assurance code to prevent it from unduly interfering with other functionalities. Budget-driven servers[3] are a traditional mechanism for limiting execution interference over time. For example, deferrable servers [1] limit execution over fixed windows of time. A thread's budget has an initial value, and the budget is depleted corresponding to a thread's execution. When budget is exhausted, the thread is suspended awaiting replenishment. A *replenishment policy* determines when the budget is increased for a thread. For example, deferrable servers replenish up to the initial budget value periodically.

IPC between threads complicates budget-management policy. When a server computes for a client, it can use its own budget, or it can *inherit* the budget of the client (a relationship that can be transitive). The former can lead to budget attacks, where a client might aim to expend a server's budget, thus delaying (until replenishment) the server from servicing other (perhaps latency-sensitive) clients. The latter requires us to define a policy for when the client provides insufficient budget to complete the execution of the server's functionality.

D. seL4 IPC and Rate-limiting Policies

seL4 has a number of design decisions that represent trade-offs between IPC efficiency, predictability, and the functional verifiability of the kernel code-base [13].

Mutually exclusive kernel execution. One important design consideration is that kernel execution is not concurrent nor parallel (*i.e.,* a single sequential execution flow executes kernel logic at a time). This makes functional verification possible and in practice means that the kernel executes with interrupts disabled (*i.e.,* it is non-preemptible). On multicore systems, it executes within a lock (*i.e.,* a big kernel lock). This lock is a demonstrated attack vector [14] whereby computation on one core can delay processing on another core. In this paper we focus on a single core; however, because seL4 uses a big kernel lock and our attacks target kernel computation within that lock, our attacks apply equally to the multicore case.

There is a tension that exists between the mutually exclusive execution of the kernel and kernel operations that execute for a potentially unbounded number of iterations [15] (*e.g.,* revocation of capabilities). seL4 uses *preemption points* to solve this, whereby kernel execution will back out of a loop after a fixed number of iterations and process any pending interrupts. The thread resumes the loop where it left off. This effectively adds controlled and explicit kernel preemptions.

Server contention queuing and priority. Clients that use IPC to request service from a server awaiting IPC are queued. The waiting server may be currently executing a request or blocked on another operation. seL4's default policy uses FIFO queueing of these client threads. This FIFO order does *not* represent a priority-sorted order, but it does guarantee client progress. Each client must only wait for a fixed number of threads before it receives service. Servers do *not* inherit client priorities. Therefore the system designer must carefully assign priorities at the ceiling of the clients should predictable service be required. Such a policy represents the Immediate Priority Ceiling Protocol (I-PCP) [16].

Rate-limiting policy and Mixed-Criticality Extensions. Default seL4 does not provide rate-limiting policies. However, the seL4 MCS extensions (henceforth referred to as seL4-MCS) [7] that are intended to replace the existing seL4 mechanisms include mechanisms and policies for budget management. Importantly, servers can be *passive*, inheriting the budget (though not the priority) of client threads upon IPC. These budgets are implemented as sporadic servers. If a budget is depleted while executing a server, a *temporal exception* activates a policy server that can decide to provide enough budget to finish server execution, or to take other remedial action (*e.g.,* extending the budget to include the rest of the server's execution).

Additionally, seL4-MCS uses priority-sorted IPC wait queues. This has an impact on IPC performance as it converts a simple constant-time operation to enqueue a thread into an iterative operation. The priority-sorted wait queues use simple linked lists, resulting in *O(n)* complexity.

Thread Creation and Control. seL4's capability system provides the means to create and control threads. All memory in the system is initially `untyped` and has to be retyped into capabilities for use by the system, including as new threads, known as `TCB` capabilities. To be usable, a thread will probably need an IPC buffer and stack, which can also be created from untyped memory. Hence, untyped memory is required to create new threads in seL4.

Using the previously mentioned `TCB` capability, a new thread can be started. However, to adjust the priority of this thread, another `TCB` capability must be used. This second `TCB` capability must have a maximum priority greater or equal to the desired priority for the new thread. In practice, this means that a thread with access to untyped memory and its own `TCB` capability can start more threads of equal or lesser priority.

seL4-MCS extends the capability system with `SchedContext` capabilities. These capabilities describe and track the budget and period of a thread. In seL4-MCS, a `SchedContext` capability must be added to each `TCB` capability in order for the thread to be runnable. While the `SchedContext` capability can be created from untyped memory, configuring it requires access to a `SchedControl` capability given to the root-task at boot. In practice, any component of a system that starts new threads must either have a copy of this `SchedControl` capability or be able to issue a request to an admission-control server that can populate the `SchedContext` budget and period.

III. Threat Model

In this work, we consider a system where applications exist as processes with separate threads and memory address spaces. Some of these applications may be highly critical (*i.e.,* a vehicle steering application), while others may be less so (*i.e.,* a vehicle infotainment system). These applications interact

[3]"Servers" here refers to the logic associated with the budget. To disambiguate, we'll refer explicitly to budgets and budget management.

	Job Scheduled		Work for a Low-Priority Thread		Blocked on I/O		Budget Replenished
	Work on Behalf of Victim		Blocked on IPC Endpoint		Replenishment Period		Budget Spent

	Synchronous IPC to Server		I/O Fired		Replenishment Queue Sort
	Request I/O		IPC Endpoint Queue Sort		Replenishment Processing

Fig. 1: Legend used throughout attack examples.

with one or more shared services, which are also processes, to accomplish their goals. These shared services could manage hardware devices, like a timer or network interface, or they could provide common abstractions, like buffered channels or key-value stores. Additionally, we assume that applications initially receive some quantity of raw memory from which to create any resources and kernel objects they require.

Such a design is a common way to realize typical embedded and cyber-physical systems. While a system could be designed with all applications as threads sharing a memory address space, such a design makes it easy for applications to interfere with each other and modify each other's private state, making minimal use of the strong isolation abilities provided by a μ-kernel. Furthermore, functionalities are increasingly being consolidated onto shared computing platforms to reduce the size, weight, and power (SWaP) of systems.

In some systems it may be possible to pre-plan all IPC paths, memory pages, and other kernel objects used in the entire system, allowing only the needed resources and memory to be given to each process at startup (as in CAmkES [17]). This is limited to static systems, where the resources required by all components can be easily determined. For more dynamic systems, such as one built on top of an OS Patina [18], or systems that integrate components from different vendors, this kind of determination may be impractical or impossible. Instead, each component can be assigned a limited amount of raw memory from which to create the kernel objects (threads, IPC endpoints, *etc.*) and memory pages it needs, with the limit on that memory chosen to protect the rest of the system.

The dynamic behaviors of processes includes the creation of an a-priori unknown number of threads. As such, we assume the use of execution budgets and admission-control tests on all threads. Such tests are often permissive in the admission of threads at the lowest priority as they generally have limited interference with higher-priority threads.

We consider an attacker who has compromised a low-assurance process (*e.g.,* code that is not certified or fully trusted) and seeks to interfere with mission-critical processing. The attacker cannot directly communicate with these critical processes, but seeks to influence them indirectly via shared servers, thread-execution interference, or the μ-kernel itself.

While assuming that a malicious attacker has compromised part of the system might be considered admitting defeat at the outset, experience shows that stopping all compromise across a system is effectively impossible. Formal verification approaches have provided functional verification of kernels [12], which increase the barrier for attacks on the system. However, user-space processes, shared services, and non-functional properties (*e.g.,* timing) remain vectors for potential attacks.

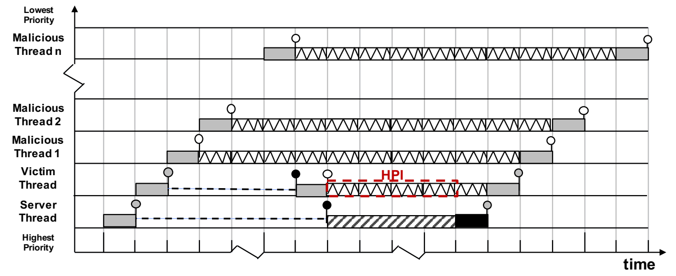


Fig. 2: FIFO Endpoint Flood Interference example schedule.

Such attacks can be conducted via a number of possible vectors, such as a remote attacker exploiting a vulnerability and hijacking control, or a software supply-chain exploitation in which an attacker may be able to compromise the code provided by a vendor. The latter has been used effectively by attackers in non-real-time domains, such as the recent SolarWinds attack [19] and the 2017 NotPetya attack [20].

Thundering Herd Attacks can be especially effective if the malicious process knows when the victim thread will activate so that interference can be created at that point. For this, an attacker could use previous approaches to reverse-engineer the activation timing of high-priority threads [21]–[23].

Finally, we assume that the other components in the system, namely the μ-kernel, the high-assurance applications, and any shared services, are benign and uncompromised.

IV. Traditional IPC Interference

In this section, we describe a series of well-known, synchronous-IPC-based interference issues that delay high-priority tasks that use shared services. These issues serve as the motivation for the mechanisms that are exploited by our Thundering Herd Attacks in §V. In particular, though the academic community has introduced these issues previously, in order to fully understand the system-coordination dilemma that drives our Thundering Herd Attacks, we provide a detailed analysis of these relevant interference issues known by the community. For each interference issue, we provide an overview of the issue, describe the issue in detail, and finally, discuss existing mitigations. Moreover, we later quantify the negative effect of these interferences in §VI.

A. Overview

In order to create these interferences, we leverage a set of low-priority threads to cause a high-priority shared server to perform work on behalf of low-priority threads, instead of either a) working for the high-priority victim task or b) scheduling the high-priority victim task itself. We consider any such work that the server executes *on behalf of* a low-priority thread to be high-priority interference (HPI). Note that, following the discussion in §II-B, we assume that the shared server is executing at a higher priority than any client. Lastly, for each attack example, we refer the reader to Fig. 1 for the notation found in each figure.

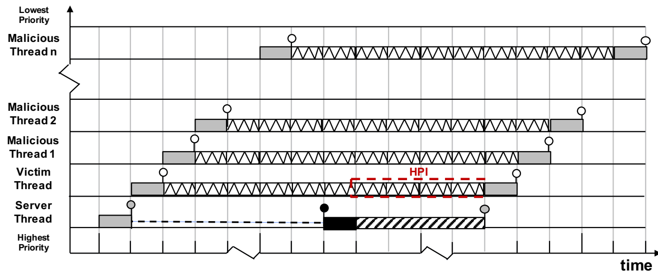


Fig. 3: Priority Ceiling Processing Interference example schedule.

B. *FIFO Endpoint Flood Interference*

The first HPI we describe is the FIFO Endpoint Flood Interference discussed by Liedtke, *et al.* almost 25 years ago [24]. We depict this in Fig. 2. In this HPI, a set of low-priority threads all make synchronous IPC requests to a currently blocked shared server before a high-priority victim task also makes a synchronous IPC request. If the IPC endpoint queue is a first-in, first-out (FIFO) queue, as in seL4 and other common μ-kernels, then this forces the shared server to receive and process the (low-priority) requests from the low-priority threads before the request from the high-priority task. This period in which the server executes on behalf of the low-priority threads instead of the higher-priority client is HPI.

We note that a server executing on behalf of a client shares some commonalities, at least analytically, with executing a critical section within a lock. In the seminal work on the Priority Ceiling Protocol (PCP) [16], it was identified that self-suspensions within critical sections invalidate the PCP analysis. This interference effectively exploits this same phenomenon, and reinforces that neither critical sections nor servers executing on behalf of a blocked client should suspend. While seL4 has never, to the best of our knowledge, documented that servers should never suspend, it has been part of the informal developer knowledge in the seL4 community for some time [25]. Additionally, we note that self suspensions are difficult to analyze, and can have surprising consequences, as show by Chen et al. [26] who demonstrated that many papers on the topic going back as early as 1994 are incorrect due to common misconceptions about suspensions.

Interference Mitigations. In order for this HPI to occur, there must be a moment when both the victim task and the shared server are blocked. Thus, one potential mitigation is to either not allow the victim task or the shared server to block. For the victim task this is often an unreasonable request, as it may be periodically activated or block waiting on some long-duration operation. For the shared server, this is more reasonable, but may still be problematic, especially in the case of a server for an I/O device, which may need to occasionally block waiting for the I/O device to catch up. This momentary suspension of the shared server, however, allows the lower-priority threads the opportunity to run and queue on the server's endpoint, resulting in this interference.

Instead, in order to mitigate this interference, seL4-MCS introduces priority sorting for its IPC endpoint queues. Because the victim task has higher priority than the malicious threads, it will sort to the front of the IPC endpoint queue, and eliminate this particular set of HPI.

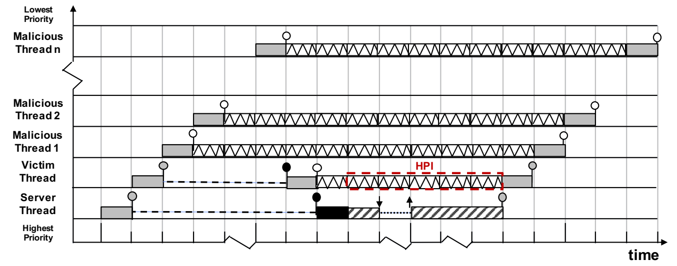


Fig. 4: Budget Drain Interference example schedule.

C. *Priority Ceiling Processing Interference*

We now describe Priority Ceiling Processing Interference, depicted in Fig. 3. This interference again leverages the results from the Priority Ceiling Protocol (PCP) [16] work. However, in this case, the IPC endpoint queue does not have to be FIFO; it may be priority-sorted or arbitrarily ordered. Without loss of generality, we assume a priority-sorted endpoint queue, like provided by seL4-MCS, as that protects against the previous FIFO Endpoint Flood Interference. Similar to the previous FIFO Endpoint Flood Interference, the victim is a high-priority task that performs a synchronous IPC request to a shared server and a set of low-priority threads to introduce interference. Again, the shared server executes at a higher priority than all clients and we assume that it blocks, possibly due to handling I/O.

When the shared server is blocked, the victim task can run and perform its IPC request to the server, resulting in it being queued on the endpoint. Now, all the low-priority threads run and make IPC requests and queue on the endpoint. Note that whatever order these clients run, the victim task will be sorted to the front of the priority-ordered queue. When the server wakes up, it will process the request from the victim task first. However, once that request is completed, the server will go on to process the requests from the low-priority threads instead of allowing the high-priority victim thread to be scheduled, because the shared server runs at a higher priority than all its clients. Thus, this period between when the shared server completes processing the request from the victim thread and when that thread is actually scheduled is HPI.

Interference Mitigations. Previous studies have investigated the issues with priority ceiling processing [16], and the general recommendation is that, in order to prevent this problem, shared-server threads should be designed to never block. When high-priority servers do not suspend, the low-priority threads are handled as they arrive and there is no opportunity for a large queue to form. Although seL4-MCS does not contain an explicit defense against this interference, its design philosophy follows this recommendation that servers never block, as evident in its sporadic-server design [25], [27].

D. *Budget Drain Interference*

Finally, we describe Budget Drain Interference, which enables low-priority threads to delay a high-priority victim task

by making repeated requests to a shared server. In particular, as discussed by Shapiro [5], low-priority threads can make repeated requests to a shared server to drain its budget. Then, when the victim task makes a request to the shared server, it must wait for the server's budget to be replenished before its request is processed. This requires an opportunity for the malicious threads to make many requests to the shared server, but can have very large impacts. See Fig. 4 for more details of this HPI. The exact HPI that results scales with the replenishment period of the server's budget. We refer to prior work on this interference for a more exact measure of its impact [5].

Interference Mitigations. Mitigations to this HPI include sporadic servers as well as budget inheritance/donation systems that allow servers to use a client's budget for processing their requests. seL4-MCS includes support for passive threads, requiring budget donation, as well as support for sporadic servers that can track a bounded number of replenishments [27].

E. Relationship to the System-Coordination Dilemma

As discussed, a variety of mitigations exist to prevent these interferences. However, in many cases, particularly priority-sorting IPC endpoint queues, providing sporadic servers, and providing budget inheritance, these mitigations result in additional complexity in μ-kernel implementations. Many μ-kernels, including seL4-MCS, implement these additional features to protect against these kinds of HPI.

We show next that this additional complexity can be attacked, resulting in our Thundering Herd Attacks. Thus, the system-coordination dilemma requires that a system designer must choose between HPI from these traditional interference issues, and HPI from Thundering Herd Attacks.

V. Thundering Herd Attacks

In this section, we introduce and describe our Thundering Herd Attacks. These attacks target the mitigations deployed against the traditional IPC interference issues (discussed in §IV) to introduce additional non-preemptive kernel processing into the system with the aim of causing schedule overruns for high-priority tasks. We first provide an overview of these attacks and then discuss each of them in detail.

A. Overview

We refer to these attacks as Thundering Herd Attacks because they use many attacker-controlled threads to perform IPC and consume budget in ways that force the kernel mechanisms necessary for handling budgets and prioritized queues to do large quantities of non-preemptive work. This kernel-level work supersedes execution of all user-space threads, and, when combined with a non-preemptable kernel, like seL4 (see §II), these attacks can even supersede interrupts.

The kernel mechanisms we exploit are necessary to mitigate the previously discussed IPC interference issues. In particular, we leverage priority-sorted IPC endpoint queues and sorted queues of threads waiting for budget replenishment. This

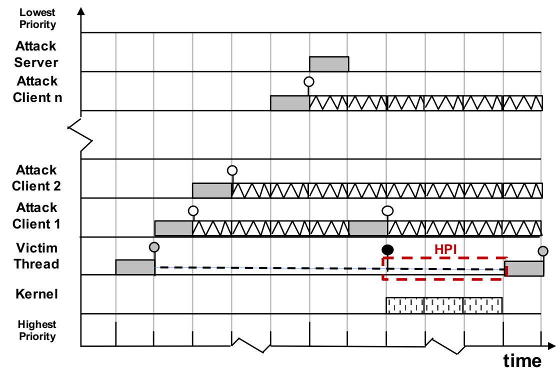


Fig. 5: Endpoint Queue Sorting Attack example schedule.

forces a system-coordination dilemma on the user: either deal with the client-interference issues illustrated in the traditional IPC interference issues or deploy defense mechanisms against the traditional IPC interference issues and be exposed to our Thundering Herd Attacks.

For each Thundering Herd attack, the attacker leverages a set of low-priority threads. However, unlike the traditional IPC interference issues, a shared server is no longer required. By manipulating its own threads, the attacker can cause extended non-preemptive kernel processing that will delay any thread or interrupt in the system. Similar to §IV, we consider this kernel-level processing *on behalf* of a low-priority thread, when the high-priority victim thread is ready, to be HPI. We emphasize that this is possible even when the attacker threads and the victim thread are *completely disjoint* with no shared servers or resources.

B. Endpoint Queue Sorting Attack

The first attack we describe is the Endpoint Queue Sorting Attack. This attack takes advantage of the fact that μ-kernels, such as seL4-MCS, often priority sort the threads blocked on synchronous IPC endpoints. Recall from §IV-B that this is an essential technique to mitigate the FIFO Endpoint Flood Interference. seL4-MCS uses a simple linked list to implement this priority queue, making it $O(n)$ to insert a new thread in sorted order. Threads are added to the back of the queue and sorted forward in increasing priority. The only requirement for an attacker to launch this attack is the ability to either have or create both a single endpoint and many schedulable threads at three different priorities. This is consistent with our threat model as a malicious thread may spawn other threads. We reiterate that both the shared server and its clients in this attack can be attacker controlled tasks of low priority and disjoint from the rest of the benign system.

The Endpoint Queue Sorting Attack works as shown in Fig. 5. First, the attacker finds or creates an IPC endpoint and creates a server thread to listen on this endpoint. This thread should have the lowest priority. Then the attacker iteratively starts a series of attack threads with the middle priority and causes them to perform a synchronous IPC on the IPC endpoint. This generates a long queue of threads on the endpoint, but without any sorting as they are added. The attacker then creates another thread with its higher priority that will

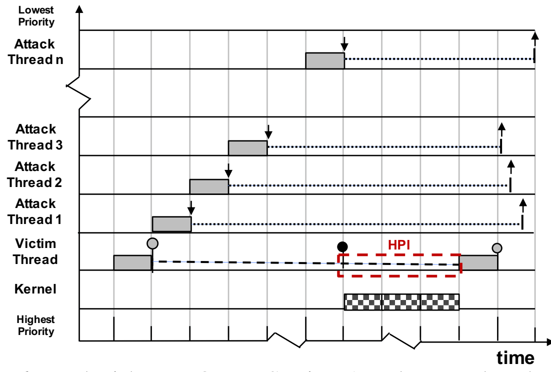


Fig. 6: Replenishment Queue Sorting Attack example schedule.

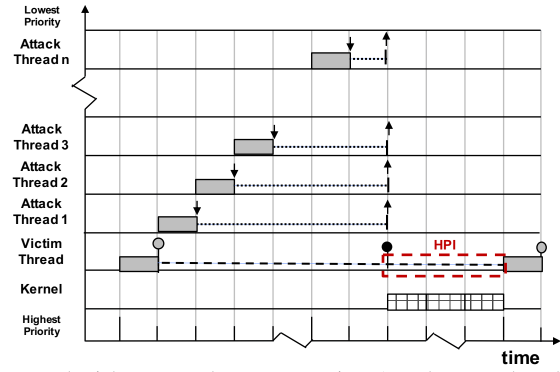


Fig. 7: Replenishment Wakeup Processing Attack example schedule.

repeatedly perform a synchronous IPC on the IPC endpoint. When this occurs, the kernel will sort this attacker thread to the front of the endpoint queue. At this point, the attacker's server thread runs, handling the attacker's higher-priority thread and allowing the attacker's higher-priority thread to run again. The higher-priority thread will immediately make another IPC call which will force it to be sorted to the front of the endpoint queue again. When the high-priority victim thread becomes runnable, often as a result of an interrupt, it will be delayed due to the kernel non-preemptively sorting the endpoint queue, resulting in HPI.

Attack Mitigations. Because the kernel uses a linked list for its priority queue, insertion is $O(n)$. One way to mitigate this attack would be to use a different data structure with smaller insertion complexity for the endpoint priority queues. For example, a red-black tree would offer $O(log(n))$ worst-case performance against our attack, which could reduce the impact of the attack, but not eliminate it altogether (see §VI).

Another option would be for the kernel to maintain a separate queue for each priority level on each IPC endpoint (similar to the `plist` data structure found in Linux [28]). If there were a separate queue for each priority, there would be no need to sort the queues, which would eliminate the vector for this attack. With no queues to sort, insertion would again be $O(1)$, eliminating the HPI caused by the Endpoint Queue Sorting Attack at the cost of a small amount of memory.

C. Replenishment Queue Sorting Attack

Next we introduce the Replenishment Queue Sorting Attack. This attack leverages the fact that when a thread expends its budget, it is placed on a queue of pending replenishments. Replenishments are essential to budgets which, as discussed in §II, are critical to constrain the execution of low-assurance threads and prevent their interference with other threads.

seL4-MCS implements replenishment as set of replenishment queues, one for each core, with each queue sorted based on soonest period expiration (*i.e.,* the thread that will be replenished soonest). In seL4-MCS these are, again, implemented as priority-ordered linked lists, making insertion of new threads $O(n)$. As a result, we can linearly increase the computation required to enqueue a thread by increasing the length of this queue. The only requirements for an attacker are the ability to either have or create many schedulable threads at the same priority and to have some control over their budgets and periods. Note that this attack does not rely on IPC.

The Replenishment Queue Sorting Attack works as shown in Fig. 6. The attacker creates and runs a large number of attack threads that execute infinite loops to spend their budgets and be queued in the replenishment queue. The periods of these threads, however, are carefully chosen. Since seL4-MCS inserts a new thread at the front of the replenishment queue and then sorts it towards the back, inserting threads with longer periods results in more sorting while threads with shorter periods are $O(1)$. As a result, each of the attack threads should have a shorter period than the one before it, thus generating a large replenishment queue without sorting. Finally, to trigger the attack, the attacker creates and executes a thread with a period longer than any other thread. As a result, once this thread exceeds its budget, it will be sorted all the way to the back of the replenishment queue. When the high-priority victim thread becomes runnable, often as a result of an interrupt, it will be delayed due to the kernel non-preemptively sorting the replenishment queue, resulting in HPI.

The specifics of exactly how the replenishment queue is sorted are immaterial to this attack. For example, if the kernel instead inserted threads at the tail of the replenishment queue and sorted the soonest-to-be-refilled replenishments to the front, the attacker would initialize periods in the opposite manner: the first threads would have increasingly larger periods, with the last attack thread having the shortest period.

Attack Mitigations. As with the Endpoint Queue Sorting Attack, potential mitigations include data structures with lower insertion complexity (*e.g.,* red-black trees). However, unlike the Endpoint Queue Sorting Attack, because the Replenishment Queue Sorting Attack does not target a queue sorted by priority or leverage threads of different priorities, separate queues per priority will not affect the outcome of this attack. In particular, the replenishment queue is sorted based on time-until-replenishment, not the priority of the respective thread.

D. Replenishment Wakeup Processing Attack

Finally, we introduce the Replenishment Wakeup Processing Attack. Similar to the previous attack, this attack takes advantage of the queue of pending budget replenishments maintained by seL4-MCS. However, rather than focusing on

the sorting that occurs when a new thread is added to the replenishment queue, this attack focuses on the processing that occurs when threads are ready to be replenished. In particular, this attack attempts to cause a large number of attacker-controlled threads to be replenished at the same moment, directly prior to the execution of the high-priority victim thread. This will cause HPI as the low-priority attacker threads are replenished instead of running the high-priority victim thread. The only requirements for an attacker are the ability to either have or create many schedulable threads and to have some control over their budgets and periods. This could be realized by either having a copy of the `SchedControl` capability, or issuing a request that is serviced by an admission-control server to populate the `SchedContext` budget and period.

The Replenishment Wakeup Processing Attack works as shown in Fig. 7. The attacker creates and runs a large number of attack threads. These threads execute infinite loops to spend their budgets and be queued in the replenishment queue. However, their budgets and periods need to be carefully chosen. In particular, the attacker must choose the budget and period of each thread such that all threads will be replenished at the same moment (or at least within the same timer tick). This moment when all the threads need to be replenished is chosen to be just before the high-priority victim thread is ready. As a result, when the high-priority victim thread becomes runnable, often as a result of an interrupt, it will be delayed due to the kernel non-preemptively processing the replenishment queue, resulting in HPI.

Attack Mitigations. The Replenishment Wakeup Processing Attack appears to illustrate a fundamental instance of the system-coordination dilemma. As discussed in §II, kernel processing of replenishments is fundamental to proper enforcement of temporal budgets. Moreover, as we will see in §VI, if a system designer deploys a logarithmic data structure to alleviate the Replenishment Queue Sorting Attack, such a logarithmic data structure will *increase* the effect of the Replenishment Wakeup Processing Attack. Namely, while the logarithmic data structure decreases insert processing from $O(n)$ to $O(log(n))$, it also *increases* deletion from $O(1)$ to $O(log(n))$.

Lastly, because the kernel processes all available replenishments at once, a separate queue per priority will not mitigate the Replenishment Wakeup Attack alone. In particular, even if the replenishments were in different queues by priority, the current design of the kernel requires it to process all replenishments to avoid replenishment starvation (including replenishments associated with low-priority threads). However, while separate queues per priority alone would not mitigate the attack, such an implementation could enable a new priority-aware replenishment processing scheme in the kernel. Unfortunately, such a design would require significant structural changes to the kernel (e.g., a significant redesign of most aspects of the system including the IPC path). Nonetheless, such a redesign could prove fruitful, with the caveat of a careful examination of the trade-offs of such an implementation.

VI. Evaluation

In this section, we implement and quantify both the synchronous-IPC-based interference issues that delay high-priority tasks using shared services (§IV) and our Thundering Herd attacks that introduce additional non-preemptive kernel processing into the system (§V). We then empirically evaluate a possible mitigation strategy for our Thundering Herd Attacks that modifies the seL4-MCS kernel to use red-black trees to priority sort both the IPC endpoint queues and replenishment queues that the kernel maintains with logarithmic complexity. Finally, we analyze the impact of another mitigation strategy that uses a queue-per-priority data structure.

A. Experimental Setup

We implement each studied attack on seL4 (or seL4-MCS) version 12.0.0 and quantify their impact using the popular Zynq-7000 XC7Z020 SoC, which includes a dual-core Arm Cortex-A9 processor running at 667 MHz and a Xilinx FPGA. We use only a single core for this evaluation and do not use the FPGA at all. We use gcc version 8.3.0 (`Debian 8.3.0-2`) for arm-linux-gnueabi-gcc to compile seL4 and our test code. We use the on-chip performance counters to determine overheads. Unless otherwise noted, all results are computed from 100 iterations. Because of the high accuracy of our testbed and test suites standard deviations are frequently very small and may not be visible in all graphs.

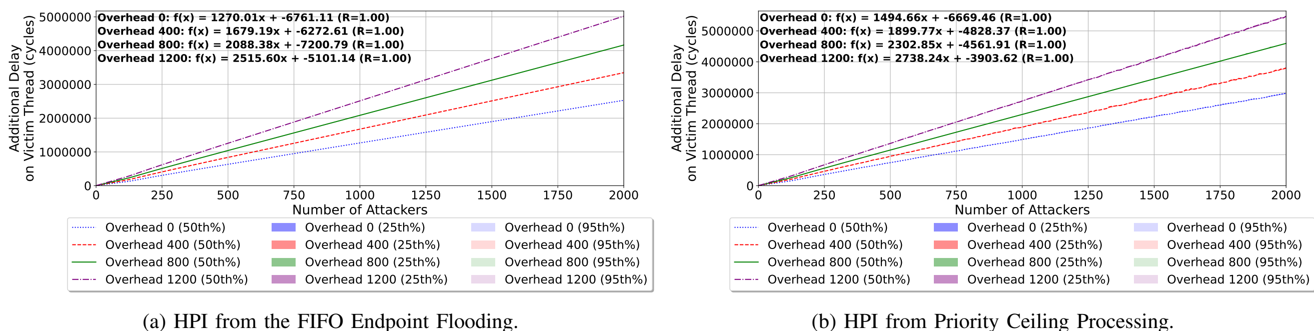
B. Traditional IPC Interference Results

We implemented FIFO Endpoint Flood Interference in our testbed using seL4 and minimal client and server applications and report its impact in Fig. 8a. We observe that significant impacts are possible, with a thousand low-priority threads causing over 1 million cycles (1.5ms) of interference. Further, as the number of low-priority threads increases we see that HPI increases linearly. We additionally perform experiments where the server performs different amounts of work for each request, which we refer to as overhead. For example, with overhead 400, the server performs approximately 400 cycles of work for each request, while with overhead 1200 the server performs approximately 1200 cycles of work for each request. Thus, the imposed delay also increases linearly with respect to the amount of work the server performs for each request.

We also implemented Priority Ceiling Processing Interference in our testbed using seL4-MCS, which priority sorts the endpoint queue, and report the impact of this HPI in Fig. 8b. Much like the prior interference issue, we observe that HPI scales linearly with both the number of low-priority threads and the amount of work the server performs on behalf of each client. Compared to the FIFO Endpoint Flood Interference, we observe that this interference is slightly more powerful in terms of the HPI from each low-priority thread.

C. Thundering Herd Attack Results

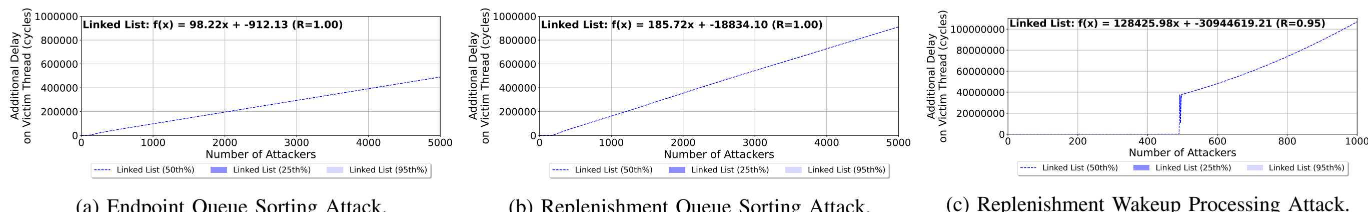
Next, we empirically evaluate the impact of the Endpoint Queue Sorting Attack on seL4-MCS in Fig. 9a. We observe that significant impacts are possible, with 1,000 attack threads



(a) HPI from the FIFO Endpoint Flooding.

(b) HPI from Priority Ceiling Processing.

Fig. 8: HPI from the traditional IPC interference issues with different numbers of threads and quantities of work for each request.

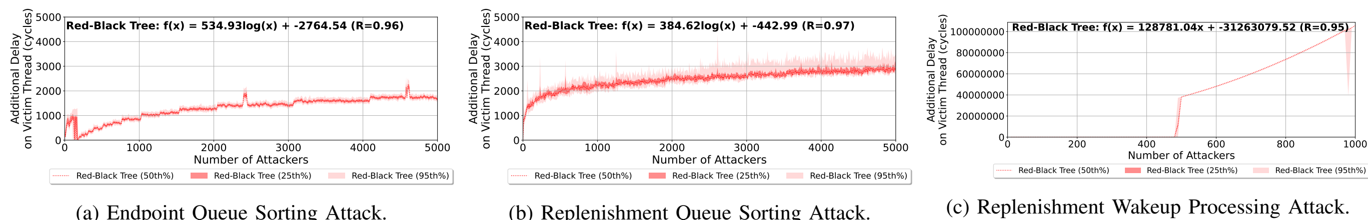


(a) Endpoint Queue Sorting Attack.

(b) Replenishment Queue Sorting Attack.

(c) Replenishment Wakeup Processing Attack.

Fig. 9: HPI from the Thundering Herd attacks with a linked-list kernel data structure.



(a) Endpoint Queue Sorting Attack.

(b) Replenishment Queue Sorting Attack.

(c) Replenishment Wakeup Processing Attack.

Fig. 10: HPI from the Thundering Herd attacks with a red-black tree kernel data structure.

introducing about 100,000 cycles (150 μs) of HPI. The introduced HPI is also linear with the number of attack threads. It is also interesting to note that, unlike the IPC interference issues discussed previously, it takes a number of attacker threads (~ 125) before an impact is noticeable. This is likely due to a combination of cache effects, where for very small numbers of threads everything may be in L2, and the difficulty of targeting the attack when it introduces only a small delay.

Similarly, we empirically evaluate the impact of the Replenishment Queue Sorting Attack on seL4-MCS in Fig. 9b. We see that the impact of this attack grows linearly with the number of attacker threads and that it requires a number of attack threads before the attack becomes noticeable. However, this attack has a larger impact for each individual attacker thread of approximately 185 cycles.

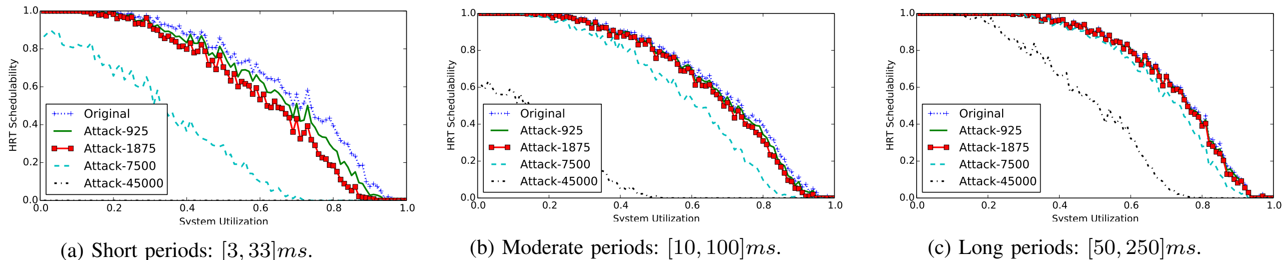
Lastly, we implemented the Replenishment Wakeup Processing Attack on seL4-MCS and demonstrate its impact in Fig. 9c. This attack is more challenging than the others to properly orchestrate since we must predict execution times very accurately so that all attacker threads will be replenished at the same instant as the victim thread becomes runnable. For a small number of attacker threads, we are unable to precisely align the attack with the victim thread, as seen on the left of Fig. 9c. However, once we get above about 450 attackers, the impact of the attack becomes large enough that we can reliably impact the victim. While this attack is harder to execute, it is also much more powerful, as each thread causes about 100,000 cycles of HPI. The impact also grows linearly with the number of threads, meaning that 1,000 attack threads can introduce approximately 100 million cycles (150ms) of interference.

D. Red-Black Tree Mitigation Results

We now consider one possible mitigation for our Thundering Herd Attacks: the use of red-black trees instead of linked lists for the sorted queues in the kernel. Because a red-black tree maintains a $O(log(n))$ insert complexity, this would seem to be an attractive means to mitigate the Endpoint Queue Sorting and Replenishment Queue Sorting Attacks. As a result, we replaced the linked list implementation for kernel metadata processing in seL4-MCS with a red-black tree and repeated the same tests from §VI-C.

Indeed, in Fig. 10a and Fig. 10b, we can see that the red-black tree limits the Endpoint Queue Sorting Attack and Replenishment Queue Sorting Attack respectively to a logarithmic increase in HPI based on the number of attack threads. However, these data structures do not mitigate the attack completely, as an increase of up to 2,000 cycles and 3,000 cycles respectively is still demonstrated.

In contrast, we see from Fig. 10c that using a red-black tree still results in similar impacts for the Replenishment Wakeup Processing Attack. Just as for the linked-list version (Fig. 9c) we see the attack become effective at about 450 attackers and cause about 100,000 cycles of HPI per attacker above that point. If we investigate the differences between the linked list and red-black tree versions, we find that the red-black tree



(a) Short periods: $[3, 33]ms$.

(b) Moderate periods: $[10, 100]ms$.

(c) Long periods: $[50, 250]ms$.

Fig. 11: Sample schedulability graphs. All distributions uniformly distributed. Medium per-task utilizations in $[0.1, 0.4]$.

increases the impact of the attack by around 1,500 cycles. We posit this effect comes from the fact that while the red-black tree decreases insert complexity from $O(n)$ to $O(log(n))$, it also increases removal complexity from $O(1)$ to $O(log(n))$, but we leave a detailed investigation for future work.

E. Queue-per-priority Mitigation Analysis

We consider one final mitigation strategy for our Thundering Herd Attacks: the use of a data structure with separate queues per priority level, similar to the `plist` structure found in Linux [28]. A common implementation would be an array of queues, one for each priority level. seL4-MCS has 256 priority levels (0-255), so such a data structure would have 256 queues.

For our Endpoint Queue Sorting Attack, this data structure would completely eliminate the attack. This is because if there were a separate queue for each priority, there would be no need to sort the queues. With no queues to sort, insertion would be $O(1)$. The only cost is a small amount of additional memory.

For the Replenishment Queue Sorting Attack, a data structure with separate queues for each priority level would have no impact. This is because the queue being attacked is *not* sorted by priority, but rather by time of next replenishment. As a result, a data structure with separate queues per priority would merely move the sorting into the per-priority queues rather than eliminate it. Further, an attacker would still be able to create a large number of threads with the same priority to cause extensive sorting in these per-priority queues.

Finally, for the Replenishment Wakeup Processing Attack, this kind of data structure could provide benefits, but only with extensive kernel-wide modifications. In particular, it would be desirable to only process wakeups for processes of greater or equal priority to the currently running process. This would prevent wakeups for lower-priority processes from causing HPI and those lower-priority processes would not be run immediately anyway. While this is a promising idea, it requires extensive redesign of seL4-MCS, which we leave for future work, to check for pending wakeups on every context switch, including along the IPC hot path.

VII. Implications for System Provisioning

A promising mitigation to these attacks is to explicitly provision a system to be resilient to the HPI that attackers can induce using these attack techniques. Specifically, if a system is provisioned with sufficient slack time, it is resilient to attacker-induced HPI. Next we demonstrate through simple schedulability experiments the utilization loss associated with such provisioning. Systems that are not provisioned with such slack are vulnerable to Thundering Herd Attacks, which can potentially maliciously cause timing violations.

Experimental Design. For our schedulability experiments, we consider a simple sporadic task system scheduled with fixed-priority scheduling, as is implemented in seL4. In particular, we consider a task system as a system of n tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$. Each task $\tau_i = (C_i, T_i)$ is a recurring sequence of jobs, each of which has an execution time of C_i, which are released sporadically with a minimum separation of T_i time units. We assume implicit deadlines, and thus the deadline of each job is T_i time from its release. The utilization of τ_i is defined as $u_i = C_i/T_i$, and the utilization of the task system $U(\Gamma) = \sum_{\tau_i \in \Gamma} = u_i$.

We randomly generated task systems using commonly used task-system distributions. We considered all combinations of task-utilization and period distributions used in Brandenburg's dissertation [29] and codified in the associated Schedcat library [30]. This resulted in 54 unique task-utilization and period combinations. For each such combination, we varied $U(\Gamma) \in \{0.01, 0.02, \ldots, 1.0\}$, which resulted in 54 unique schedulability graphs. Three sample graphs from this larger study are depicted in Fig. 11.

We evaluated schedulability using the classic response-time analysis. We model the high-priority interference induced by our attack techniques as the highest-priority task in the system, τ_{HPI}. The period of τ_{HPI} is defined to be hyperperiod, or the least-common multiple of all task periods. This represents the possibility that an attacker can induce HPI at any time, but cannot necessarily trigger such attacks multiple times to trigger compounding effects. The execution time τ_{HPI} is chosen to reflect the amount of HPI that an attacker is assumed to be able to induce. Based on our previous results, we chose to evaluate $C_{\text{HPI}} \in \{0.975, 1.875, 7.5, 45\}ms$. These values are chosen from the FIFO Endpoint Flood Attack HPI with 500 and 1000 attackers ($0.975ms$, 1.875, resp.), and the Replenishment Wakeup Attack with 500 and 1000 attackers ($7.5ms$, $45ms$, resp.), as representative values to demonstrate the potential range of consequences of Thundering Herd Attacks.

An admission-control test that considers this interference derives this overhead from the number of threads in the system paired with the results in §VI.

Results. These results demonstrate that, for task systems with shorter periods and hence tighter timing constraints, the

HPI that these attacks can induce can significantly impact schedulability. Indeed, the short periods considered (uniformly distributed among $[3, 33]ms$) are less than the HPI from the Replenishment Wakeup Processing Attack ($45ms$), and therefore no such tasks can be guaranteed to meet their deadlines if such attacks are possible. Even if the attacker is assumed to only be able to spawn fewer threads and therefore induce less HPI, there can still be significant utilization loss.

In addition to enabling a proper admission-control test to mitigate the Thundering Herd Attacks, these results can also be interpreted as demonstrating what systems are *vulnerable* to Thundering Herd Attacks. Any task system that is deemed schedulable without considering HPI but is not schedulable with HPI is vulnerable to an attack that can trigger a deadline overrun. This is true even in budgeted systems where temporal interference from one task to another is controlled, as the HPI can be induced by low-priority tasks with minimal budgets.

These evaluations demonstrate the schedulability-related implications of the system-coordination dilemma in light of our Thundering Herd Attacks. We next discuss alternative means of mitigating these attacks and resolving this dilemma.

VIII. Discussion and Related Work

Synchronous IPC mechanisms have been studied as an attack vector [5] as synchronous IPC ties the execution of clients to a server's computation and introduces inter-client interference when execution is serialized through server threads. We have demonstrated in §V and §VI that the kernel mechanisms for maintaining the necessary IPC and execution metadata – which track the state of communication and properly schedule threads – can themselves become attack targets. A common characteristic of each attack is that kernel processing on this metadata is not constant-time, and can thus be targeted by attackers. These attacks are enabled by the non-preemptive nature of the kernel, made worse by multicore systems that prevent parallel kernel execution using a lock.

While we study attacks on seL4-MCS, the core challenges generalize to other systems with non-preemptive processing of IPC and budget-management data structures. For example, μ-kernels commonly use non-preemptive spin-locks to protect data-structures and disable interrupts during timer processing. Thus these attacks might be more broadly impactful.

In the rest of this section we discuss implications of these findings for μ-kernel design and highlight related work.

A. Implications for μ-Kernel Design

§VII discusses how to integrate the measured overheads from the various attacks into schedulability analysis. This test can be integrated into the system's admission controller. Unfortunately, we demonstrate that doing so can cause significant utilization loss. Here we qualitatively suggest and assess a number of alternate kernel-design options.

Track metadata with $O(log(n))$ data-structures. IPC endpoint wait queues are tracked with linked lists and are sorted in seL4-MCS. Replacing these with balanced binary trees will asymptotically decrease the cost of adding threads to the queue, which would comparably decrease the amount of non-preemptible execution, as shown in §VI-D. This does *not* defeat the attacks, but does lessen their impact. This effect has been observed when using $O(log(n))$ data structures for other potentially contended kernel data structures such as timers [31] and futexes [32].

Logarithmic structures can also be used to track replenishments, which would similarly decrease the asymptotic overheads for replenishment attacks. Unfortunately, this is not a clear benefit. Though adding threads to the replenishment queue on budget depletion would benefit, the overhead for processing replenishments during seL4's timer will *increase*, as shown in §VI-D. Each of the n replenishments that require processing will increase from constant to logarithmic overhead. Our results suggest this may be a reasonable trade off.

Track wait queues with queue-per-priority data structures. Instead of using a balanced tree, a constant-time structure common in fixed-priority scheduling implementations could be used for IPC endpoint wait queues. This is an array with one entry per priority, each containing a list of waiting threads. A bitmap (or nested bitmaps) are used to track which priorities have waiting threads. The constant-time overhead of this approach would defeat the attacks on kernel IPC endpoint queues. The primary cost of this approach would be a minor increase in IPC endpoint memory consumption commensurate with the number of potential priorities.

Expanded use of preemption points. Preemption points are explicit closures that capture a kernel in-progress operation and allow interrupts to be processed, later continuing kernel execution from the closure. If they could be applied to bound the cost of the wait-queue and replenishment operations, they could be an important part of a solution. Preemption points add significant complexity to the system and require kernel operations to be iteratively computed. For example, preemption points are used to enable capability revocation to revoke a limited number of resources, enable preemptions, and later resume revocation from where it had previously left off. Unfortunately, iterating through a wait queue, or through a queue of replenishments, does not fall into the traditional type of logic that preemption points are designed for, as each iteration does not remove work from the computation to be done after a preemption point. Such an iteration could not be resumed after a preemption point as the preemption implies that the queue's structure could have been updated by intervening operations (*e.g.,* removing the thread from the wait queue referenced by the current iterator). Preemption point logic would need to increase in complexity to handle wait-queue operations and budget depletion.

Preemption points also *cannot* be added to the replenishment processing. Preemption points rely on being able to resume a thread that continues kernel processing from where it had previously left off. Despite preemption point's superficial applicability to these problems, unfortunately it won't help with all attacks, particularly with replenishment processing in timer-interrupt context.

Partial processing of wakeups in interrupts. TimerShield [31] represents a potential solution to the replenishment-processing problem. The key insight is that if replenishments can be tracked per-priority, then all replenishments for time t do not need to be processed at that time. Instead, at each scheduling decision, the replenishment queue can be consulted and processed if the highest-priority threads requiring replenishments have the same or higher priority than the highest-priority thread in the run queue. Though this approach is appealing, it complicates the system, requiring the design of the replenishment logic to be considered more broadly within the system as a whole.

B. Other μ-Kernel Designs

Although seL4 is based on the L4 μ-kernel heritage, it is a unique μ-kernel with functional verification as its primary goal. Below we discuss other μ-kernel designs.

Fiasco and Nova. Fiasco [8] takes a different view on priority and budget management during IPC. In Fiasco, servers execute using the budget of the client requesting their service (as in seL4-MCS). However, in both Fiasco and Nova [9] the server *inherits* the highest priority of any client transitively waiting on the service. This can add overhead to the scheduling path as the dependencies of the highest-priority thread (and its dependencies' dependencies, *etc.*) are traversed to find the server to execute. Additionally, Fiasco adds vCPU budgets [33], which require depletion and replenishment processing.

Generally, these kernels execute *preemptively*, which prevents Thundering Herd attacks on kernel structures from causing global interference. However, both use spin-locks to protect kernel structures, which selectively disable interrupts while processing data structures, and both disable interrupts for interrupt execution. We have not assessed if Fiasco or Nova exhibit similar attacks on their non-preemptive access to data-structures. The lessons of this research should inform the assessment of their vulnerability to such attacks.

Thread migration in Composite. Thread-migration-based IPC [34]–[36] is a different mechanism for synchronous coordination between client and server. A server process is the target of the IPC, not a server thread. IPC from a client triggers execution in the server that proceeds within the same scheduler context as in the client process. It is called "thread migration" because the same thread simply *continues* execution in the server, though spatial isolation is maintained by splitting client/server execution across separate stacks and register contents. Since the same schedulable client thread executes in the server, the same scheduler abstractions such as priority and budget are maintained. This structure imposes a few requirements: (1) servers are concurrent by default and thus require synchronized access to shared data structures, and (2) server stacks must be allocated upon IPC to the server as the first action within the server's computation. Both of these challenges can be addressed by efficient, predictable mechanisms for stacks and mutual exclusion for both fixed [37], and dynamic [38] sets of threads. Thread migration can avoid blocking semantics in the kernel; instead, schedulers can be implemented in user-level processes [39], [40]. Even where budgets are tracked in the kernel [41], replenishments are exported from the kernel to user-level schedulers.

As thread migration enables the policies for contention, scheduling, and budget management to be extracted from the kernel, all kernel operations can be constant-time as demonstrated by [42]. However, within the scheduling processes that maintain wait queues, budgets, and priorities, it is possible that Thundering Herd Attacks could be impactful. Though interrupts are never disabled for user-level processing, critical sections within the scheduler might comparably be attacked, delaying necessary scheduling decisions.

C. Static Partitioning Hypervisors

Another potential solution to these attacks is to use a static partitioning hypervisor, such as Jailhouse [43] to statically isolate untrusted parts of the system from one another. This is perhaps a reasonable solution for coarse-grained isolation of untrusted and uncooperating components, provided that there are sufficient hardware resources to be dedicated to individual partitions. However, μ-kernels such as seL4 enable more fine-grained isolation, enabling the principle of least privilege to be employed in system design, such as in Patina [18]. For example, many trusted, but not trustworthy, components may communicate and collaborate and therefore need to be co-located within a single static partition. But even a trusted component may be compromised, and a component-based architecture, and resource sharing and strong isolation enabled by a trustworthy μ-kernel such as seL4 limit the damage an attacker can do. A static partitioning hypervisor can therefore help ameliorate some of these concerns, but does not fundamentally solve the system-coordination dilemma.

Summary. While we've discussed several potential solutions to Thundering Herd Attacks in seL4, they all present trade-offs. We've also analyzed different systems and shown that Thundering Herd Attacks are a more general concern.

IX. Conclusion

In this work we have explored attacks on temporal isolation in μ-kernels focusing on the synchronous IPC and budget-replenishment mechanisms. We first discussed long-standing issues with synchronous IPC-based shared servers and characterized the impact of these issues on the seL4 μ-kernel. The usual mitigations for these attacks are priority-sorted endpoint queues and replenishment policies. Unfortunately, we then showed that these very mechanisms enable novel, powerful attacks. We implemented and evaluated these attacks on endpoint-queue sorting and replenishment queues in seL4 with MCS extensions and demonstrated their linear impact to the number of attacker threads, with up to 100,000 cycles of interference per thread. We presented an approach to account for these attacks in a schedulability analysis and illustrated associated utilization loss. Lastly, we discussed mitigation approaches and their limitations.

References

[1] B. Sprunt, L. Sha, and J. Lehoczky, “Aperiodic task scheduling for hard-real-time systems,” *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[2] S. Vestal, “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance,” in *28th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, 2007, pp. 239–243.

[3] A. Burns and R. Davis, “Mixed criticality systems - a review,” Department of Computer Science, University of York, Tech. Rep., 2013.

[4] J. Liedtke, “Improving IPC by kernel design,” in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 1993, pp. 175–188.

[5] J. S. Shapiro, “Vulnerabilities in synchronous IPC designs,” in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2003, pp. 251–262.

[6] S. Ruocco, “A real-time programmer’s tour of general-purpose L4 microkernels,” *EURASIP Journal on Embedded Systems*, vol. 2008, 2008.

[7] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, “Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time,” in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, pp. 26:1–26:16.

[8] U. Steinberg, J. Wolter, and H. Härtig, “Fast component interaction for real-time systems,” in *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE Computer Society, 2005, pp. 89–97.

[9] U. Steinberg and B. Kauer, “Nova: A microhypervisor-based secure virtualization architecture,” in *Proceedings of the 5th European conference on Computer systems (EuroSys)*. ACM, 2010, pp. 209–222.

[10] L. M. Ruane, “Process synchronization in the UTS kernel,” *Computing systems*, vol. 3, no. 3, pp. 387–421, 1990.

[11] J. Liedtke, “On μ-kernel construction,” in *15th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 1995, pp. 237–250.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: Formal verification of an OS kernel,” in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2009, pp. 207–220.

[13] K. Elphinstone and G. Heiser, “From L3 to seL4 what have we learnt in 20 years of L4 microkernels?” in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013, pp. 133–150.

[14] P. K. Gadepalli, G. Peach, G. Parmer, J. Espy, and Z. Day, “Chaos: A system for criticality-aware, multi-core coordination,” in *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 77–89.

[15] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, “Timing analysis of a protected operating system kernel,” in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2011, pp. 339–348.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Transactions on computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[17] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, “CAmkES: A component model for secure microkernel-based embedded systems,” *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007.

[18] S. Jero, J. Furgala, R. Pan, P. K. Gadepalli, A. Clifford, B. Ye, R. Khazan, B. C. Ward, G. Parmer, and R. Skowyra, “Practical principle of least privilege for secure embedded systems,” in *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 1–13.

[19] CrowdStrike, Inc. (2021) 2021 global threat report. [Online]. Available: https://go.crowdstrike.com/rs/281-OBQ-266/images/Report2021GTR.pdf

[20] M. Crosignani, M. Macchiavelli, and A. F. Silva, “Pirates without borders: The propagation of cyberattacks through firms’ supply chains,” *FRB of New York Staff Report*, no. 937, 2021.

[21] S. Liu, N. Guan, D. Ji, W. Liu, X. Liu, and W. Yi, “Leaking your engine speed by spectrum analysis of real-time scheduling sequences,” *Journal of Systems Architecture*, vol. 97, pp. 455–466, 2019.

[22] C. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash, “A novel side-channel in real-time schedulers,” in *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 90–102.

[23] S. Liu and W. Yi, “Task parameters analysis in schedule-based timing side-channel attack,” *IEEE Access*, vol. 8, pp. 157 103–157 115, 2020.

[24] J. Liedtke, N. Islam, and T. Jaeger, “Preventing denial-of-service attacks on a µ-kernel for WebOSes,” in *Proceedings of The Sixth Workshop on Hot Topics in Operating Systems (HotOS)*. IEEE Computer Society, 1997, pp. 73–79.

[25] G. Heiser. (2019) How to (and how not to) use seL4 IPC. [Online]. Available: https://microkerneldude.org/2019/03/07/how-to-and-how-not-to-use-sel4-ipc/

[26] J. Chen, G. Nelissen, W. Huang, M. Yang, B. B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. C. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen, “Many suspensions, many problems: A review of self-suspending tasks in real-time systems,” *Real-Time Systems*, vol. 55, no. 1, pp. 144–207, 2019.

[27] Trustworthy Systems Team, Data61. (2020) seL4 reference manual: Version 12.0.0. [Online]. Available: https://sel4.systems/Info/Docs/seL4-manual-12.0.0.pdf

[28] (2022) plist.h. [Online]. Available: https://github.com/torvalds/linux/blob/master/include/linux/plist.h

[29] B. B. Brandenburg, “Scheduling and locking in multiprocessor real-time operating systems,” Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[30] ——. (2022) SchedCAT: The schedulability test collection and toolkit. [Online]. Available: https://github.com/brandenburg/schedcat

[31] P. Patel, M. Vanga, and B. B. Brandenburg, “TimerShield: Protecting high-priority tasks from low-priority timer interference,” in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 2017, pp. 3–12.

[32] A. Zuepke and R. Kaiser, “Deterministic futexes: Addressing WCET and bounded interference concerns,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 65–76.

[33] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, “Flattening hierarchical scheduling,” in *Proceedings of the 12th International Conference on Embedded Software (EMSOFT)*. ACM, 2012, pp. 93–102.

[34] B. Ford and J. Lepreau, “Evolving Mach 3.0 to a migrating thread model,” in *Proceedings of the Winter 1994 USENIX Technical Conference*. USENIX Association, 1994, pp. 97–114.

[35] E. Gabber, C. Small, J. L. Bruno, J. C. Brustoloni, and A. Silberschatz, “Pebble: A component-based operating system for embedded applications,” in *USENIX Workshop on Embedded Systems*. USENIX Association, 1999, pp. 55–65.

[36] G. Parmer, “The case for thread migration: Predictable IPC in a customizable and reliable OS,” in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2010, p. 91.

[37] Q. Wang, J. Song, and G. Parmer, “Stack management for hard real-time computation in a component-based OS,” in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2011, pp. 78–89.

[38] Q. Wang, J. Song, G. Parmer, G. Venkataramani, and A. Sweeney, “Increasing memory utilization with transient memory scheduling,” in *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2012, pp. 248–259.

[39] G. Parmer and R. West, “Predictable interrupt management and scheduling in the Composite component-based system,” in *Proceedings of the 29th IEEE International Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2008, pp. 232–243.

[40] P. K. Gadepalli, R. Pan, and G. Parmer, “Slite: OS support for near zero-cost, configurable scheduling,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 160–173.

[41] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, “Temporal capabilities: Access control for time,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 2017, pp. 56–67.

[42] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, “SPeCK: A kernel for scalable predictability,” in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 2015, pp. 121–132.

[43] J. Kiszka. (2022) Jailhouse hypervisor. [Online]. Available: https://github.com/siemens/jailhouse