PolyRhythm: Adaptive Tuning of a Multi-Channel Attack Template for Timing Interference

Ao Li*, Marion Sudvarg*, Han Liu, Zhiyuan Yu, Chris Gill, Ning Zhang
{ao, msudvarg, h.liu1, yu.zhiyuan, cdgill, zhang.ning}@wustl.edu
Department of Computer Science & Engineering, Washington University in St. Louis

***Abstract*—As cyber-physical systems have become increasingly complex, rising computational demand has led to the ubiquitous use of multicore processors in embedded environments. Size, Weight, Power, and Cost (SWaP-C) constraints have pushed more processes onto shared platforms, including real-time tasks with deadline requirements. To prevent temporal interference among tasks running concurrently or in parallel in such systems, many operating systems provide priority-based scheduling and enforce processor reservations based on Worst-Case Execution Time (WCET) estimates. However, shared resources (both *architectural* components and data structures within the *operating system*) provide channels through which these constraints can be broken. Prior work has demonstrated that malicious execution by one or more processes can cause significant delays, leading to potential deadline misses in victim tasks.**

In this paper, we introduce *PolyRhythm*, a three-phase attack template that combines primitives across multiple architectural and kernel-based channels: (1) it uses an offline genetic algorithm to tune attack parameters based on the target hardware and OS platform; then (2) it performs an online search for regions of the attack parameter space where contention is most likely; and finally (3) it runs the attack primitives, using online reinforcement learning to adapt to dynamic execution patterns in the victim task. On a representative platform (Raspberry Pi 3B) PolyRhythm outperforms prior work, achieving significantly more slowdown. As we show for several hardware/software platforms, PolyRhythm also allows us to characterize the extent to which interference can occur; this helps to inform better estimates of execution times and overheads, towards preventing deadline misses in real-time systems.

***Index Terms*—real-time systems, interference channels, attack templates, parameter tuning, contention regions**

I. Introduction

Real-time and embedded systems are now designed for increasingly complex cyber-physical applications with high computational demands, e.g., object recognition [1] and localization [2] in autonomous vehicles [3], [4]. Such systems are often constrained in Size, Weight, Power, and Cost (SWaP-C), motivating the need to run these computationally-intensive applications concurrently on a single computer. These constraints, combined with the challenges of heat dissipation and the need to maintain stable processor frequencies [5], have driven a move toward multi-core platforms. Recent approaches to partitioned scheduling of tasks on multiple processors have yielded greater utilization bounds *in theory* [6]–[8]; yet in practice, contention for hardware and kernel resources shared between cores can cause temporal interference, inflating worst-case execution times (WCETs) beyond typical estimates [9].

Prior work has demonstrated several architectural channels in multicore systems over which concurrent access can cause delays in task execution times. The main memory bus, controller, and DRAM row buffers [10], as well as the cache, its writeback buffers, and miss-status-holding-registers [11], can all be targeted by malicious processes to interfere with victim task execution. The translation lookaside buffer (TLB) adds another source of execution time unpredictability, with misses incurring additional overhead to traverse page tables during memory access [12]. Besides these architecture-based attacks, shared kernel data structures requiring synchronized access serve as additional channels for interference [13]. Attacks on these can increase context switching times and even induce priority inversion in kernel execution pathways [14], increasing CPU utilization and delaying or blocking task activation. Additionally, even in kernels that impose temporal budgets on isolated processes (e.g., through the use of bandwidth servers [15] or scheduling contexts [16]), frequent context switching can lead to greater overheads and increased contention for shared kernel data structures.

Existing attacks typically focus on individual vectors, e.g., cache contention in multicore systems. These require significant effort to tune manually, though machine learning approaches also have proven effective for increasing interference [17]. Additionally, adversaries aiming to force a system to violate its timing guarantees might use multiple vectors. To understand the extent to which an attacker can cause interference by *automating* a multi-channel timing interference attack, we introduce and evaluate PolyRhythm[1], a three-phase attack template that combines primitives across multiple architectural and kernel-based channels.

PolyRhythm is *multi-channel*, adopting a strong threat model in which an attacker (constrained by core affinity, priority, and/or CPU utilization by the kernel) uses combinations of attack primitives over architectural and operating system channels to interfere with the timing of a *victim* process. We consider hardware contention in main memory, cache, and the TLB; and contention for operating system resources such

* denotes equal contribution. This work is supported in part by the US National Science Foundation under grants CNS-1837519, CNS-1916926, CNS-2038995, CNS-2154930, CNS-2229427, CSR-1814739, and CNS-17653503; the National Aeronautics and Space Administration under grant 80NSSC21K1741; and by the Army Research Office under contract W911NF-20-1-0141. Many thanks to Sanjoy Baruah for his insights and advice through the entire process.

[1]The PolyRhythm tool, associated source code, and instructions for use are all available at https://github.com/WUSTL-CSPL/PolyRhythm.

2022 IEEE Real-Time Systems Symposium (RTSS) | 978-1-6654-5346-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/RTSS55097.2022.00028

2576-3172/22/$31.00 ©2022 IEEE
DOI 10.1109/RTSS55097.2022.00028

as network and I/O queues, as well as syscalls for which a portion of the kernel execution is synchronized (e.g., locked by a mutex).

PolyRhythm is *templated*, providing a general framework of contention attacks on architectural and operating system resources that decouples attack design from concrete implementation on a given platform. It is also *automated*, exploiting more effective and efficient combinations of parameter values than could be found manually. We evaluate to what extent such attacks can inflate task execution or response times, potentially causing victim tasks to miss deadlines. This paper makes the following contributions:

- It surveys architectural and operating system channels that can be exploited by one or more attacking processes to induce timing interference on a victim task, identifying a broader unified attack surface than has been explored in any single example of prior work.
- It introduces *PolyRhythm*, a three-phase, template-based technique for constructing multi-channel attacks. The first phase uses a genetic algorithm to optimize the template parameters of each attack primitive according to characteristics of the target hardware and operating system platform. In the second phase, PolyRhythm performs online search to identify *contention regions*, which are regions of the attack parameter space where contention is most likely (e.g., targeted cache eviction sets).
- It shows how by observing workload-specific execution and resource usage patterns at run-time, an attacker can leverage a deep deterministic policy gradient to adapt uniquely to online variations in the victim's behavior. In its third phase, PolyRhythm executes attacks using this learning-based technique to switch dynamically between active primitives, thus continuing to induce optimized interference.
- It uses these techniques to characterize – for several hardware/software platforms – the extent to which interference can occur. This allows us to make recommendations about both how to avoid this interference where possible and how to account for it where necessary (e.g., by informing WCET estimates), to prevent deadline misses in real-time systems.

II. Background and Related Work

The attack primitives used in PolyRhythm are based on interference channels that have been explored in prior work. In this section we identify both architectural and operating system channels that have been shown effective, including in prior template based attacks. In Section VI we explain how PolyRhythm goes beyond prior work to use such channels even more effectively than has been achieved to date.

A. Architectural Channels

Shared Memory Resources: On multi-core platforms, there are typically fewer memory nodes than processor cores; therefore concurrent requests to main memory must be brokered across common **interconnect buses**. These can take the form of a single shared bus [18], or may be distributed across a tree-like structure that multiplexes requests at multiple stages [19]. More complex memory routing, via packet switching networks, is employed in network-on-chip designs [20]. No matter the granularity at which interconnects are shared, competing access can be exploited to induce delays: in [21], it was shown that bus contention can significantly impact task WCETs in real-time systems; and in the Android OS, an application targeting the memory bus was demonstrated to interfere with victim application performance [22]. Direct memory access (DMA) by devices must traverse these same buses (as well as additional interconnects for the device controllers), which can induce further interference [23].

Even after a request is delivered over the interconnect, subsequent memory latency may vary significantly. Data already in DRAM **row buffers** can be retrieved several times faster than on a buffer miss or conflict [24]. Row buffer latency and **memory controller** and bus contention have been exploited successfully by Denial of Service (DoS) attacks [10]. Other memory architectures have been exploited as well: multi-core and multi-processor systems that employ Non-Uniform Memory Access (NUMA) use multiple memory nodes with separate controllers; these induce even greater variability in access latencies, which may hurt performance in unexpected ways [25]. Though NUMA-based architectures are becoming more common, we consider only embedded platforms with uniform memory access and defer integration of NUMA attack primitives into our attack template to future work.

Cache-Based Attacks: The cache can reduce memory request latency by keeping recently-accessed data closer to the CPU. A cache hit obviates the delays associated with main memory access; however, a miss may result in orders of magnitude greater latency in the worst case. Managing the associated unpredictability, especially where task switches disrupt cache working sets, or in shared cache levels in multi-core systems, is challenging in real-time systems. Techniques such as cache coloring [26] partition the cache among separate tasks or cores, isolating task performance from interference caused by other processes' access patterns. Despite careful partitioning, however, some architectures incorporate shared hardware over which cache contention can still occur. For example, in non-blocking caches, miss-status-holding-registers (**MSHRs**) are used to queue memory requests to handle outstanding cache misses [27]; when the registers are exhausted, the cache reverts to blocking behavior, which can cause substantial delays. Writeback caches buffer data to be written back to main memory, attempting to perform the writes only during quiescent periods. Because the **writeback buffer** is shared among all users of the cache, it too has been exploited successfully as a channel for interference in multi-core systems [11], [28].

Unpredictability of the TLB: As another type of cache, the translation lookaside buffer (TLB) also introduces timing unpredictability: the overhead induced by virtual address translation is dramatically worse on a TLB miss, especially in virtualization environments that use second-level address

translation [29]. However, many multi-core architectures provide a dedicated TLB per core, and many operating system kernels perform a TLB shootdown on each context switch [30], making opportunities for interference harder to identify. Nonetheless, the implications of this flushing are significant: when a process is context-switched onto the processor, its initial memory access will automatically result in a TLB miss. As we demonstrate in this paper, context-switch attacks (where an attacker forces a high rate of context switching) can cause delays in victim processes, as the rate of TLB misses drastically increases. Some architectures, however, allow the kernel to tag TLB entries with process-specific identifiers, meaning that the TLB need not be flushed on every context switch. TLB coloring techniques (which are conceptually similar to cache coloring) have been proposed to reduce interference [12]; however, as the number of concurrent processes increases, each dedicated partition must decrease in size. Context-aware preloading techniques may decrease the potential for interference under these circumstances [30]. In this paper, we constrain our evaluation to systems for which the TLB is flushed on each context switch; consideration of TLB coloring is deferred to future work.

Recent work also has demonstrated effective attacks on the TLB as an *information* side-channel [31], [32]. While we focus on attack channels for *timing* interference, these are inextricably linked [33]: information side-channels are typically exploited by measuring timing inconsistencies; similarly, the presence of timing inconsistencies results in unpredictable execution and possible interference and response time delays. Mechanisms that address either will surely be relevant to both.

B. Operating System Channels

I/O and Network Attacks: As block and network device speeds have increased, in-kernel **request queues** have become a bottleneck in multi-core and many-core systems, especially where (like in the Linux block layer) a single queue is shared among cores [34]. Cascade attacks, which exhaust one type of hardware resource to cause a cascade of performance degradation over other channels, have proven effective in causing significant **network** and **disk I/O** interference amongst co-resident VMs in virtualized environments [35]. Denial of service (DoS) attacks on these same resources have been launched by malicious VMs in cloud environments [36]. In those same environments, resource-freeing attacks [37] may interfere with a victim VM's resource usage to free other resources for an attacker VM's benefit. In microkernels where I/O access is brokered by userspace components, specially-crafted attacks have demonstrated that a "Thundering Herd" of low-priority tasks can induce priority inversion and significantly delay the delivery of I/O replies to a high-priority task [13].

Kernel Data Structures: Shared data structures over which control flows maintain consistent state among cores present another class of interference channels. Accessing these structures can cause delays that scale with the number of elements or even induce priority inversion (i.e., when kernel execution on behalf of low-priority threads obtains a lock on the resource). Lock scalability to multiple cores is an area of significant concern [38], as contention for locks that do not properly scale can cause significant blocking delays [39]. In systems that use thread migration for IPC, the set of available stacks may be a limited resource, and contention can induce priority inversion [14]. Some modern microkernels, such as SPeCK [40] have attempted to address these scalability issues. Others, however, are still vulnerable to attacks that leverage contention on run queues, signaling and IPC endpoints, budget replenishment queues, and timer queues [13].

C. Tunable Attack Templates

Exploiting architectural and kernel vulnerabilities has traditionally required detailed knowledge of the target. However, focus has recently shifted to attacks that generalize to a broader range of platforms, allowing exploits to be constructed without prior knowledge of victim behavior or system specifications [17], [41], [42]. These so-called template attacks combine one or more tunable execution primitives that target a given resource. In [41], template attacks are presented to exploit the cache as an information side channel. An initial **profiling** phase adjusts attack parameters (e.g., access patterns) for the target platform before carrying out the attack to extract information. "Slow and Steady" [17] template attacks are extended to *timing interference* channels. The authors propose a template for attacking the memory hierarchy and explore multiple automated search techniques to tune associated parameters. Fuzzing, which is the process of finding vulnerabilities by repeatedly testing code with modified inputs, is conceptually similar. In [43], the authors present an approach for fuzzing via **reinforcement learning** (RL) to optimize an attack.

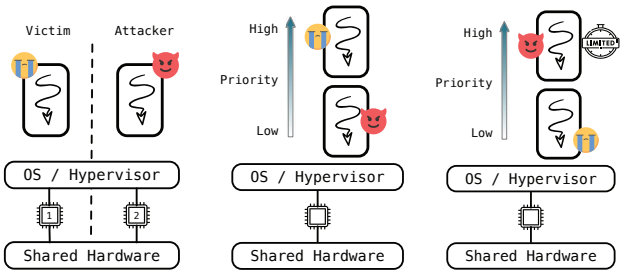
Inspired by the template attacks and learning-based approaches of prior work, in this paper we propose a dynamic attack template that is tuned in multiple phases, including by an offline genetic algorithm (GA) and then through online RL. GA [44] evolves parameter selections over a set of candidate solutions, attempting to determine those most fit for a given task. We will leverage GA for primitive tuning, since the search space is large. For strategy tuning, we will leverage deep deterministic policy gradient (DDPG) [45], [46], an RL approach that inherits the advantages of both value-based [47] and policy-based [48] methods: it is a model-free, off-policy algorithm that uses an actor-critic framework. Unlike the Deep Q Network [49], which handles only discrete states and cannot efficiently solve large action spaces, DDPG is tractable for high-dimensional, *continuous* action spaces.

III. System and Threat Model

In this paper, we target implicit-deadline sporadic task sets under fixed-priority preemptive scheduling. Task execution is fully partitioned among cores as in [8], i.e., all instances (jobs) of a given task are executed on the same set of processors, and all processors are identical. In Section VIII, we evaluate in the context of both *individual jobs* (profiling single runs of a benchmark workload) and *sporadic tasks*

(measuring execution times in real-world applications with recurrent execution). We assign fixed priorities to threads, using rate-monotonic (RM) scheduling for sporadic tasks.

We consider task interference under three increasingly strong threat models, illustrated in Figure 1. Common to all three, we assume that the attacker and victim processes are isolated into separate address spaces, with CPU core affinity and memory access partitioning enforced by the operating system or hypervisor, though the attacker and victim may indirectly share the resources detailed in Sections IV and V. Attacker processes execute without privileges, and so cannot (1) modify a victim's address space or access a victim's memory, (2) change its own, or a victim's, CPU affinity, or (3) modify task priorities, including its own.



(a) Separate Cores (b) Lower-Priority (c) Utilization Constraint

Fig. 1: Threat models considered in this work.

(a) Separate Cores: As shown in Figure 1(a), the attacker and victim processes run on separate cores. The attacker may run at an arbitrary priority relative to that of its victim, but cannot directly preempt the victim's execution. The attacker may attempt to interfere with the victim's timing through access to shared resources and kernel functionality. Nonetheless, it cannot modify its own, or the victim's, priority.

(b) Same Core, Lower-Priority Attacker: In this scenario, shown in Figure 1(b), the attacker and victim processes are on the same processor core. The attacker has a lower priority than any victim and thus can never preempt execution by a victim. However, the attacker may increase the victim's execution time by carefully targeting shared hardware resources; it also may attempt to block the victim through system calls and other mechanisms that can induce priority inversion.

(c) Same Core, Utilization-Constrained Attacker: In the third scenario we consider, illustrated in Figure 1(c), the attacker and victim processes again reside on the same processor core. This time, however, the attacker may be assigned a higher priority than the victim (e.g., because the corresponding task releases jobs at a higher rate). The attacker's CPU utilization, however, is constrained, e.g. via a budget server [50]. Given the victim's estimated WCET (appropriately padded for context switching time), the attacker's execution time is constrained to a limit that guarantees system schedulability under the hyperbolic bound [51]. Under this threat model, the attacker may try to extend the victim's WCET via the same interference channels as under the previous threat model. It also may attempt to induce frequent context switching, which can cause interference as described in Section V-C.

IV. Architecture Attack Channels

PolyRhythm supplies an *attack template* that combines multiple attack primitives over both *hardware architectural* and *operating system kernel* channels, providing a general framework that can be tuned to maximize resource contention. A summary is given in Table 1 in Section VI. In this section, we outline the targeted architectural channels, describe the design and implementation of the corresponding attack primitives, and characterize their associated tunable parameters.

A. Memory

Even with an operating system isolating memory between processes, shared hardware introduces channels for contention. These include the last-level *cache*; the common *interconnect bus* and *memory controller* over which multiple cores compete for access; and the DRAM *row buffers* for which memory latency depends on whether they are holding the requested data. PolyRhythm provides three attack primitives that target these resources: **cache**, **memory bandwidth**, and **row-buffer**.

Cache: PolyRhythm can force last-level cache eviction, similarly to the attack in [41]. This attack primitive initializes by allocating an integer array with SIZE elements, then setting the value of each element to its index. It continuously iterates over the array, reading or writing elements according to the policy defined by the ACCESS parameter. Its iteration step is the number of cache lines specified by STRIDE, which can be tuned so that subsequent cache misses require row switching at the DRAM level, leading to inefficient processing of requests [28]. In our online attack, we use cache profiling to narrow the scope of the attack to *eviction sets* containing victim data; this is described in more detail in Section VI-B. Depending on the hardware architecture, this primitive may also cause writeback buffer contention or blocking due to exhaustion of the MSHRs [11]. While we do not identify these scenarios explicitly, our tuning phases may induce such behavior as a byproduct of optimizing the attack.

Memory Bandwidth: Using a primitive derived from the memory contention stress test provided by **stress-ng** [52], PolyRhythm can target the memory controller and interconnect bus. The attack creates a file, then maps it into memory *twice* (at two different addresses), using a mapping size of SIZE. It then launches a number of threads specified by the NTHREADS parameter; these threads each continuously run a procedure that writes to then reads from the memory, depending on the ACCESS parameter. Carefully-placed memory barriers and cache line flushes ensure that the threads generate a high volume of requests to the memory controller.

Row-Buffer: We target the DRAM row buffer using an attack primitive derived from those in [10]. It dynamically allocates two integer arrays of size SIZE, initializes each with random values, then continually iterates over the arrays, copying and

modifying values between the two. Iteration is performed either sequentially (with access separated by **STRIDE** indices) or using a randomly-generated sequence of indices (which remains consistent over each iteration), according to the parameter **PATTERN**. In doing so, we attempt to target each bank in main memory, flushing the corresponding row buffer.

B. Translation Lookaside Buffer (TLB)

In this work, we consider three TLB levels which may be present in different architectures: (1) a per-core L1 TLB that is typically shot down by the kernel on every context switch; (2) a per-core L2 TLB that provides a shared cache among multiple processes on the same core;[2] and (3) a last-level TLB common to all cores.[3] We consider two ways to induce TLB contention: **context-switching** and **page eviction** attacks.

Context-Switching: A high-priority attacker, running on the same core as a victim task, induces a high rate of context switches (despite having its utilization constrained, as in threat model **(c)** in Section III). This results in a high rate of L1 TLB misses during victim execution, which can cause significant delays. More details on this attack are given in Section V-C.

Page Eviction: To attack the L2 and last-level shared TLB, PolyRhythm includes a primitive that continuously, in a loop, maps a number of memory pages specified by the **PAGES** parameter and zeroes the entire mapped region. It then iterates over each page, marking the page read-only, copying the contents of the page into a page-sized buffer on the stack, then unmapping the page. This has the effect of evicting each page from the TLB; given enough pages, the victim will experience a high rate of TLB misses, forcing expensive hardware page table traversals for each virtual memory access.

V. Operating System Attack Channels

In this section we outline operating system channels our attack template can target, which augment the architectural channels in Section IV, again describing the design and implementation of corresponding primitives and tunable parameters.

A. I/O Queues

As network and block devices have become faster, contention for in-kernel I/O request queues may increase when they are shared among multiple cores [34]. Where multiple tasks compete for these resources, synchronization may cause priority inversion and inflate blocking times (especially in ordered queues without constant-time insertion). Additionally, as queues grow longer, the latency between enqueueing and handling a request induces delays. PolyRhythm's attack template includes primitives to target **network** and **block devices**.

Network: If a network controller is shared among multiple cores, a kernel driver will enqueue packet transfer operations, typically using only a single queue per address family. In many operating systems, these queues are unaware of requesting task priorities, which may introduce priority inversion [53]. Concurrent packet transmission from these queues also may subsequently contend for lookups in a shared routing table.

To attack these points of contention, PolyRhythm provides a primitive that produces a flood of UDP traffic to the local loopback address, continuously sending packets containing a random string of length **SIZE**. Because the attack may be highly dependent on the ports and socket address families in use by a specific victim application, these are selected online in PolyRhythm's second phase, as described in Section VI-B. By targeting the local loopback only, this attack avoids the network hardware, and instead focuses on associated kernel data structures (which are still heavily used by many middleware systems [54]–[56], even when intercomponent communication remains on the same host). We defer a study of network hardware contention, e.g. for its ability to induce delays in distributed real-time systems, to future work.

Block Devices: I/O schedulers merge and reorder queued operations to try to reduce latency and service requests fairly, but this comes with tradeoffs: as the queue grows, overhead induced by scheduler operations and blocking times induced by synchronization over multiple cores may cause delays. Additionally, as the number of requested transfers increases, so too will the interrupts generated by the corresponding device. On platforms where interrupt routines for a given device all run on a single core, this can cause the performance of tasks running on that core to degrade significantly.

PolyRhythm includes a primitive that produces a large number of block I/O operations. It initializes by generating a random string of length **SIZE**. It creates and opens a new file with a randomly-generated name, declares an access pattern by calling `posix_fadvise()` with advice specified by the **ADV** parameter, and then continuously writes to the file sequentially or in random locations according to **PATTERN**. For a sequential access pattern, file access is separated by an offset of **STRIDE** bytes. Each write is followed by a call to `fsync()` to force the operating system's page cache to write back the dirty page. We tune these parameters to maximize interference by, as much as possible, preventing file readahead [57] and bypassing the kernel's page cache to force a large number of individual requests to the block device.

B. Kernel Data Structures

Operating system kernels also maintain shared data structures that must remain consistent among cores; increasing contention for these structures may cause delays and priority inversion, e.g., many kernels that provide a filesystem abstraction layer (such as the Linux VFS) use several locks to protect shared state [39]. Other interference channels include shared run queues, signaling and IPC endpoints, budget replenishment queues, and timer queues [13]. PolyRhythm exploits these with the following attack primitives: **file system** and **fork bomb**.

File System: PolyRhythm provides a primitive that targets shared kernel data structures used by the filesystem abstrac-

[2]These are present, for example, on the ARM Cortex-A72 in the Raspberry Pi 4: https://developer.arm.com/documentation/100095/0002/memory-management-unit/tlb-organization/l2-tlb

[3]The ARMv9 Cortex-A510 has an L2 TLB common to all cores: https://developer.arm.com/Processors/Cortex-A510. This architecture is not yet generally available at time of writing, so we defer evaluation to future work.

tion layer. This attack is characterized by an **ACCESS** mode parameter: *create* continually creates files; *move* creates a file then continually renames it; *delete* runs in a loop, creating then deleting files; *reopen* creates **NFILES** files and then continually (re-)opens each one. Each version of the attack uses randomly-generated filenames. Depending on the target filesystem, contention can manifest differently: in Linux, for example, these attacks may induce kernel-level contention for acquisition of a global spin lock over the list of superblocks.

Fork Bomb: The PolyRhythm template includes a fork bomb attack that continuously forks new processes (each with its own address space) until **NPROCS** processes have been created. This is exponential: each forked process runs the same fork bomb routine. Such an attack rapidly consumes hardware resources. Even with kernel-enforced resource constraints, a rapid burst of thread creation may cause contention over the kernel's run queue, task or thread control block allocators, etc. After the threads are active, their presence may cause scheduling delays, as the run and wait queues grow in size [13].

C. Context-Switching

PolyRhythm provides a ***Context-Switching*** primitive that continuously suspends the attacking thread for a number of milliseconds defined by the **DURATION** parameter. If the attacker is scheduled at a higher priority than a victim on the same core, this forces rapid context switching between processes. Even if the attacker's CPU utilization is constrained (as in threat model **(c)** of Section III) it can still: (1) decrease available CPU utilization by increasing total context switching time beyond what any WCET padding accommodates; (2) increase contention for shared kernel data structures; (3) cause incorrect in-kernel accounting of its own execution time, allowing it to overrun its budget and thus deny service to the victim; and (4) increase task execution times as various hardware caches (e.g., the CPU cache and TLB) are refilled.

VI. Adaptive Multi-Channel Attack Template

PolyRhythm provides a three-phase dynamic attack template that can combine primitives (described in Sections IV and V) targeting multiple architectural and operating system channels as illustrated in Figure 2. Each phase addresses a specific challenge associated with keeping the attack both *general* (targeting multiple platforms) and *effective* (maximizing induced temporal interference):

(1) Platform-Dependent Attack Parameters: Each of PolyRhythm's primitives has associated parameters, summarized in Table 1, for which optimal values depend strongly on the target platform: even with reverse engineering, the efficacy of the attack may remain unpredictable. To address this challenge, in its first phase PolyRhythm executes an offline genetic algorithm [44] to search the parameter space, tuning the values to maximize delays over each individual resource.

(2) Unpredictable Resource Locality: Resource allocation is often dynamic, both in hardware (e.g., placement of cache lines) and the operating system (e.g., virtual-to-physical memory mappings). This results in unpredictable *locality* of the victim's resource footprint. To address that challenge, PolyRhythm's second phase uses online search for regions of the parameter space where contention is more likely to learn *where* to target its attacks.

(3) Dynamic Execution Patterns: PolyRhythm does not assume explicit knowledge of victim task execution patterns and resource usage. Its targets may have unknown or dynamic control flows, which require online *adaptation* of attack strategies. To address this challenge, PolyRhythm's third phase uses a trained reinforcement learning model to adjust selection of running primitives to reactively optimize interference.

A. Platform-Dependent Parameter Tuning

Genetic algorithms are adaptive optimization heuristics inspired by natural selection [44] which have proven effective for such applications as resource scheduling [58] and hyperparameter tuning [59]. A genetic algorithm (GA) can search efficiently over a large domain [60], making it well suited for optimization over the multidimensional space described by PolyRhythm's template parameters.

To optimize these parameters for a given platform, characterized by both its architecture and OS, PolyRhythm begins by running a GA on the target system. We denote each template attack primitive as A_i, with associated parameters $a_{i,j}$. A parameterized attack primitive is scored according to its *interference potential* V_i:

$$V_i = \sum_{k=1}^{m} w_k \times \frac{\phi_{k,i}}{\phi_k^0} \tag{1}$$

Here, ϕ_k counts a *performance event* indicative of temporal interference, and is normalized by ϕ_k^0, the observed count without interference. The profiled events for each primitive are listed in Table 1. On platforms that do not support a given event, execution time is measured instead. Each event is assigned a weight, w_k, according to its relative ability to predict delay. On low-power platforms with high variation [61]–[63] (e.g., a Raspberry Pi), we assign weights according to the stability of the event: $w_k = \bar{\phi}_k^0/\sigma_k^0$, with $\bar{\phi}_k^0$ the mean and σ_k^0 the standard deviation of the event count without interference. On more stable platforms (e.g., an Intel Nuc with hyperthreading disabled), we instead weight according to the relative impact of the primitive on each event: $w_k = \phi_k^{max}/\phi_k^{min}$ – respectively, the maximum and minimum observed event counts induced by different parameterized runs of the given channel.

The GA tunes each primitive independently, finding the parameters that maximize its interference potential over representative victim workloads from the **stress-ng** and **CortexSuite** benchmarks [52]. For every parameterized run of A_i, it measures (e.g., with `perf`) each associated event count $\phi_{k,i}$ from the concurrently-running victim. We initialize the population with an initial hand-picked set of parameter values, similarly to the methodology in [11] (e.g., the **SIZE** and **STRIDE** may be initialized according to the size of the last-level cache and

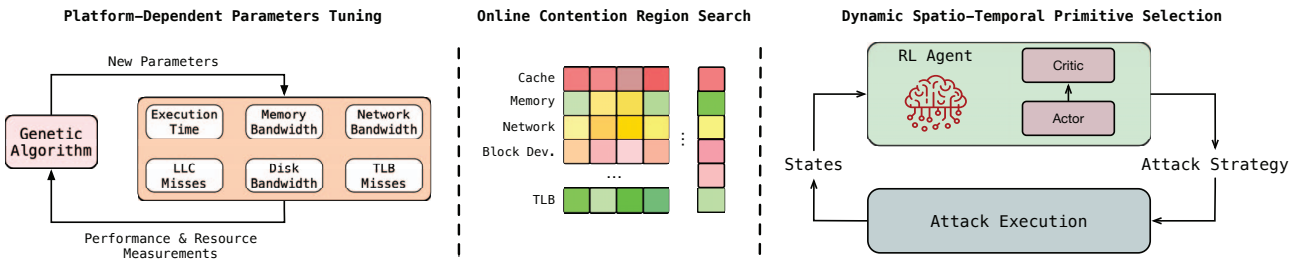


Fig. 2: PolyRhythm Pipeline

Primitive	Description	Parameters	Perf Events	Phase 2
Cache Attack	Force cache evictions	SIZE, STRIDE, ACCESS	LLC Misses	✓
Memory Bandwidth	High volume of requests to main memory	SIZE, NTHREADS, ACCESS	LLC Misses,Bus Cycles	✓
DRAM Row Buffer	Swaps and modifies values between two arrays	SIZE, STRIDE, PATTERN	LLC Misses,Bus Cycles	✓
TLB Page Eviction	Maps, reads, and unmaps memory pages	PAGES	TLB Misses	–
Network I/O	Floods local loopback with UDP packets	SIZE	CPU Cycles	✓
Block Device I/O	High rate of write operations to a block device	SIZE, STRIDE, PATTERN, ADV	CPU Cycles	–
File System	Contention over the file system abstraction layer	NFILES, ACCESS	CPU Cycles	–
Fork Bomb	Rapidly forks a large number of processes	NPROCS	CPU Cycles	–
Context Switching	Forces rapid context switching of victim	DURATION	CPU Cycles,Sched_Switch	–

TABLE 1: PolyRhythm Attack Template

cache line). The GA then runs until convergence, or until a maximum number of generations have passed. For each generation, each member of the population is traced independently to determine its interference potential, and then those having the greatest potential are selected as candidate parents. In the crossover step, each member of the subsequent generation is assigned two parents by random selection from the candidates, which are weighted exponentially by interference potential. Each child's parameter values are selected at random from both parents, with no bias toward either.

For each generation g_l, a subsequent mutation step introduces stochastic noise to avoid converging on a local (rather than global) maximum. For every member of the population, each parameter is mutated with probability p_l by adding a term sampled from a normal distribution about 0 with a constant standard deviation normalized to each parameter. The choice of p_l is significant: too low, and there may be insufficient noise to escape local maxima; too high, and the excessive randomization might prevent convergence. To address these issues, we use an approach based on momentum mutation [64], updating p_l according to:

$$p_l = \mu \cdot p_{l-1} + \frac{\alpha}{|v_l - v_{l-1}|}, \quad v_0 = 0, \quad p_0 = 0.01 \qquad (2)$$

Here, $\mu = 0.9$ is the scaling factor for the momentum's running sum, $\alpha = 0.01$ is the acceleration factor, and v_l denotes the maximum normalized interference potential among all members of generation g_l. By adding to the mutation probability when a generation fails to adapt (i.e., when v_l and v_{l-1} are close in value), the algorithm can escape local maxima, with the increase capped at 0.5. By maintaining a moving average over prior updates, the probability is smoothed, making it less susceptible to outliers in the population.

When the GA terminates, the member with the greatest interference potential from the last generation is selected, and its parameter values are assigned to the associated primitive. Despite being specific to the platform on which they were trained, these values may nevertheless generalize to other platforms, and thus potentially could be used as inputs to a PolyRhythm attack without running the offline parameter search phase, or as initial inputs to that phase on that platform.

B. Online Contention Region Search

Even with parameters tuned for the target platform, variations in victim resource *locality* and random factors in runtime allocation make for unpredictable attack efficacy. This is highlighted by the results in Section VIII-B, which illustrates that even with tuning, some attack channels are ineffective for certain victim tasks, and by Section VIII-C, which illustrates that the delay induced by a tuned cache attack can be highly inconsistent across instances of a victim task. To address this, PolyRhythm's second phase is an online search for targeted contention regions that overlap with victim resource locality, relying on similar search techniques to those in [65], in which an attack is constructed that targets small *eviction sets* in cache. For many real-time applications, predictability of control flow and memory footprint often imply that, once found, these contention regions remain stable across task instances.

As discussed in Section III, no elevated privileges are conferred to the attacker – PolyRhythm is presumed unable to trace its victim's resource usage via direct hardware or kernel channels. Instead, it estimates locality using online tracking of its own execution over different regions within the search space, with semantics that depend on the individual attack channel. The **Phase 2** column of Table 1 indicates those channels for which online search has been implemented.

Memory-Based Channels: For the cache attack primitive, given a value of SIZE found by the GA, PolyRhythm partitions the allocated memory of that size into several equal regions.

It then sequentially launches and profiles the execution time of the attack primitive over each region while the victim runs concurrently. Longer execution times indicate greater contention over that region, suggesting that the region may share underlying resources with the victim, based on its allocation patterns. It runs with a fixed number of iterations; in each iteration, it prunes the regions with the least contention, then allocates new regions so that **SIZE** remains consistent. The memory bandwidth and row buffer primitives execute over two dynamically mapped or allocated regions of memory; at each iteration, the region is remapped or reallocated. At the end of the search, the pair of addresses indicating the greatest contention are used for the remainder of the attack. After performing this iterative pruning technique, we are left with contention regions that form a more precise attack surface over which we may induce a higher degree of interference with victim response times.

Network I/O: Greater contention may be achieved when targeting the network ports and address families used by the victim task. PolyRhythm attempts to find the port and address family combination indicative of the greatest contention. It first enumerates the list of open ports (on Linux, it does so by parsing the **`tcp`** and **`udp`** files in **`/proc/net`**, which do not require special privileges to read), then runs the network I/O attack across the cartesian product of those ports and the **`{UNIX,INET,INET6,ALG}`** address families. The **`(PORT,AF)`** pair with the slowest packet rate is assumed to be the most effective for inducing delays in the victim task.

C. Dynamic Spatio-Temporal Primitive Selection

Armed with tuned parameters for its target hardware and OS platform, as well as profiled sets describing an attack surface that contends with the victim's resource footprint, the third piece of the puzzle is optimization of the spatio-temporal selection of attack primitives. Victim control flows and resource usage patterns are dynamic; PolyRhythm must adaptively decide *which* attack primitive to run at *each time step*. To this end, PolyRhythm executes its online attack using an adaptive strategy based on reinforcement learning (RL). In particular, given an initial system state s_0, PolyRhythm will perform a sequence of actions a_i that optimize at each time step t a reward function r of the system state s_t that indicates its current interference with victim task execution:

$$\underset{a_1,a_2,\ldots,a_t}{\arg\max}\ r(s_t \mid s_0, a_1, a_2, \ldots, a_t) \tag{3}$$

We represent this as a Markov decision process [66]: at each time step t a *policy* selects an *action* a_t representing one of the PolyRhythm template's tuned attack primitives; the attack is run for the duration of the time step, which advances the process to the next step. This action returns a reward $r(s_t, a_t)$ characterized by the *slowdown* observed for the attack primitive, which is indicative of the interference induced on the victim.

To find the policy that maximizes the cumulative reward over the entire process, we use the deep deterministic policy gradient (DDPG) framework [45], [46], as it does not require a priori knowledge of the environment's dynamics, and it is efficient over high-dimensional continuous action spaces. DDPG is an actor-critic framework that composes two neural networks: an actor network π learns an action policy according to closed-loop feedback from a critic network Q, which estimates the cumulative reward given the actor's observed policy. Because training is resource-intensive, if run concurrently with the attacker and victim, it can affect the observed behavior. To minimize the extent to which results are skewed by the training, we use an offline hardware-in-the-loop approach detailed in Section VIII.

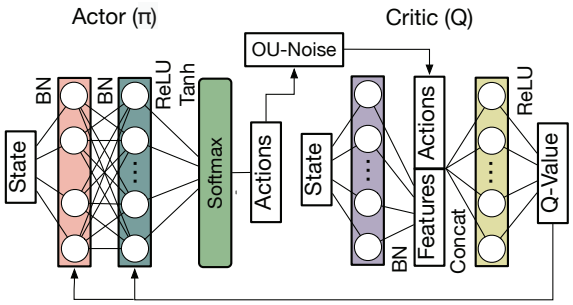


Fig. 3: Actor-Critic Network Architecture

The design of the actor (π) and critic (Q) networks is illustrated in Figure 3. In both networks, we use three fully-connected layers (a layer encoding the state, then two hidden layers), each followed by batch normalization; all hidden layers have 40 nodes. In π the final output vector has the **`tanh`** and **`ReLU`** activation functions applied, followed by a **`softmax`**. Ornstein-Uhlenbeck noise is added to the resulting values, improving the efficiency and stability of the search [67]. When training, the state and result action vectors are passed to the Q network, which concatenates the action vector with the normalized output of its second layer, using this as input to the third layer. The **`ReLU`** activation function is applied at the end to obtain the cumulative reward score, which is used as feedback for the actor. PolyRhythm implements its DDPG framework with PyTorch [68] and uses the same hyperparameter values as FIRM [69]. By using a fine-grained time step (discussed in Section VIII) and training using a replay buffer with 10^5 entries to cache previous action/state pairs, PolyRhythm is able to learn an attack action policy targeting even a highly dynamic victim.

In its online attack, at each time step PolyRhythm runs the highest-scored primitive on all targeted cores. The design of our model supports a further expansion of this configuration in which PolyRhythm may select different primitives for different cores. However, we defer a more comprehensive study of combinations over the design space to future work.

VII. Implementation

PolyRhythm's implementation is composed of two parts: a program, written primarily in C, that takes a set of template parameters and runs the attack; and a set of scripts, written

primary in Python, that can launch the attack to tune its parameters and train the DDPG model to perform spatio-temporal primitive selection.

As described in Section VI-A, a Genetic Algorithm tunes attack channel parameters for a target hardware and operating system platform. The GA is implemented as a Python script that independently tunes each primitive. On a system with n processors, it concurrently launches $n-1$ instances of the first PolyRhythm binary (specifying the current primitive), then separately launches an instance of the representative victim task, profiling it with **`perf`**. When the victim job completes, the script terminates the PolyRhythm instances, then parses the performance information to inform its next iteration. Once the GA has run to completion, the script outputs the tuned parameters, then proceeds to the next primitive.

The PolyRhythm attack program has two modes of execution. The first runs a specified attack primitive according to specified parameters. The second mode runs one or more PolyRhythm instances concurrently with the reinforcement-learning model described in Section VI-C, synchronizing via a shared memory region. Tuned parameters for all primitives are provided via a text file. The RL script establishes a periodic timer, and at each interval, it writes an attack template into shared memory from a list generated by the DDPG model. The PolyRhythm instances run the specified attack, at each iteration checking if the selected primitive has changed and writing back execution time statistics to shared memory, which are used as feedback by the DDPG. Either PolyRhythm attack mode can additionally be instructed to run the online contention region search described in Section VI-B. If enabled, the first several iterations of the attack primitive are used to search for a more precise attack surface over which it may induce a higher degree of interference with victim response times. For those attack channels that do not implement the online search, the behavior remains unchanged. When running concurrently with the RL model, if a primitive switch occurs before the search completes, state is saved so that the search resumes next time it is invoked.

We provide a separate launcher program to launch the PolyRhythm attack instances concurrently with the DDPG script. Each child process can be pinned to a core and assigned a fixed priority under the **`SCHED_RR`** scheduling class or a runtime, deadline, and period under the **`SCHED_DEADLINE`** priority class according to optional command-line parameters. If either real-time scheduling class is used, the launcher assigns itself scheduling attributes that allow it to preempt its children, signals them by writing to a pipe, then executes a shell (allowing the user to inspect or terminate any of the child processes).

VIII. Evaluation

PolyRhythm presents a dynamic attack template for interference on multicore systems. We evaluate it in the context of recent work, comparing its effectiveness to the **`BwWrite`** attack [11]. We also test the impact of each of PolyRhythm's three phases, attacking synthetic workloads and a real-world SLAM system. We target five hardware platforms – the Raspberry Pi 2B, 3B, and 4B, an Nvidia Jetson Nano, and the Intel Nuc 8 – detailed in Table 2. The Nuc 8 kernel has been patched to enable **`PREEMPT_RT`**. We disabled frequency throttling on all devices, except where otherwise noted.

Platform	Abbr	Arch	RAM	Cores	Linux	HT
Raspberry Pi 2B	RPi2	Cortex-A7	1GB	4	5.10	
Raspberry Pi 3B	RPi3	Cortex-A53	1GB	4	5.10	
Raspberry Pi 4B	RPi4	Cortex-A72	4GB	4	5.10	
Nvidia Jetson Nano	Nano	Cortex-A57	4GB	4	4.9	
Intel Nuc 8	Nuc	Skylake	16GB	4	5.13	✓

HT: Hyper-Threading.

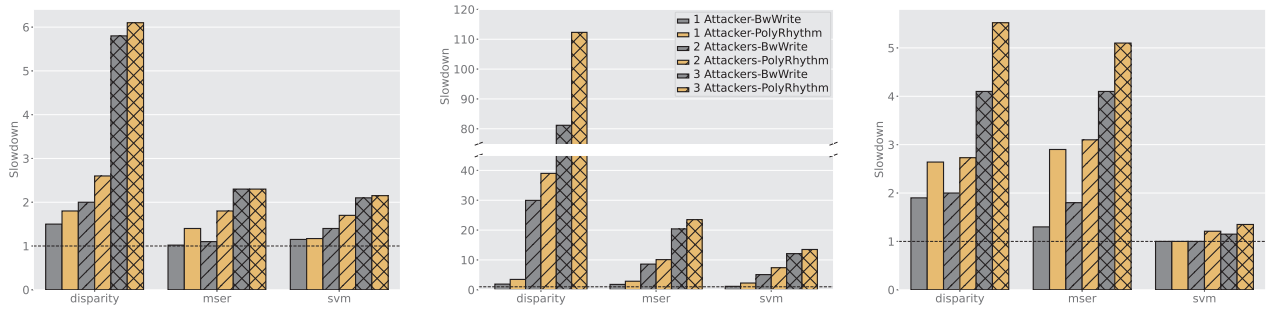
TABLE 2: Evaluation Platforms

A. Phase 1: Parameters Tuned by the Genetic Algorithm

Comparison to BwWrite: To gauge the genetic algorithm (GA) that PolyRhythm uses to tune attack template parameters, we evaluate its cache attack primitive in the context of the **`BwWrite`** attack from [11]. **`BwWrite`** similarly creates contention in shared cache channels on multicore systems, but its parameters are not tuned: it iterates using a cache line stride length over a region of memory equal to the size of the last-level cache. We launch concurrent attacker instances on 1, 2, and 3 cores, with a victim task pinned to its own core, according to threat model **(a)** in Section III. We separately consider three victim workloads from the **CortexSuite** benchmark suite [70]: Disparity Map (**`disparity`**), which computes depth information for objects jointly represented in a pair of stereographic images; Maximally Stable Regions (**`mser`**), which performs blob detection in images; and Support Vector Machines (**`svm`**), which separates data into two categories. We measure the relative victim slowdown induced by each attack, taking the mean over 100 samples for each scenario.

In Figures 4(a) and (b), we compare the slowdowns induced by PolyRhythm to those for **`BwWrite`** [11] on a Raspberry Pi 2B and 3B respectively. Absent an identified kernel on which to reproduce their approach, we use an up-to-date kernel and compare our results to those reported in their paper. To remain consistent with the previous work, we left frequency throttling *enabled* for these experiments. For completeness, we also compared the attacks on a Raspberry Pi 4B (also with frequency throttling enabled). As the previous work did not target this platform, all results illustrated in Figure 4(c) were our own measurements. We observed that by using the GA to tune its parameters for each target platform, PolyRhythm's cache attack primitive typically induces more interference than **`BwWrite`** over the tested benchmarks. We observed that the attacks were especially effective on the RPi3; this may be due to MSHR contention, which is particularly mismatched in speed to the rest of the memory pipeline on the RPi3.

Additionally, we observed that, when tuning both attacker and victim parameters, PolyRhythm was able to induce a maximum slowdown of $715\times$ in these victim workloads on the RPi3 with frequency throttling enabled; this is significantly more than the maximum value reported for **`BwWrite`** ($346\times$)



(a) RPi2 (b) RPi3 (c) RPi4

Fig. 4: Relative Slowdowns Induced by **`BwWrite`** and PolyRhythm Attacks on **CortexSuite** Benchmarks

in [11]. Even for the same victim workload evaluated in [11], PolyRhythm achieves a maximum $412\times$ slowdown. This suggests that hand-picked parameters based on known hardware characteristics (e.g., cache size and cache line length) may not perform as well as learned parameters; other hardware factors besides the line and cache size affect the efficacy of the attack. Additionally, it suggests that while parameters selected for one platform could be ported another platform, tuning the parameters for a specific target is highly effective. Compared to previous work, PolyRhythm better captures the possible interference among attacker and victim because it automates the state space search over a target application and platform.

Comparison of Attack Channels: To quantify the sensitivity of each attack channel, we individually tuned (with the GA) and evaluated several of PolyRhythm's attack primitives on each target platform, using **stress-ng** workloads [52] as victim tasks. We target **`stress-cache`** with the cache attack, **`stress-stream`** with the memory bandwidth and row buffer attacks, **`stress-udp`** with the network I/O attack, **`stress-io`** with the block device I/O attack, and **`stress-brk`** with the TLB attack. As before, we run attackers in parallel with the victim according to threat model **(a)**, and collect relative slowdown metrics for each test. The maximum values observed for the RPi4, Nuc, and Nano are illustrated in Figure 5.

We observe that the interference caused by primitives targeting architectural channels vary significantly by platform. For example, though both the RPi4 and Nano are ARM-based and each has a shared L2 cache, the cache attack is far more effective on the RPi4. Even primitives that target kernel-level contention are highly dependent on hardware. We found that on the RPi3 and Nuc, PolyRhythm was effective at causing network contention; however, the tuned primitives had little impact on the RPi2, RPi4, and Nano, despite sharing similar operating system kernels. Further, block device I/O contention was shown to be dependent on the storage medium: it was effective on the Raspberry Pi devices and Jetson Nano, which all use SD cards; the Nuc uses a faster solid state drive, and was therefore only minimally affected.

As illustrated in Figure 5(b), Hyper-Threading opens a channel for TLB interference. The bars labeled "TLB (no HT)" indicate interference with Hyper-Threading disabled; relative slowdowns are only barely greater than 1, even with three parallel attackers. However, with Hyper-Threading enabled – labeled "TLB (HT)" – slowdowns exceeded $2.7\times$, due to multiple logical processors executing concurrently on a single physical core, enabling contention over its TLB.

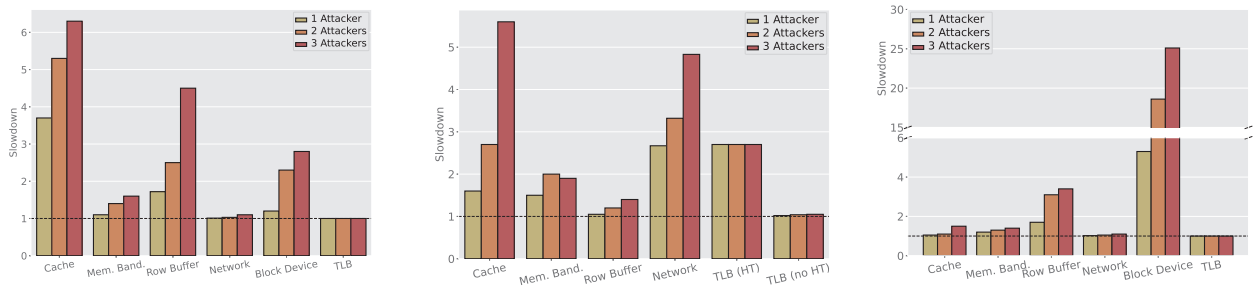
We also implemented the fork bomb and file system attack primitives. These both were able to cause severe contention issues, preventing us from even measuring the interference: the file system attack caused the whole system to become unresponsive, while the forkbomb attack was terminated by the operating system after generating too many child processes.

To evaluate the stability of the GA, we tuned the parameters for the cache primitive an additional 10 times on the RPi3. Each time, the optimal **STRIDE** was found to be 1 cache line and writing was determined as the optimal **ACCESS** mode. Only the **SIZE** of the allocated region varied: the mean was found to be 845.6kB, with a standard deviation of 38.2kB, less than 5% of the mean.

B. Phase 2: Online Contention Region Search

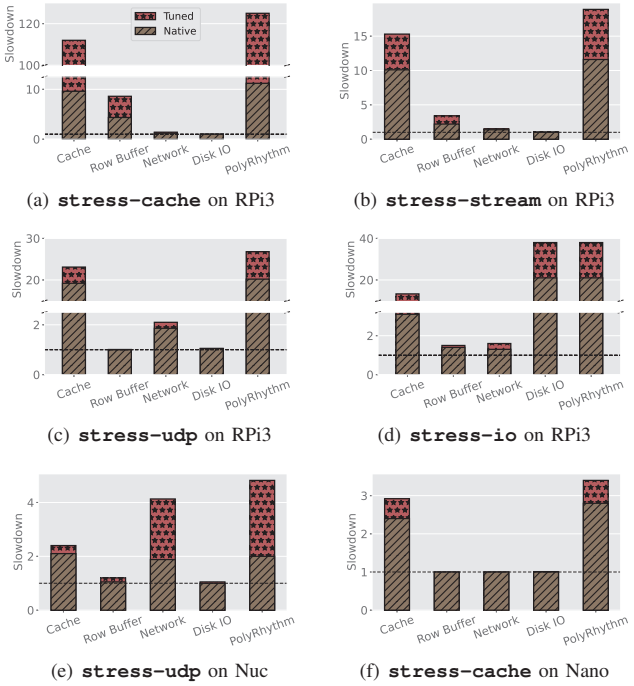
PolyRhythm's second phase searches online for targeted contention regions that may overlap with resources used by the victim. To gauge its effectiveness against different victim workloads, we evaluate four primitives against a variety of victim workloads. We run three PolyRhythm instances concurrently with a victim according to threat model **(a)**. We again find the mean relative slowdown over 100 iterations, which are plotted in Figure 6. Values labeled "Tuned" represent the slowdown observed with victim workloads also tuned for the target platform; "Native" are for the victim workloads run with default parameters. For a given workload, on a given platform, we select the *most effective* attack (i.e., the one with the greatest mean slowdown) and implement it in PolyRhythm with an initial online search for contention regions. On the Nuc, the attackers and victim are all launched on different *physical* cores. The resulting slowdowns are illustrated by the bars labeled "PolyRhythm" in the same figure.

On the platforms tested, GA-tuned cache attacks are often the most effective. By searching online for contention regions (which are likely eviction sets in this case), even more interference can be induced; on the RPi3, we found that targeting



(a) RPi4 (b) Nuc (c) Nano

Fig. 5: Maximum Relative Slowdowns Induced by PolyRhythm's Tuned Primitives



(a) **`stress-cache`** on RPi3 (b) **`stress-stream`** on RPi3

(c) **`stress-udp`** on RPi3 (d) **`stress-io`** on RPi3

(e) **`stress-udp`** on Nuc (f) **`stress-cache`** on Nano

Fig. 6: Comparing Effectiveness of Online Contention Region Search

contention regions increased the slowdown by 2.5×. Even though the cache attack was most effective against a network workload on the RPi3, we found that a network attack is most effective against the same workload on the Nuc. We observed that PolyRhythm was able to find a contention region over the domain of address families; targeting this forced the associated socket domain to repeatedly trigger error handling routines in the kernel's network driver, which caused more severe slowdown.

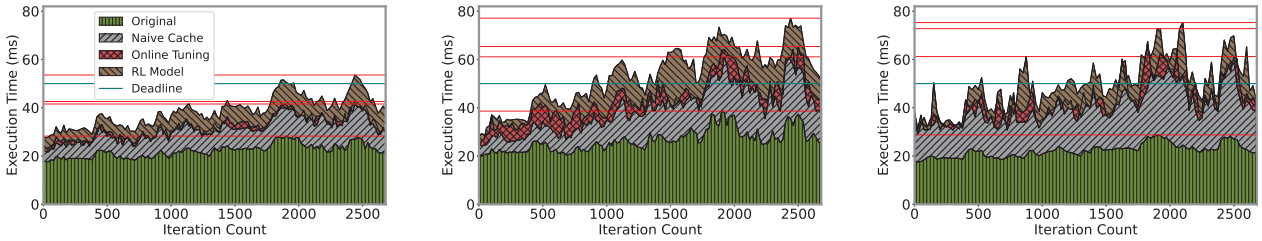
C. Phase 3: Dynamic Spatio-Temporal Primitive Selection

The synthetic workloads provided by **stress-ng** have highly regular resource usage patterns; these are therefore not well suited to evaluate the efficacy of PolyRhythm's third phase, which uses reinforcement learning to adapt to dynamic victims. Instead, we evaluate a full three-phase implementation of PolyRhythm, targeting the ORB-SLAM robotic system [71], [72] running over the EuRoC MAV dataset [73], which includes test videos taken by drones in real-world environments.

We run ORB-SLAM on the Nuc with a 50ms event loop in the context of both threat models **(b)** and **(c)** of Section III. For threat model **(b)**, ORB-SLAM runs at priority 30 concurrently with instances of PolyRhythm running at priority 20 and pinned to each logical core. We separately test with Hyper-Threading disabled and enabled, launching respectively 4 and 8 PolyRhythm instances. For **(c)**, we disabled Hyper-Threading and run ORB-SLAM at priority 20 concurrently with 4 higher-priority instances of PolyRhythm, each pinned to a distinct core, and each having its utilization constrained to the minimum idle percentage of that CPU when ORB-SLAM runs in isolation. Assigned budgets are 36%, 36%, 16%, and 55%; these are enforced with CBS under the **`SCHED_DEADLINE`** scheduling class, using a period and deadline of 50ms to match the ORB-SLAM event loop. Because the **`SCHED_DEADLINE`** class is prioritized over **`SCHED_RR`**, this guarantees that the attack threads preempt the victim when their budget replenishes. Both scenarios may also capture aspects of threat model **(a)**, as ORB-SLAM might not occupy all cores for the entirety of its execution.

We train the DDPG model via hardware-in-the-loop. A host PC with a 12-core AMD Ryzen 9 3900X and 128GB of RAM remotely launches PolyRhythm and the intended victim task on the target platform. At each time step, the remote machine runs the attack primitives (action steps) on the target. The PolyRhythm processes on the target monitor their state during action execution, sending feedback to the PC. Once the RL model has been trained offline, it is included in PolyRhythm's online attack, which can then run independently of the remote machine. We use a 5ms time step, which is granular enough to capture distinct behavioral patterns within ORB-SLAM's subtasks, but remains long enough for each individual primitive to run several iterations. To evaluate the attack, the trained model runs at the highest priority (priority 90 for threat model **(b)**, 5ms period and 1ms deadline for **(c)**) to guarantee that it preempts other execution.

ORB-SLAM main loop response times are illustrated in Figure 7; shown are the mean values over every 10 loops for execution across the entire dataset. We observe that, without interference (labeled "Original"), ORB-SLAM never



(a) Threat model **(b)**, Hyper-Threading disabled. (b) Threat model **(b)**, Hyper-Threading enabled. (c) Threat model **(c)**, Hyper-Threading disabled.

Fig. 7: DDPG-Based Dynamic Attack Against ORB-SLAM.

misses a deadline; the response time of its main loop remains under the 50ms period. However, by interfering with ORB-SLAM sufficiently to cause its response times to exceed 50ms, PolyRhythm is able to induce overruns.

Under threat model **(b)** with Hyper-Threading disabled, all three phases are necessary to make ORB-SLAM miss its deadlines: PolyRhythm demonstrated an average delay of $1.8\times$ and induced a worst-case measured response time of 58.6ms (compared to 31.8ms in the uncontended case). Additionally, the DDPG model for primitive selection, when combined with the online contention region search, produces more delay than the naïve cache attack 97.5% of the time. In comparison, with Hyper-Threading enabled, the naïve cache attack alone can induce deadline misses. With all three phases, ORB-SLAM's worst-case measured response time is pushed to 120.0ms (compared to 47.1ms in the uncontended case), with an average delay of $2.03\times$. Here, the DDPG model produces more delay than the naïve cache attack 98.3% of the time. While the greater delays may be due in part to the additional instances of PolyRhythm, the longer response times of ORB-SLAM even in the uncontended case suggest that the global scheduler might not be aware of the mapping of logical cores to physical cores, resulting in suboptimal load balancing.

Under threat model **(c)** without Hyper-Threading, PolyRhythm induces an average delay of $2.3\times$ compared to the uncontended case, and pushes ORB-SLAM's worst-case measured response time to 155.4ms from 32.5ms without interference. Here the DDPG model produces more delay than the naïve cache attack 71.8% of the time. This suggests that such attacks may be highly effective under this threat model, since bandwidth reservations for the higher priority attacker remain fixed, even as the victim's demand for utilization increases with execution time interference.

IX. Detectability and Defense

PolyRhythm or similar attacks may be effective at causing missed deadlines in a real-time system, despite attempts to detect it. In its attack phase, PolyRhythm uses all available CPU time; in a real-time system where this execution is limited by priority or server-based bandwidth constraints, execution patterns may be similar to a task that runs for the extent of its analyzed WCET. Because PolyRhythm's resource usage does not require any special privileges, its patterns may be difficult to immediately distinguish from a non-malicious workload. Once its execution profile becomes indicative of targeted interference (e.g., identifying a high rate of cache misses), the target workload may have already missed deadlines; as illustrated in Figure 7, PolyRhythm was able to quickly cause overruns in ORB-SLAM's main loop, which may cause collisions in autonomous systems [74].

A potential defense is to measure task execution times when running concurrently with PolyRhythm, improving WCET estimates by accounting for targeted interference. Still, mitigating multi-core interference is a long-standing challenge. A traditional approach is to use resource isolation [75], [76] to remove the attack surface. However, as we demonstrate, several architectural and operating system channels expose resource contention surfaces; it is therefore necessary to vertically integrate isolation mechanisms across multiple system layers. A more dynamic approach that remains robust uses adaptive algorithms [77]–[79] or elastic scheduling [80]–[82], where the fail-safe or operational-safe mechanisms can adjust execution state in response to deadline misses.

X. Conclusions and Future Work

In this paper, we have presented PolyRhythm, a framework for automated, multi-channel, templated resource contention attacks. By abstracting a template of known resource contention primitives, PolyRhythm can automatically tune their associated parameters to a target platform and its runtime environment, finding contention regions based on dynamic placement of victim resources. It then searches for the optimal spatio-temporal interference strategy using reinforcement learning. Our evaluation shows both the *feasibility* and *efficacy* of this approach. By fully open-sourcing this AI-aided tool for interference measurement, we hope to complement existing methods to better characterize target workload/platform susceptibility to adversarial interference, informing WCET estimates, resulting in more secure and reliable real-time systems. In future work, we intend to conduct a more comprehensive study of combinations over the design space and target platforms, including applications in modern, complex safety-critical and mixed-criticality systems.

REFERENCES

[1] G. A. Elliott, K. Yang, and J. H. Anderson, "Supporting real-time computer vision workloads using openvx on multicore+gpu platforms," in *2015 IEEE Real-Time Systems Symposium*, 2015, pp. 273–284.

[2] B. Williams, G. Klein, and I. Reid, "Real-time slam relocalisation," in *2007 IEEE 11th International Conference on Computer Vision*, 2007, pp. 1–8.

[3] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013, pp. 31–40.

[4] R. Bonatti, Z. Yanfu, S. Choudhury, W. Wang, and S. Scherer, "Autonomous drone cinematographer: Using artistic principles to create smooth, safe, occlusion-free trajectories for aerial filming," in *Proceedings of International Symposium on Experimental Robotics (ISER)*, 2018, pp. 119–129.

[5] R. Rao and S. Vrudhula, "Performance optimal processor throttling under thermal constraints," in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 257–266. [Online]. Available: https://doi.org/10.1145/1289881.1289925

[6] S. K. Baruah and N. W. Fisher, "The partitioned dynamic-priority scheduling of sporadic task systems," *Real-Time Syst.*, vol. 36, no. 3, p. 199–226, aug 2007. [Online]. Available: https://doi.org/10.1007/s11241-007-9022-5

[7] S. Baruah, "Partitioned edf scheduling: A closer look," *Real-Time Syst.*, vol. 49, no. 6, p. 715–729, nov 2013. [Online]. Available: https://doi.org/10.1007/s11241-013-9186-0

[8] J. Chen, "Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2016, pp. 251–261. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ECRTS.2016.26

[9] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla, "Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art," in *14th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), H. Falk, Ed., vol. 39. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 31–42. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4602

[10] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in Multi-Core systems," in *16th USENIX Security Symposium (USENIX Security 07)*. Boston, MA: USENIX Association, Aug. 2007. [Online]. Available: https://www.usenix.org/conference/16th-usenix-security-symposium/memory-performance-attacks-denial-memory-service-multi

[11] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 357–367.

[12] S. A. Panchamukhi and F. Mueller, "Providing task isolation via tlb coloring," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 3–13.

[13] S. Mergendahl, S. Jero, B. C. Ward, J. Furgala, G. Parmer, and R. Skowyra, "The thundering herd: Amplifying kernel interference to attack response times," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 95–107.

[14] Q. Wang, J. Song, and G. Parmer, "Execution stack management for hard real-time computation in a component-based os," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 78–89.

[15] S. Baruah, J. Goossens, and G. Lipari, "Implementing constant-bandwidth servers upon multiprocessor platforms," in *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002, pp. 154–163.

[16] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 93–102. [Online]. Available: https://doi.org/10.1145/2380356.2380376

[17] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 200–212.

[18] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, jan 2012. [Online]. Available: https://doi.org/10.1145/2086696.2086713

[19] H. Wang, N. C. Audsley, and W. Chang, "Addressing resource contention and timing predictability for multi-core architectures with shared memory interconnects," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 70–81.

[20] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 684–689.

[21] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on wcet analysis of real-time embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 400–415, 2010.

[22] M. S. Inci, T. Eisenbarth, and B. Sunar, "Hit by the bus: Qos degradation attack on android," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 716–727. [Online]. Available: https://doi.org/10.1145/3052973.3053028

[23] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 174–187.

[24] O. Seongil, Y. H. Son, N. S. Kim, and J. H. Ahn, "Row-buffer decoupling: A case for low-latency dram microarchitecture," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 337–348.

[25] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for numa-aware contention management on multicore systems," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. USA: USENIX Association, 2011, p. 1.

[26] J. Liedtke, H. Hartig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," in *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, 1997, pp. 213–224.

[27] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81. Washington, DC, USA: IEEE Computer Society Press, 1981, p. 81–87.

[28] M. Bechtel and H. Yun, "Memory-aware denial-of-service attacks on shared cache in multicore real-time systems," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[29] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile paging: Exceeding the best of nested and shadow paging," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 707–718. [Online]. Available: https://doi.org/10.1109/ISCA.2016.67

[30] I. Chukhman and P. Petrov, "Context-aware tlb preloading for interference reduction in embedded multi-tasked systems," in *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*, ser. GLSVLSI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 401–404. [Online]. Available: https://doi.org/10.1145/1785481.1785574

[31] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 346–359. [Online]. Available: https://doi.org/10.1145/3307650.3322238

[32] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Truspy: Cache side-channel information leakage from the secure world on arm devices," *Cryptology ePrint Archive*, 2016.

[33] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing os abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303976

[34] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th international systems and storage conference*, 2013, pp. 1–10.

[35] Q. Huang and P. P. Lee, "An experimental study of cascading performance interference in a virtualized environment," *SIGMETRICS*

Perform. Eval. Rev., vol. 40, no. 4, p. 43–52, apr 2013. [Online]. Available: https://doi.org/10.1145/2479942.2479948

[36] T. Zhang and R. B. Lee, "Host-based dos attacks and defense in the cloud," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, ser. HASP '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3092627.3092630

[37] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense)," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 281–292. [Online]. Available: https://doi.org/10.1145/2382196.2382228

[38] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[39] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proceedings of the Linux Symposium*, 2012, pp. 119–130.

[40] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: a kernel for scalable predictability," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015, pp. 121–132.

[41] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.

[42] N. Zhang, R. Zhang, K. Sun, W. Lou, Y. T. Hou, and S. Jajodia, "Memory forensic challenges under misused architectural features," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 9, pp. 2345–2358, 2018.

[43] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *2018 IEEE Security and Privacy Workshops (SPW)*, 2018, pp. 116–122.

[44] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems." Ph.D. dissertation, University of Michigan, USA, 1975, aAI7609381.

[45] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1509.02971

[46] ——, "Continuous control with deep reinforcement learning," 2015. [Online]. Available: https://arxiv.org/abs/1509.02971

[47] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[48] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, ser. NIPS'99. Cambridge, MA, USA: MIT Press, 1999, p. 1057–1063.

[49] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[50] B. Sprunt, "Aperiodic task scheduling for real-time systems," Carnegie Mellon University, Tech. Rep., 1990.

[51] E. Bini, G. Buttazzo, and G. Buttazzo, "Rate monotonic analysis: the hyperbolic bound," *IEEE Transactions on Computers*, vol. 52, no. 7, pp. 933–942, 2003.

[52] C. King, "stress-ng," https://wiki.ubuntu.com/Kernel/Reference/stress-ng, Canonical, accessed: May 20, 2022.

[53] C. Li, S. Xi, C. Lu, C. D. Gill, and R. Guerin, "Prioritizing soft real-time network traffic in virtualized hosts based on xen," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 145–156.

[54] V. Subramonian, N. Wang, L.-J. Shen, and C. Gill, "The design and performance of configurable component middleware for distributed real-time and embedded systems," in *IEEE Real-Time Systems Symposium (RTSS)*, December 2004, pp. 252–261.

[55] 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, May 2007.

[56] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[57] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new linux readahead framework," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, p. 75–84, jul 2008. [Online]. Available: https://doi.org/10.1145/1400097.1400106

[58] Z. Zheng, R. Wang, H. Zhong, and X. Zhang, "An approach for cloud resource scheduling based on parallel genetic algorithm," in *2011 3rd International conference on computer research and development*, vol. 2. IEEE, 2011, pp. 444–447.

[59] H. Alibrahim and S. A. Ludwig, "Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization," in *2021 IEEE Congress on Evolutionary Computation (CEC)*, 2021, pp. 1551–1559.

[60] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia Tools and Applications*, vol. 80, no. 5, pp. 8091–8126, 2021.

[61] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 23–32.

[62] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 141–150.

[63] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 215–224.

[64] R. Taori, A. Kamsetty, B. Chu, and N. Vemuri, "Targeted adversarial examples for black box audio systems," in *2019 IEEE security and privacy workshops (SPW)*. IEEE, 2019, pp. 15–20.

[65] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 39–54.

[66] E. Feinberg, "Total reward criteria. feinberg ea, shwartz a, eds. handbook of markov decision processes," 2002.

[67] P. Cheridito, H. Kawaguchi, and M. Maejima, "Fractional ornstein-uhlenbeck processes," *Electronic Journal of probability*, vol. 8, pp. 1–14, 2003.

[68] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[69] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 805–825. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/qiu

[70] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 55–64.

[71] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. M. Montiel, and J. D. Tardós, "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam," *IEEE Transactions on Robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.

[72] "Orb-slam3," https://github.com/UZ-SLAMLab/ORB_SLAM3, Universidad de Zaragoza, accessed: May 20, 2022.

[73] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, "The euroc micro aerial vehicle datasets," *The International Journal of Robotics Research*, vol. 35, no. 10, pp. 1157–1163, 2016.

[74] A. Li, J. Wang, and N. Zhang, "Chronos: Timing interference as a new attack vector on autonomous cyber-physical systems," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2426–2428.

[75] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1573–1573.

[76] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee, "Holistic resource allocation for multicore real-time systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 345–356.

[77] A. Li, H. Liu, J. Wang, and N. Zhang, "From timing variations to performance degradation: Understanding and mitigating the impact of software execution timing in slam," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022.

[78] P. Pazzaglia, C. Mandrioli, M. Maggio, and A. Cervin, "Dmac: Deadline-miss-aware control," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[79] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica, "D3: A dynamic deadline-driven approach for building autonomous vehicles," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 453–471. [Online]. Available: https://doi.org/10.1145/3492321.3519576

[80] J. Orr, C. Gill, K. Agrawal, J. Li, and S. Baruah, "Elastic scheduling for parallel real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 6, no. 1, p. 05:1–05:14, 5 2019. [Online]. Available: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v006-i001-a005

[81] J. Orr, C. Gill, K. Agrawal, S. Baruah *et al.*, "Elasticity of workloads and periods of parallel real-time tasks," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, ser. RTNS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 61–71. [Online]. Available: https://doi.org/10.1145/3273905.3273915

[82] M. Sudvarg, C. Gill, and S. Baruah, "Linear-time admission control for elastic scheduling," *Real-Time Systems*, vol. 57, no. 4, pp. 485–490, 10 2021. [Online]. Available: https://doi.org/10.1007/s11241-021-09373-4