ABSTRACT

MITTAL, SWASTIK. Timing Integrity and Predictable Execution in High-Performance Real-Time Systems. (Under the direction of Dr. Frank Mueller).

Modern real-time and embedded systems increasingly integrate heterogeneous computing resources — multi-core CPUs, GPUs, and high-speed networks — to execute complex AI and safety-critical workloads under strict timing constraints. This heterogeneity improves performance but introduces unbounded and unobservable execution-time interference, leading to unpredictable delays, missed deadlines and potential safety and security violations. This dissertation presents three complementary techniques that detects, expose, and also control timing behavior at the network, multi-core runtime, and accelerator levels of a modern real-time system.

First, this dissertation presents T-Pack, a timed packet protection mechanism that detects network-based intrusions by analyzing end-to-end packet transmission delays. By extending the Linux network stack to embed and monitor timing information within the kernel for both TCP and UDP communication, T-Pack identifies compromised communication at packet reception time, complementing conventional global timeouts by catching transmission delays that arrive within the timeout window but exceed the real-time schedule's expected end-to-end time. The method composes with IPSec for cryptographic integrity, enabling early detection of network-induced timing attacks in distributed real-time systems.

Second, the dissertation presents Timed Threaded Execution (T-Tex), a runtime and compiler-assisted approach for detecting delay-based attacks and unexplained timing anomalies in multi-threaded applications. T-Tex extends OpenMP with timed execution monitoring, leveraging LLVM-based instrumentation and OpenMP tracing (OMPT) to correlate execution progress with expected timing behavior. Linux kernel-level support enables precise tracking of thread execution across context switches, allowing detection of time dilation with microsecond-level resolution on real hardware.

Finally, this dissertation presents OpenMP-RT-Offload, a transparent GPU offloading framework that brings real-time control to accelerator execution. Implemented as a CUDA interposer for NVIDIA GPUs, OpenMP-RT-Offload partitions streaming multiprocessors into isolated groups and routes OpenMP target regions to pre-created GPU contexts (green contexts) bound to these partitions. A priority-aware dispatcher enforces predictable kernel execution within each group while enabling concurrency across groups, achieving spatial isolation and reduced cross-task interference without code instrumentation or driver modifications.

Collectively, these contributions establish timing integrity across high-performance real-time systems and predictability at the accelerator level, advancing the state of the art in predictable and secure execution for modern real-time computing platforms.

Timing Integrity and Predictable Execution in High-Performance Real-Time Systems

by
Swastik Mittal

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2026

APPROVED BY:

Dr. Man Ki Yoon	Dr. Zhishan Guo
Dr. Justin Bradley	Dr. Frank Mueller Chair of Advisory Committee

DEDICATION

To my loved ones, whose patience and belief carried me through every challenge, and to the resilience I discovered along the way.

BIOGRAPHY

The author was born in Dehradun, India, and grew up in Pune and Delhi. He completed his schooling in 2014 and began pursuing a B.Tech. degree in Computer Science and Engineering at Vellore Institute of Technology, Chennai, India, graduating in 2018. He then moved to the United States to enroll in the graduate program in Computer Science at North Carolina State University, Raleigh. He completed his Master of Science degree in 2022 under the direction of Dr. Frank Mueller and continued at NC State for his doctoral studies under the same advisor. During his graduate studies, he held summer internships at Uber as a Compiler Optimization Engineer and at AIM Intelligent Machines as a Performance Optimization Engineer. Along with his thesis, he exhibits interests and expertise in compiler optimization, GPU scheduling/optimization, parallel computing, Linux-based networks, and real-time safety.

ACKNOWLEDGEMENTS

I would like to offer my deepest thanks to my MS and PhD advisor, Dr. Frank Mueller, for his unwavering support, mentorship, and guidance over the years.

I am grateful to my committee members, Dr. Man-Ki Yoon, Dr. Zhishan Guo, and Dr. Justin Bradley, for their time, insightful feedback, and valuable suggestions on my research.

I would also like to thank Dr. George Rouskas and Dr. Steffen Heber for their guidance through administrative matters during my graduate studies.

Finally, I would like to thank my loved ones — my family, friends, and my girlfriend — for their support throughout the highs and lows of my academic journey.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **viii**

AUTHORSHIP STATEMENT . . . . . **x**

Chapter 1 Introduction **1**

Chapter 2 T-Pack: Timed Network Security for Real Time Systems **3**
2.1 Introduction 3
2.2 Attack Model 5
2.3 Design 7
2.4 Implementation 11
2.5 Experimental Setup And Applications 13
2.6 Results 14
2.7 Related Work 18
2.8 Summary 19

Chapter 3 T-Tex: Timed Threaded Execution for Real-time Security and Safety **21**
3.1 Introduction 21
3.2 Related Work 23
3.3 Assumptions and Attack Model 24
3.4 Design 26
3.4.1 Identifying Code Regions 26
3.4.2 Timers 28
3.4.3 Accurate Analysis via Maintaining Timers at Context-Switch 28
3.5 Implementation 29
3.5.1 Compiler Modifications: Clang Frontend 29
3.5.2 Compiler Modifications: Linker Optimization (LLVM Pass) 30
3.5.3 T-Tex Data Structure and Subsequent Phases 32
3.5.4 Runtime Modifications: OpenMP & OMPT 32
3.5.5 Kernel Implementation of T-Tex 33
3.6 Experiments and Evaluation 34
3.6.1 Experiments 34
3.6.2 Evaluation 35
3.7 Summary 39
3.8 Supplementary Information 40
3.8.1 Multiple Timers (T-Tex) vs. Single Timers 40

Chapter 4 OpenMP-RT-Offload: Real-Time Support for OpenMP GPU Offloading **42**
4.1 Introduction 42
4.2 Related Work 44
4.3 Background 46
4.3.1 Executing Tasks with OpenMP-RT 46
4.3.2 NVIDIA GPUs in Embedded Systems 46
4.3.3 Dynamic Interpositioning of GPU Calls 48

4.3.4 RT-Mutex Design and Queuing 48
4.4 Design 48
4.4.1 Task and Platform Model 48
4.4.2 Queuing Abstraction 49
4.4.3 Timing Analysis (RM) 50
4.4.4 Timing Analysis (EDF) 52
4.4.5 Per Resource Queue 53
4.4.6 NVIDIA Write Lock Considerations 53
4.5 Implementation 54
4.5.1 Initialization and Partitioning 55
4.5.2 Task Identification 55
4.5.3 Interception and Dispatch 56
4.5.4 Single Queue Implementation For Comparison 56
4.6 Experimental Framework 57
4.7 Results 58
4.7.1 Evaluating Isolated Execution 58
4.7.2 Response Time Analysis 58
4.7.3 Weighted Schedulability Score 61
4.8 Summary 67

Chapter 5 Conclusion 68

REFERENCES 70

LIST OF TABLES

Table 2.1 ABBREVIATIONS. 8

Table 3.1 Parsec Benchmark Suite. 41

Table 4.1 Task Set Configuration - Multiple Interference. 51
Table 4.2 Task Set Configurations for OpenMP-RT Synthetic Benchmarks. 57
Table 4.3 Isolated Task Execution Times (ms), No Contention. 58
Table 4.4 Task Set Configurations for Weight Score Evaluation. 62
Table 4.5 Deadline Miss Rates (%) – Varying Partition Count (4 Tasks). 62
Table 4.6 Deadline Miss Rates (%) – Varying Task Count (2 Partitions). 65

LIST OF FIGURES

Figure 2.1 T-Pack and global timeout scenarios for tasks with real-time deadlines under (a) TCP (b) UDP. 6
Figure 2.2 a. One-Way (UDP) b. Two-Way (TCP). 7
Figure 2.3 Framework 1: Netfilter Hooks to Record Time Information for TCP (blue) and UDP (red) using T-Pack. 11
Figure 2.4 Framework 2: Custom Header Insertion/Removal using Socket Buffers for T-Pack in UDP. 12
Figure 2.5 RTT in ms with/without T-Pack; Configurations for 2.5b and 2.5c: No attack A1:P(0,0,0,0), attacks A2:P(1,10,500,0.5), A3:P(1,10,500,0.1), A4:P(2,10,500,0.1), A5:P(2,30,500,0.05):, A6:P(2,10,500,0), A7:P(2,30,1000,0.001). 15
Figure 2.6 Frequency Distribution and Overlapping Region of Attack 1 vs. 7. 16
Figure 2.7 Frequency Distribution and Overlapping Region of Attack 1 vs. 6. 16
Figure 2.8 Frequency Distribution and Overlapping Region of Attack 1 vs. 2. 16
Figure 2.9 T-Pack measured ETT in ms for Paparazzi over UDP without IPSec under different attack scenarios. 17
Figure 2.10 T-Pack measured ETT in ms for Paparazzi over UDP with IPSec under different attack scenarios. 17
Figure 2.11 T-Pack measured RTT in ms for Drone under different attack scenarios. 17

Figure 3.1 Analysis Model: Timed analysis of Parallel Region - broken into finer code regions - broken into loop iterations - analyzing context-switch time (t_1-t_7 represent the execution times of the regions they are marked alongside). . . . 25
Figure 3.2 Code Regions for Implicit Parallelism in OpenMP. 27
Figure 3.3 T-Tex Workflow. 29
Figure 3.4 Loop Control Flow Graph: Red Arrow = possible edges, Black Arrow = 100% existing edge, Blue Box = one or more basic blocks, Green condition = may or may not be in the given block a) Loop before T-Tex; b) Loop after T-Tex. . . 31
Figure 3.5 Phase 1-5 of T-Tex (further division of code regions to reduce WCET of each region) : Execution Time (ns) (black line: 60us security threshold). 36
Figure 3.6 Attack Detection and Vulnerability 1) T-Tex (No Attack) 2) 10us Delay 3) 20us Delay 4) 40us Delay 5) 500us Delay 6) Basic Block Protection (T-SYS: No Attack). 37
Figure 3.7 Time Distribution from Fig 3.6 — Image inside is a zoomed in image of the black rectangular box (Frequency Density of 0.001%): 1) 0.7% of red overlaps with purple 2) 0.05% of green overlaps with blue. 38
Figure 3.8 Benefit of Kernel Timer Crediting 1) T-Tex without kernel time crediting 2) T-Tex with kernel time crediting. 38
Figure 3.9 T-Tex security on parsec — frequency mining application: 1-4) Number of T-Tex iterations to reach security threshold of 60us 5) Attack: 60us Delay. 39
Figure 3.10 Single Timer: Custom Callbacks are instrumented timer calls for further dividing OpenMP recognized regions for finer analysis. More synchronizations needed for a smaller vulnerability window (resulting in more custom callbacks) defeating the purpose of multi-threaded model. 40

Figure 3.11 Multiple Timers: Timers armed at T_x are canceled (X) before being triggered, and a parallel region is partitioned into multiple sub-regions protected by finer-grained callbacks, which are not synchronized between threads until a final barrier. 41

Figure 4.1 Partitioned GPU with per-partition queues and separate memory queues for H2D/D2H. Example shown: partitions with 4/4/8/8 SMs. 49
Figure 4.2 Execution Timeline for τ_i Along with HP (higher priority) and LP (lower priority) Tasks on single CPU and GPU (Different Queues For H2D, Kern, D2H). Each task runs a sequence of CPU-GPU as described in the task model. Legends in the image show what each colored box represents. 52
Figure 4.3 OpenMP-RT-Offload Pipeline. 54
Figure 4.4 Comparison of Response times for Tasks A & C under the default baseline vs State-of-the-art (Single Queue) vs. OpenMP-RT Offload (Multi-Queue). 59
Figure 4.5 Comparison of Response times for Tasks B & D under the default baseline vs. State-of-the-art (Single Queue) vs. OpenMP-RT Offload (Multi-Queue). 60
Figure 4.6 Weighted schedulability score for 4 tasks under varying GPU partition configurations: default baseline, single queue, and multi queue. 63
Figure 4.7 Weighted schedulability score for 2 partitions under varying task counts: default baseline, single queue, and multi queue. 66

AUTHORSHIP STATEMENT

Contributions of Swastik Mittal and co-authors are listed below for each chapter.

Chapter 1: Introduction

Contributions:

- Swastik Mittal: sole author of Chapter 1.

Chapter 2: T-Pack: Timed Packet Protection for Network Security in Real-Time Systems

Citation: S. Mittal and F. Mueller, "T-Pack: Timed Network Security for Real Time Systems," in *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, IEEE, 2021, pp. 20–28.

Contributions:

- Swastik Mittal: preliminary investigation, design, implementation, literature review, development of methodologies, coding, evaluation, and writing (drafting).
- Frank Mueller: design, supervision, feedback, and writing (reviews/edits).

Chapter 3: T-Tex: Timed Threaded Execution for Real-Time Security and Safety

Citation: S. Mittal and F. Mueller, "T-Tex: Timed Threaded Execution for Real-time Security and Safety," in *ACM/IEEE 16th International Conference on Cyber-Physical Systems (ICCPS '25)*, ACM, 2025, pp. 1–11.

Contributions:

- Swastik Mittal: preliminary investigation, design, implementation, literature review, development of methodologies, coding, evaluation, and writing (drafting).
- Frank Mueller: design, supervision, feedback, and writing (reviews/edits).

Chapter 4: OpenMP-RT-Offload: Real-Time Support for OpenMP GPU Offloading

Citation: Submitted.

Contributions:

- Swastik Mittal: preliminary investigation, design, implementation, literature review, development of methodologies, coding, evaluation, and writing (drafting).
- Frank Mueller: design, supervision, feedback, and writing (reviews/edits).

Chapter 5: Conclusion

Contributions:

- Swastik Mittal: sole author of Chapter 5.

Use of generative artificial intelligence: In chapters 1-5, Large Language Model-based tool (Google Gemini & Claude) were used to improve clarity and grammatical correctness and bring smooth transitions between sentences.

CHAPTER

1

INTRODUCTION

Systems which emphasize predictability, efficiency, and include features to support timing constraints are real-time systems [Sta04]. Unlike commodity systems, which optimize for average-case throughput and fairness, real-time systems have execution deadlines for tasks. Missing these deadlines could result in fatal loss of life or environmental damage [Mit20]. Real-time systems are typically embedded within a larger physical system and must respond to its dynamics within bounded time [Lee08; Lee16], which makes worst-case — not average-case — execution time the figure of merit [Wil07]. This requirement is fundamentally at odds with the micro-architectural features (caches, speculation, out-of-order execution) and software stacks (fair-share schedulers, unbounded kernel paths) that commodity platforms rely on for performance [Wil08; Reg19].

Real-time systems increasingly execute safety-critical workloads on high-performance platforms that integrate distributed networked nodes, multi-core CPUs, and GPU accelerators [Cer20]. While this heterogeneity provides the computational capacity required by modern AI and control workloads, it also exposes the system to unbounded or unobservable execution-time interference — whether caused by adversarial attacks [Fra11b; Fra11a; Cal09; SHA18; Lou19], contention between tasks using shared resources [Pel10], or device complexity (e.g., GPU execution) [Yan18] — threatening the timing guarantees on which these systems depend.

Hypothesis: Unbounded or unobservable execution-time interference is a primary obstacle to reliable real-time behavior in modern high-performance systems. Software mechanisms that provide fine-grained timing observability and enforce bounded resource usage through priority-based scheduling and isolation are therefore necessary to preserve timing guarantees and detect timing anomalies across distributed, multi-threaded, and accelerator-based execution domains.

To evaluate this hypothesis, we develop three primary contributions, each operating at a dif-

ferent level of a modern real-time system: the network, the multi-threaded runtime, and the GPU accelerator. For the first two contributions, fine-grained timing observability is achieved by decomposing task-level timing bounds into per-packet and per-code-region bounds, enabling detection of timing anomalies well in advance of task-deadline expiration. The third contribution addresses timing predictability through priority-based scheduling and spatial isolation at the accelerator level.

In Chapter 2, we present T-Pack, a timed packet protection mechanism that addresses timing integrity in networked real-time systems. We extend the Linux network stack to embed and analyze timing information in packet transmissions, enabling the detection of network-induced time dilation caused by dropped, delayed, or modified packets. T-Pack identifies compromised communication at packet reception time, complementing conventional global timeouts by catching transmission delays that arrive within the timeout window but exceed the expected end-to-end time. We evaluate T-Pack on distributed embedded platforms running a Preempt-RT Linux kernel [Reg19] to demonstrate that deviations in end-to-end packet timing can detect compromised systems well before task deadlines are violated.

In Chapter 3, we present Timed Threaded Execution (T-Tex), which addresses timing integrity at the runtime and thread-execution level in a multi-threaded real-time application. T-Tex extends OpenMP [Var15] with timed monitoring of code execution, using compiler-based instrumentation and OpenMP tracing capabilities combined with Linux kernel-level support for tracking execution time across context switches. We demonstrate that precise monitoring of thread execution time can reliably detect delay-based interference in multi-threaded real-time applications on real hardware.

In Chapter 4, we present OpenMP-RT-Offload, which addresses timing predictability in accelerator based execution. Implemented as a transparent CUDA [Cud] interposer for NVIDIA GPUs, OpenMP-RT-Offload partitions streaming multiprocessors into isolated groups and routes OpenMP target regions to preassigned groups. A priority-aware dispatcher enforces predictable kernel execution within each group, while spatial isolation eliminates interference between tasks assigned to different groups. This improves execution predictability without modifying application code or the OpenMP runtime.

Together, these contributions demonstrate that analyzing and scheduling execution behavior at the network, thread, and accelerator layers can enforce timing integrity against malicious interference at the network and multi-threaded execution levels, and timing predictability at the accelerator level — advancing predictable and secure execution for modern high-performance real-time computing platforms.

CHAPTER

2

T-PACK: TIMED NETWORK SECURITY FOR REAL TIME SYSTEMS

2.1 Introduction

Computer security is a critical requirement for any networked application nowadays. Main components of computer security include: (1) System level security, securing end system's code at different levels and layers from attacks leveraging memory (e.g., buffer overflow attacks including stack smashing [One96], heap overflows [Con99]) or non-memory related attacks (e.g., value range overflows [Hor02], shell shock [Mar15], port smashing); and (2) network security, securing systems on the network from being attacked by malicious users. For the latter, a wide range of attacks are common, including relay, replay, phishing, spoofing, man-in-the-middle, denial of service, eavesdropping etc. [Fra11b; Fra11a; Cal09; SHA18]. One of the common attacks in real-time systems is the delay attack. The objective of this attack is to stall execution/packets of a time-sensitive event (e.g., a code section within the system or some client/server request) causing excessive delays and resulting in performance degradation [Lou19]. Real-time systems are particularly susceptible to such attacks as system behavior is compromised when deadlines are missed, i.e., time dilation not only results in performance penalties or reduced network throughput but may cause a control system to malfunction, which can result in damage to the controlled environment or even loss of life. Past work shows how delay attacks have affected cyber-physical systems (CPS) [Lou19] and network control systems [Sar15] subject to real-time constraints.

Significant work has been invested in analyzing and mitigating the impact of these attacks [Lou19; Sar15]. Real-time systems offer a unique opportunity for intrusion detection besides traditional,

general-purpose cyber-security techniques: Their inherent knowledge of worst-case execution times (WCET) [Wil07], an upper bound on a task's execution budget required for real-time scheduling, opens up opportunities for additional monitoring and protection. The same techniques for establishing timing bounds on the execution of the real-time task may be applied to bound execution of any code section within the application [Gor04; Wil08]. Past works have used this model for security in real-time systems for detecting memory attacks [Zim10], securing clock synchronizations over the network [Nar18] and protecting smart grid systems [Ped16]. Timed analysis is not just restricted to security, it can also be used to design attack models, e.g., in the context of hardware security tokens such as Smartcards [Dhe98; Koc96; Sch00], exported secret decryption keys [Koc96] and remote timing attacks [Bru05]. This work applies time-based security to detect delay attacks on a network subject to communication with real-time constraints. This complements methods monitoring execution times for intrusion detection by providing a similar mechanism for packet transmissions.

In safety-critical distributed real-time systems, missed deadlines due to slower or missing packets could result in significant environmental damage or even in loss of life. System restarts often cannot be instantaneous due to an unstable physical system state. This research focuses on detecting intrusions due to such attacks at the packet level, i.e., before malware in one subsystem can enter another subsystem or even result in a deadline miss within the yet unharmed subsystems. The earlier intrusion is detected, the easier it is to resort to a safe operational mode with reduced or even without communication to another subsystem that has been compromised, e.g., using the Simplex design [Cre07; Bak09], thereby avoiding any significant compldamage. Our work focuses on early intrusion detection on end nodes (as opposed to routers/switches as that would incur additional time to notify nodes), and while it relies on established methods to transition to a safe state (e.g., Simplex), the safe methods are beyond the scope of this paper.

The foundational concept of timing-based packet protection was introduced in the author's prior MS thesis [Mit20], which established the core design for TCP, a preliminary UDP formulation assuming synchronized clocks, and evaluation of TCP-based benchmarks on a client-server model, the Paparazzi UAV, and a drone-like multi-system. The present work extends that foundation with a formalized attack model centered on early intrusion detection that complements conventional global timeouts — catching transmission delays that arrive within the global timeout window but exceed the real-time schedule's expected end-to-end time, a working UDP implementation with periodic clock synchronization, composition with IPSec under both transport protocols, UDP evaluation on the Paparazzi UAV, a header-based rather than trailer-based timestamp placement and a broader positioning of T-Pack against contemporary time-sensitive networking standards and hardware-timestamping protocols (IEEE 802.1Qbv, IEEE 1588, and secure clock synchronization work). Detection accuracy is further quantified via F1 scores and an instruction-budget analysis of the undetected attack window.

Our work contributes:

- T-Pack, a method for packet protection combining simplicity in design and implementation

with integrity, negligible performance cost and no hardware modifications.

- T-Pack is compatible with other security protocols and utilizes IPSEC to establish data integrity alongside early detection of delay attacks.
- Malware intrusion is detected by monitoring end-to-end packet deadlines at the time of packet reception instead of conventional detection at a task's deadline.
- Experiments with real-time applications under attack scenarios assess potential and limitations.
- Results indicate that T-Pack has low overhead per packet round-trip time (≈ 0.09 milliseconds) and detected 95%-100% of the delays during ping flooding and distributed denial of service (DDOS) attacks in a number of experimental systems.

2.2 Attack Model

This work assumes a distributed environment with end nodes connected by a network with end-to-end real-time guarantees of message transmission, e.g., via packet prioritization. This can be accomplished by (expensive and proprietary) hardware solutions like TTEthernet [Kop08], protocol extensions for time-based traffic shaping (e.g., 802.1Qbv [Wik], if supported), or via enhancements on top of (less expensive) software-defined network (SDN) equipment [Qia17].

Each subsystem (node) within this CPS architecture is assumed to provide its own execution environment (processor). Inter-node communication is prioritized for real-time traffic in a statically constructed schedule. By this design, a sending subsystem puts a message on the wire via the network stack such that it is received prior to an end-to-end message deadline at the receiving subsystem, but the exact time of the send/receive activity may vary within deterministic bounds as they are determined by execution times of prior tasks.

An attacker may compromise a given subsystem and subsequently monitor network traffic and inject packets arbitrarily to either execute replay attacks (injecting duplicate packets), use IP spoofing (injecting packets with an incorrectly rewritten source address expecting lost reply packets from the destination) or flooding (SYN or ICMP packet flooding). Other subsystems remain unaffected in computation and their ability to send/receive packets as packet communication is assumed to use public key encryption, i.e., receiving nodes can detect content modifications including packet headers. (Even a replay attack with modified source can be detected by encrypting all original headers in the message such that a source mismatch between received header and decrypted header in the message can be detected). Other subsystems may, however, be affected by altered network behavior due to packets originating from the compromised node (additional or duplicated packets at any priority, modified packets, dropped packets with respect to the original static schedule). Most significantly, the compromised system neither has the ability to alter packets sent by uncompromised systems, nor may it change any router/switch functionality. A delayed

packet reception results in an intrusion notification, just as an omitted packet, as a timeout will be raised at expected arrival time if the packet has not arrived yet. The benefit of our method is an early notification upon expected message arrival rather than a late timeout upon deadline violation, which leaves more time for exception handling / transitioning to a safe mode.

Fig. 2.1 illustrates T-Pack capabilities for both (a) TCP and (b) UDP, for a packet sent at t_S and received at t_R under real-time constraints with a task's absolute deadline of t_O (in line ①), which will trigger a timeout and exception if the packet has not been received by then. This deadline is present with or without T-Pack. The figure further depicts the expected arrival of the packet in range Δt_R. ②,③ & ④ illustrate the different distributions of the real-time events at times t_S, t_R & t_E for the packet sent, packet received and worst case end-to-end time (with time duration as Δ), respectively. For TCP, t_E is also the T-Pack timeout relative to the time when the packet is sent allowing earlier exception irrespective of packet arrival at t_R. Thunderbolt and cross (X) show triggered and canceled timers, respectively, plus exception (if triggered).

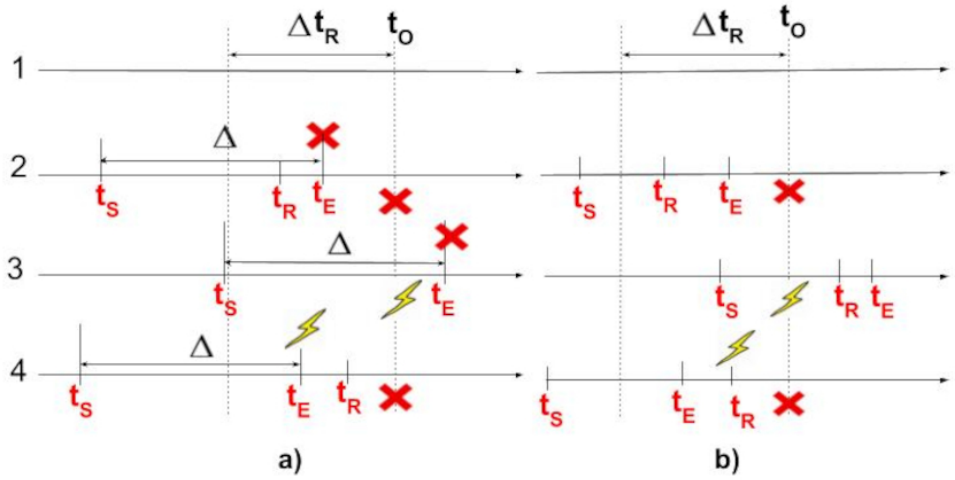


Figure 2.1 T-Pack and global timeout scenarios for tasks with real-time deadlines under (a) TCP (b) UDP.

② depicts a T-Pack scenario accepting an incoming packet at t_R followed by validation of the embedded timestamp. In Fig.2.1(a) under TCP, the packet is received before timeout t_E, i.e., the timeout at t_E is canceled. In Fig.2.1(b) under UDP, the duration for transmission, $t_R - t_S$, matches the expected transmission costs so that (i) *no* T-Pack exception needs to be raised as the duration is less than t_E; and (ii) timeout t_O can be canceled since the packet was received (both indicated by an X).

③ depicts a lost packet or a long delay before the packet is sent at t_S — both triggering a timeout at t_O resulting in an exception. As the packet is received at t_R, it will neither raise another exception at t_E (TCP Fig.2.1(a)), nor will it indicate a delay at t_R (UDP Fig.2.1(b)) as a late packet is simply ignored in hard real-time systems. In (m,k)-firm or soft real-time environments, where $m-k$ deadlines may be missed, the packet would not be ignored so that t_E can raise an exception if the transmission

took too long, see next case.

④ depicts a packet sent early (t_S) as the sending real-time task takes less time than the budgeted WCET. A Timeout is triggered by T-Pack at t_E (TCP Fig.2.1(a)) as the packet has yet to be received; or, upon packet reception, the receiver notices that the transmission exceeded the expected duration (UDP Fig.2.1(b)) based on the embedded timestamp (i.e., $t_R - t_S > t_E$ is too long), indicating a delay at t_R, which cancels timeout t_O due to the early exception. This indicates a possible intrusion, which subsequentl triggers mitigation techniques. This illustrates the benefit of T-Pack over the global timeout as the prior exception due to timestamp validation leaves more time ($t_O - t_E$ under TCP and $t_O - t_R$ under UDP) for mitigation, i.e., to transition into a safe mode. Most of all, this scenario would *not* result in any exception without T-Pack, i.e., (a) the triggered timeout at t_E or (b) reception at time t_R would cancel the timer at t_O even though the packet was delayed significantly.

2.3 Design

T-Pack is a methodology to verify end-to-end timing of each packet on the network of a real-time system during message transfer between subsystems. Fig 2.2 depicts a high-level timing model for message transfer between two subsystems for unidirectional UDP (left) and bidirectional TCP transfers (right) using the notation established by Table 2.1. A message from sender S to receiver R is analyzed at packet level, considering packet P being sent at time t_S from S to R, where it is received at time t_R. The observed end-to-end time, T_{obs}, is compared with the expected time, T_{exp}, to detect malware intrusion in the network. The work assumes loosely synchronized clocks with a constant time difference, Δt_{rs}, between any two subsystems, which may be dynamically updated due to clock drift, as in our later implementation.

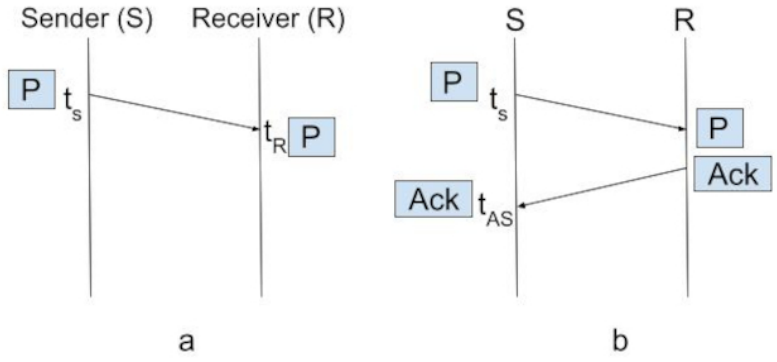


Figure 2.2 a. One-Way (UDP) b. Two-Way (TCP).

Table 2.1 ABBREVIATIONS.

t_S	Time recorded by T-Pack at which packet is sent from S
t_R	Time recorded by T-Pack at which packet is received at R
ACK	Acknowledgment packet from R to S in a 2-way message
t_{AS}	Time at which ACK is received at S (At Network Layer)
RTT	Round-trip time (TCP)
ETT	End-to-end time (UDP)
Δt_{rs}	Constant clock difference between S and R
T_{obs}	Observed end-to-end or round-trip time
T_{exp}	Expected end-to-end or round-trip time
T_d	Mean internal delay on the uncompromised network
ΔT_d	Deviation of internal delay from the mean internal delay
T_c	Added delay due to a compromised network
ΔT_c	Deviation of added delay from the mean added delay
T_{WCET}	Expected worst-case end-to-end time of packet

One Way Message Transfer

The UDP transport protocol is a suitable protocol assuming point-to-point full-duplex switch connectivity between network endpoints. Under UDP, a message is transferred from S to R (Fig. 2.2a) without any acknowledgment from R. In this scenario, the observed end-to-end time (ETT) is the time it takes for the packet to reach R having been sent by S.

$$ETT = t_R - t_S + \Delta t_{rs} \tag{2.1}$$

So, the expected end-to-end time is:

$$T_{exp} = ETT \tag{2.2}$$

Two-Way Message Transfer

TCP is a two-way message transfer transport protocol based on a handshaking protocol that acknowledges packets sent from the sender, S, to achieve reliable communication over the network

(Figure 2.2b). In this scenario, a transfer is said to be complete once the acknowledgment (ACK packet) is received at S, again under point-to-point full-duplex switch connectivity. Here, we assume a constant clock difference between sender and receiver for the duration of the packet communication. Clock drift requires this difference to be updated from time to time, which is typical for distributed systems and beyond the scope of the paper [Mil91].

Round-trip-time in TCP can be monitored without embedding time information within the packet. Instead, it suffices to measure the round-trip time of the packet (from the send to receipt of the acknowledgment at the sender).

$$RTT = t_{AS} - t_S \tag{2.3}$$

So, the expected end-to-end time is:

$$T_{exp} = RTT \tag{2.4}$$

TCP optimizes network traffic by consolidating multiple small bytes packet into one (reducing header overhead) at the sender [Rfc] and sending cumulative acknowledgments at the receiver (reducing multiple ACK packets). This creates non-deterministic execution behavior in real-time system due to delays for sending and receiving. [Buf99] shows how this affects the performance of a client-server application using the same communication pattern as distributed real-time systems. With controlled flow of packets in distributed real-time systems, additional data due to headers/trailers hardly reduce network bandwidth; instead, timely delivery is more important. We ensure timely delivery via socket options "TCP NODELAY" (sender) and "TCP QUICKACK" (receiver), which prevents packet consolidation.

The impact is analyzed for our experimental model (Paparazzi UAV, see Sec. 2.5) by monitoring the average bandwidth and number of packets flowing in and out of the interface for a period of time with and without these socket options. Without socket options, we observe an average rate of data observed at the receiving (rx) end of 353.50 Kbps and at the transmitting (tx) end of 779.67 Kbps over 60 seconds. With socket options, this decreases only slightly to 347.98 Kbps and 753.80 Kbps at receiver and sender, respectively. In and out flows of packet also decreased slightly, i.e. from 486 to 473 packets/sec (receiver) and 742 to 713 packets/sec (sender). More significantly, a packet sent with socket options was instantly sent, whereas it was non-deterministically buffered at times without options. Clearly, the latter is not acceptable for real-time system, whereas a small decrease in bandwidth is.

Relation to the Attack Model

Using our T-Pack model, we aim to detect delay attacks within uncompromised subsystems. Such an attack may originate from a compromised subsystem that maliciously induces time overheads by injecting packets arbitrarily. This may cause delays at the switch due to ingress queue processing, even if packets are prioritized, in part because the compromised subsystem can prioritize packets as well. As a result, any (non-malicious) packet that is forwarded through such a switch may be delayed before it can reach the other uncompromised subsystem. For a message transfer (UDP &

TCP), we have:

Using T_{exp} from Eq. 2.4,

$$T_{obs} = T_{exp} + T_c \pm \Delta T_c. \tag{2.5}$$

The objective of T-Pack is not to prevent intrusion but rather to **detect** it within uncompromised subsystems due to incorrect timing on network behavior. This can prevent these other subsystems from becoming compromised as well — by a timely transitioning into a safe mode, e.g., via Simplex [Cre07; Bak09] or other mode transitions depending on the application scenario, which is beyond the scope of the paper. Furthermore, intelligent switches could assist in blocking high priority packets that are non-compliant with a statically established end-to-end real-time message schedule under our attack model, but we do not expect such switches to actively notify end nodes of a compromised subsystem. This would violate the existing real-time schedule and induce sporadic messages, which may dilate latencies to where deadlines could be missed. Instead, T-Pack provides a means for uncompromised subsystems to autonomously detect intrusions by monitoring their own communication with other nodes.

Vulnerability Of T-Pack

Encryption prevents third party packet modifications by attackers as the private key of the receiver is unknown, even with access to the wire. This includes timestamp values of T-Pack within packets.

The WCET bound of a packet is determined by the end-to-end transfer in our model as follows. T_{exp} in T-Pack includes delays in an uncompromised system (expected delays on the network or internal delays). Let this delay be of magnitude $T_d \pm \Delta T_d$ (Table 2.1). Hence, the WCET bound includes a maximum internal delay of $T_d + \Delta T_d$, which signifies the maximum positive deviation of the WCET.

$$T_{WCET} = T_{exp} \tag{2.6}$$

For a compromised network, we obtain T_{obs} from Eq. 2.5. For some values of $T_c \pm \Delta T_c$, where the attacker delays the packet transfer by a small value, we may find that $T_{obs} \leq T_{WCET}$, i.e., short delays may remain undetected. In other words, our model is probabilistic and may result in missed intrusion detection (false negatives), where our model does not identify an attacker in the network. This illustrates two points: (1) Our model complements existing cyber security measures and (2) the objective of T-Pack is to make the attack window that remains undetected as small (short) as possible, but only if attack traffic affects control functionality (deadlines). As long as packets are received in time, subsystems remain intact, i.e., additional background traffic may be tolerated.

Time Information

To support UDP, T-Pack embeds timing information within each packet to verify that end-to-end times of a packet are within given WCET bounds. A custom header with timing information is added just above the packet payload within the kernel comprising the lower level of the network stack

(instead of higher networking layers). This establishes tighter and more deterministic bounds on the elapsed network delay.

The header-based approach also permits an alternative placement between the network and transport headers in L2-switched deployments, where switches forward on MAC addresses without inspecting the transport layer. In that configuration, the UDP checksum scope remains unchanged, avoiding recomputation of the UDP checksum over the payload.

To support TCP, time information of a sent packet is analyzed within the kernel with round-trip-time calculated using the corresponding acknowledgment received without the need to embed additional information within the packet. A lookup table is maintained on each subsystem to store sent time information of the packet to other subsystems.

Inclusion of TCP_NODELAY and TCP_QUICKACK ensures that only one outstanding packet from the same subsystem exists before an ACK is sent for the respective packet at any time.

Due to the static size of the table, the execution time of a table lookup is constant, which makes bounds under T-Pack highly predictable.

2.4 Implementation

Linux

T-Pack is implemented in a PREEMPT-RT patched Linux kernel that provides real time capabilities to the operating system. This provides the flexibility of utilizing Linux network APIs such as socket buffers and netfilter to implement T-Pack.

Netfilter

Netfilter is a framework provided by the Linux kernel to implement customized handlers on events in the network layer (for pre-routing, post-routing, etc.). T-Pack utilizes this framework to implement callback functions to time packets (Fig. 2.3).

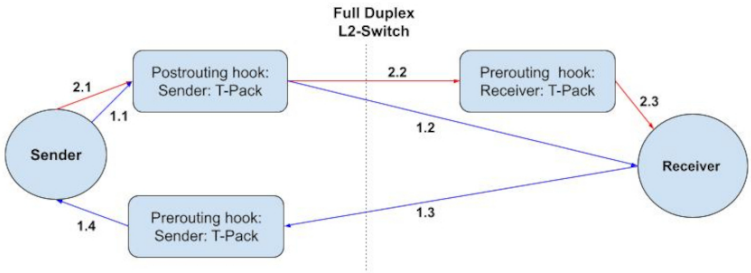


Figure 2.3 Framework 1: Netfilter Hooks to Record Time Information for TCP (blue) and UDP (red) using T-Pack.

Socket Buffers

Socket buffers are data structures provided by Linux as a common reference to packets in all layers of the network stack within the kernel. The T-Pack prototype utilizes the data types and the helper functions in the socket buffer API to record time (TCP) or manipulate packet memory in order to create additional space for the custom header (UDP) as depicted in Fig. 2.4.

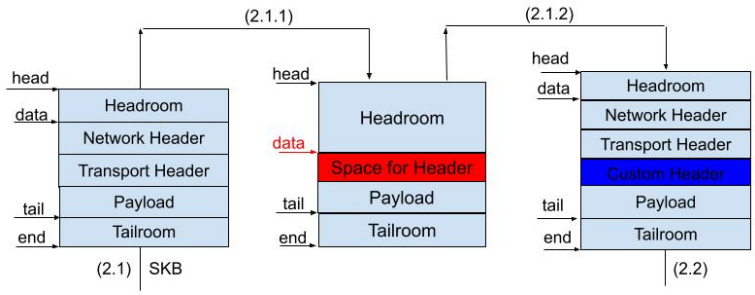


Figure 2.4 Framework 2: Custom Header Insertion/Removal using Socket Buffers for T-Pack in UDP.

Implementation Framework

1) T-Pack for TCP (Fig. 2.3 with network path in blue):

1.1) At the sender, the netfilter post routing hook is utilized to call a handler, where the socket buffer references the packet as an argument.

1.1.1) The current time at the sender is recorded and stored in a lookup table for the corresponding subsystem (IP address, port) as a key. The expected acknowledgment sequence number is also recorded for the same key, which is the sequence number in the packet plus the data payload.

1.2) According to TCP (with socket options as indicated in Section 4.4), a packet is received at the receiver (1.2) and a corresponding ACK is sent (1.3) immediately.

1.3) At the sender, the netfilter pre-routing hook is again utilized to call a handler, where the socket buffer references the ACK packet as an argument.

1.3.1) The received ACK is matched with the destination IP address and port. If the expected sequence number matches the number in the ACK packet, the stored time is subtracted from the current time to determine the round-trip-time of the packet, which is then compared with the worst case round-trip-time to detect potential intrusions.

2) T-Pack for UDP (Fig. 2.3 with network path in red):

2.1) At the sender, the netfilter post routing hook is utilized to call a handler, where the socket buffer references the packet as an argument.

2.1.1) The $skb\_pull(sizeof(network\_header))$ and $skb\_push(sizeof(custom\_header))$ function is used to create additional space just before the packet payload to attach the custom header (see Fig 2.4).

2.1.2) $memcpy(ptr, custom\_header)$ encapsulates the custom header by copying it to the packet. The custom header includes fields for *ktime_t timestamp* and *long sendtime*, where timestamp contains the current time (at header creation).

2.2) At the receiver, the netfilter pre-routing hook is utilized to call a handler, where the socket buffer references the packet as an argument.

2.2.1) Upon UDP reception, the difference of the current time and the timestamp embedded within the packet represents the end-to-end processing time of lower layer UDP activities, subject to validation against an expected upper bound for the exchange. If validation fails, an intrusion is signaled.

2.5 Experimental Setup And Applications

Experiment 1: Client Server Model

A client sends periodic messages (interval of 10ms) to the server according to the network activity of a time triggered real-time system. TCP messages that fit in a single packet are sent with an explicit reply packet from the server. Measuring this ping-pong message transfer derives the round-trip time (RTT) at the client. We measure RTT at both application and network layers to assess the benefits of implementing T-PACK at the network layer. We also measure the RTT at the application layer with and without T-Pack to analyze T-Pack's performance overhead.

Experiment 2: Paparazzi UAV Model

The Paparazzi UAV [Zim12; Bri06] models a real-time control system utilizing shared memory, which we transformed into a peer-to-peer network of 3 subsystems: an auto pilot (AP), a fly by wire (FBW) control and a ground station communicator (GSC) to relay information from subsystems to ground or vice-versa. We prototyped a model constrained to only Paparazzi's periodic messages scheduled between the above three subsystems. Each subsystem is connected via the (1) UDP and (2) TCP protocols with a persistent connection. The subsystems communicate with each other periodically transferring necessary information for flying by wire autonomously with T-Pack integration. The RTT is measured between AP and GSC communicator to monitor T-Pack. A delay attack as a Distributed Denial of Service (ICMP packet flooding / ping-of-death) [Cro03] is induced at the GSC using other nodes in the network as attackers. This resembles code injection by the attacker on the compromised subsystem to inject packets arbitrarily. We implemented this Paparazzi model on a network of Raspberry Pi systems with a Preempt RT patched Linux kernel to provide real-time capabilities.

Experiment 3: Waters Workshop Challenge 2018, a Drone-like Multi-System

A drone-like multi-system [Led18] within a peer-to-peer network of seven subsystems is implemented consisting of a Mission management system (MMS), Electrical Propulsion System (EPS), Hydraulic Braking System(HBS), Sensors (communicating with other sensors in the Waters model),

Ground Station and Maintenance System, connected via a hub and spoke topology within the same subnet. We again model functions and communication patterns in each subsystem, including periodic calls to the functions. Each subsystem is connected via TCP (persistently), sending messages to other subsystems in parallel under random network congestion with T-Pack support. The RTT is measured between EPS and MMS to monitor T-Pack functionality with and without a delay attack on Raspberry Pis with Preempt RT-patched Linux.

2.6 Results

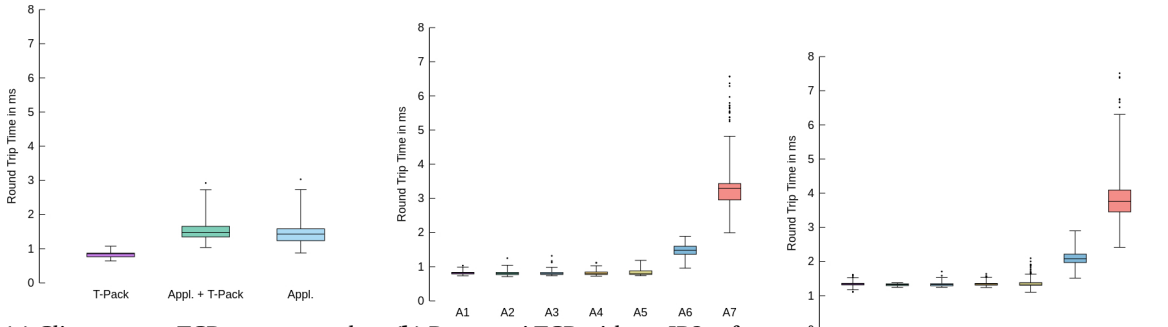
Experiment 1

Results for different server client configurations (x-axis) are depicted in Fig. 2.5a with RTT (y-axis) of messages shown as box plots indicating maximum, top quartile, median, bottom quartile and minimum times as well as outliers (dots). Min-max values or variability outside the upper and lower quartiles are denoted by whiskers with a range 3.5 times that of the inter-quartile range (constant for all other measurements within the paper). The outliers in this graph are only 0.5% of the total data values. We report all values from the experiments, even the first iteration of execution, which may be subject to additional cache misses resulting in an outlier.

We observe that the time measured for T-Pack (box plot 1 in Fig. 2.5a)for the reduced network stack is much lower than the one measured by the application, both with and without T-Pack (plots 2+3). By monitoring time within the kernel, T-Pack eliminates the cost of upper kernel layers both for sender and receiver resulting in an earlier intrusion detection at the network layer than at the application layer (baseline). Measuring RTT at the application layer would also require the applications to have explicit replies to every sent request, i.e., up to twice the number of messages are required in contrast to an implementation within the kernel. This could lead to unnecessary saturation of the write buffer of the receiver, which might be due to a send causing corresponding receivers to delay their communication. This increase in traffic would also result in higher RTTs for all the packets, in turn resulting in delayed intrusion detection because of larger timeouts. False negatives for timeouts at the network layer were measured by introducing a DDOS attack in Experiment 1 using a single attacker with 10 attack threads, each sending 100 bytes of ICMP packets at an interval of 0.001 seconds. Over a range of 300 packets sent, ∼170 detected intrusion due to timeouts at the kernel-level network layer of which 130 were false negatives (F1 score of 0.723) compared to ∼289 false negatives at the user-level application layer (F1 score of 0.071). This illustrates the benefits of our approach with T-pack within the kernel at the network layer. The attack introduced above is of intensity between A5 and A6 2.5b.

The results also reveal the overhead without the T-Pack module. Fig. 2.5a indicates that T-Pack incurs a modest performance cost as the overall mean RTT of the client request-reply increases by a marginal amount of approximately 0.09 msecs.

We analyze consistency, flexibility and integrity of T-Pack over secure communication between the client and the server by implementing the client-server model on top of a communication chan-



(a) Client-server TCP over network layers.

(b) Paparazzi TCP without IPSec for A1-7.

(c) Paparazzi TCP with IPSec for A1-7.

Figure 2.5 RTT in ms with/without T-Pack; Configurations for 2.5b and 2.5c: No attack A1:P(0,0,0,0), attacks A2:P(1,10,500,0.5), A3:P(1,10,500,0.1), A4:P(2,10,500,0.1), A5:P(2,30,500,0.05):, A6:P(2,10,500,0), A7:P(2,30,1000,0.001).

nel protected by the IP-Security (IPSec) protocol at the transport layer with RSA key authentication and encryption.

Any data packet is encrypted by IPSec after T-Pack in UDP adds its custom header. Should an attacker (who does not hold the private key) modify the packet, this would be detected as data then becomes corrupted after decryption, including T-Pack's timestamps.

IPSec provides security against session hijacking, man-in-the-middle attacks etc. This cannot prevent attackers imposing delays by transmitting unwanted packets, but T-Pack will detect such delays. Of course, data integrity comes at the price of increased RTT (TCP) and ETT (UDP), as assessed in the next experiment.

Experiment 2a: UAV Paparazzi Subsystems under TCP

We monitor the RTT under TCP from AP to GSC (i.e., send and acknowledgment) under a delay attack. Each attacker features a multi-threaded program to send large ICMP ping packets in quick intervals to the GSC. This causes a buffer overflow at the receiving interface, which is handled but results in performance degradation. The sensitivity to attack intensity is investigated dependent on a tuple, $P(n,t,b,i)$, by varying the number of attackers, n, the number of threads within an attacker, t, the ping packet size, b (in bytes), and the time interval, i (in seconds), between packets for the tuples indicated in Figures 2.5b and 2.5c. By modifying parameters, the intensity gradually increases to a level where the DDOS attack results in a noticeable impact due to buffer overflows on the network devices, which eventually causes deadline misses due to excessive RTT (TCP)/ETT (UDP).

Fig. 2.5b depicts RTT as box plots again (y-axis) over different intensities of DDOS attack (x-axis). The outliers in this graph are only 0.5% of the total data values (eliminating only 1% of the extreme outliers including the first iterations) As attack intensity increases, the RTT increases slightly. Compared to attack A1, A2 hardly effects the average RTT,

A6 increases the RTT on average by ≈ 0.65 msecs, and A7 by ≈ 2.6 msecs. For A7, all the measured

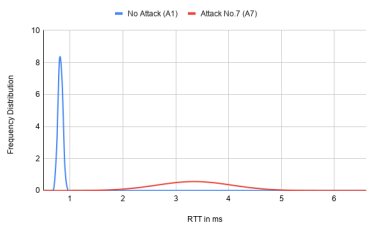


Figure 2.6 Frequency Distribution and Overlapping Region of Attack 1 vs. 7.

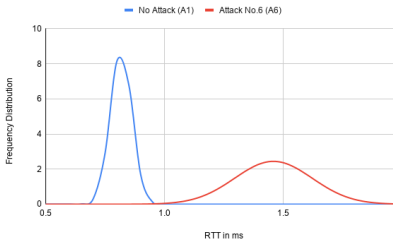


Figure 2.7 Frequency Distribution and Overlapping Region of Attack 1 vs. 6.

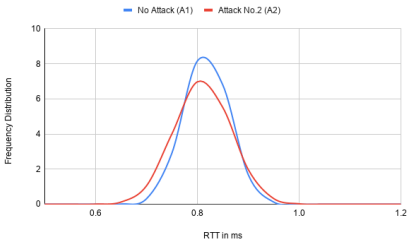


Figure 2.8 Frequency Distribution and Overlapping Region of Attack 1 vs. 2.

RTTs exceed those of A1 without any false negative (no overlapping values). In other words, T-Pack accurately captures the "attack vector", since the WCET bound of RTT without intrusion is lower than the minimum RTT on a compromised network. Recall that to take over an entire kernel, millions of instructions are typically required. We can limit an attack to 15k instructions here assuming, e.g., a CPU clock of 1GHz at an instruction per cycle rate of one, which restricts undetected intrusions under T-Pack to a ≈ 0.2 msec window. Thus, a chain of 5 attack instances with 200K instructions each would be required (adding up to 1M instructions) to take over the system without being detected.

We also observe a sudden increase in RTT in Fig. 2.5b when as attack intensity increases. DDOS prevents effective resource utilization by consuming most of the resources (network and receiver buffer here) within the attacker [Mir04]. A low intensity of such an attack does not affect the performance of a low traffic network, which is typical for a number of distributed real-time systems. However, intensifying the attack can cause sudden spikes that instantly degrade the network latency leading to packets arriving after a deadline, if at all. T-Pack detects this, which allows a system to transition to a safe mode while continuing to operate. Transitioning back online requires the attack source to be removed in a DDOS attack, both for real-time or commodity computing environments, i.e., counter measurements addressing the root cause remain unchanged and are beyond the scope of this paper.

We further analyze the results of frequency distributions in Figures 2.6, 2.7 and 2.8, which depict the number times (y-axis) a certain RTT (x-axis) was measured in experiments. Figure 2.6 indicates the distribution for without intrusion (A1 in blue) and with high-intensity intrusion (A7 in red). An empty intersection between the distributions indicates that both the cases can be discretely distinguished, i.e., the attack vector is accurately identified by T-Pack.

Fig. 2.7 depicts distribution results without intrusion (A1) and a relatively intense attack (A6). We observe a slight overlap between the blue (no attack) and red (A6) curves ranging from 0.9-0.95 msecs, a data range covering less than 1% of the samples, i.e., more than 99% of the attacks are detected (F1 score of 0.993).

Fig. 2.8 depicts distributions without intrusion (A1) and a mild attack (A2). Results indicate a significant overlap between the blue (no attack) and red (A2) curves ranging from $\approx$ 0.65-0.85 msecs, a range with over 99% of the samples (F1 score of 0.014). This illustrates the limitations of

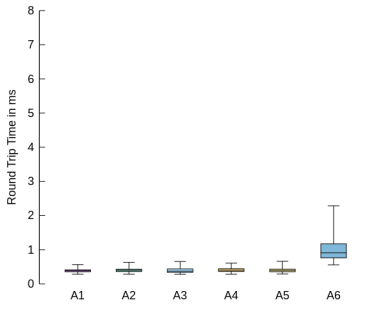


Figure 2.9 T-Pack measured ETT in ms for Paparazzi over UDP without IPSec under different attack scenarios.

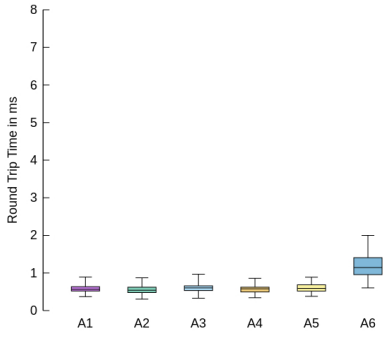


Figure 2.10 T-Pack measured ETT in ms for Paparazzi over UDP with IPSec under different attack scenarios.

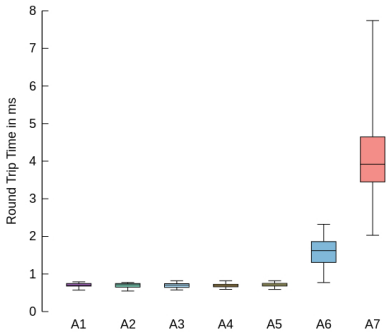


Figure 2.11 T-Pack measured RTT in ms for Drone under different attack scenarios.

T-Pack. Any attack with similar delays does not cause deadline misses. This illustrates why T-Pack *complements, but does not substitute* other security methods. In fact, T-Pack functions as expected: When deadlines are met, system functionality remains intact as sufficient network bandwidth remains available, and no intrusion is detected, but when attack intensity increases, intrusion is flagged requiring, e.g., Simplex mode changes.

Results in Fig. 2.5c reinforce our qualitative analysis of consistency, flexibility and integrity of T-Pack with IPSec from Experiment 1. Just as in Fig. 2.5b, we observe a similar trend in Fig. 2.5c in the distribution of RTT values when the Paparazzi model is subjected to different intensities of DDOS attacks. Absolute round-trip times are elevated by about 0.65ms. We observe that for higher intensity attacks (A7 in red), the RTT increases such that the entire box plot+whiskers range lies above the maximum RTT without attacks (A1 in purple). Also, ≈99% of the RTT values under A6 (blue) lie above the maximum RTT of A1.

Experiment 2b: UAV Paparazzi Subsystems under UDP

We next assess T-Pack under UDP for the UAV system by assessing packet ETT from one subsystem (AP) to another (GSC). Recall that unidirectional message transfer (UDP) requires a clock offset value between the two subsystems to calculate the ETT, which we implemented in analogy to the Network Time Protocol (NTP) within T-Pack. At system initialization, T-Pack measured RTT values of a UDP packet from AP to GSC. AP embeds timestamp information in the packet via T-Pack and sends it to GSC. GSC replies with another timestamped UDP packet plus its calculated time difference from the prior reception from AP. AP uses GSC's reply to calculate its offset to GSC using GSC's timestamp. In addition, the RTT is calculated just as for TCP using the time difference at AP between the reply from GSC and original send to GSC using AP's local clock. We experimentally verify that the RTT value closely approximates the one-way send overhead considering the clock offset between AP and GSC plus the one-way overhead plus offset for the reply. Subsequently, the clock offset is used to calculate ETT according to Eq. 2.1. Assuming loosely synchronized clocks, we expect a linear

increase in the clock offset due to clock drift. This is handled by periodic re-synchronization of clock offsets in our protocol.

Fig. 2.9 shows that the ETT increases slightly as we keep increasing the attack intensity from A1-A6. Compared to Fig. 2.5b, ETT values are close to half the values of the RTT for the corresponding attack vectors. T-Pack can detect 100% of the delay for attacks with A6 intensity as values of ETT under A6 are slightly higher than half the RTT in Fig. 2.5b. We were able to determine that this is due to a higher RTT value obtained due to increasing clock drift not reflected in offsets (before the next clock synchronization), which results in perceived increases in ETT. (Since A6 already shows 100% coverage for its attacks, A7 is omitted.) With intrusions as intense as A6 or more, attacks can be identified whereas low intensity attacks such A2 do not result in intrusion detection as the system meets all deadlines, just as for A2 in Fig. 2.5b. We also observe similar overhead when enabling IPSec for Paparazzi under UDP (Fig. 2.10) as seen previously under TCP (in Fig. 2.5c).

Discussion: IPSec encrypts the packet at the network layer including the T-Pack header. This cannot be achieved with SSL as encryption is realized above network layer. Additionally, T-Pack can incorporate SSL encryption alongside IPSec without any modification. T-Pack's ability to mesh with other security protocols underlines the advantages of its simplicity without any reliance on specialized hardware.

Experiment 3

For the drone-like multi-system, subsystems EPS and MMS are selected as sender and receiver, respectively. An attack model with the same attack parameters as in Experiment 2 (A2-A7) (see Fig. 2.5b) assesses performance and vulnerability of T-Pack.

Fig 2.11 depicts the RTT as box plots (y-axis) over DDOS attack intensities (x-axis). The outliers correspond to 0.5% of the total data. As attack intensity increases, a slight increase in RTT is seen — until a sudden and significant increase (A6 and A7) under more intense attacks. Under increasing attack, all RTT values eventually exceed those without attack, at which point 100% of attacks are detected with T-Pack at A7.

In summary, we experimentally demonstrated consistant behavior of T-Pack for different real-time applications and varying attack intensities for DDOS ping-of-death scenarios that affect network delays, ranging from tolerated low intensity attacks without missed deadlines to high intensity attacks resulting in all deadlines being missed. Notably, deadline misses are detected early, at packet reception time (or, if a packet is omitted, at the latest scheduled reception time), which is well before a task's deadline — the earliest point of intrusion detection in the absence of T-Pack.

2.7 Related Work

Prior work exploited timing bounds derived from timing analysis of code to detect malware intrusion. Zimmer et al. [Zim10] developed techniques to provide micro-timings for multiple granularity levels of application code. They implemented a set of timed analysis methods, T-Rex, T-Prot and T-Axt,

which demonstrated an advantage of timed analysis of code execution in constraining the window of vulnerability for code injections, from usually tens of millions of cycles down to tens, hundreds, or thousands of cycles, depending on the respective protection technique. In contrast, our work focuses on network protection.

Cyber-physical control systems subject to real-time constraints are vulnerable to malware intrusion over the network. Prior work [Che11; Kos10; Che18] demonstrated the viability of attacks on the network of a real-time system and uncovered potential damages. Our work proposes to mitigate damages by detecting intrusion prior to such attacks using timed analysis of packets on the network, which establishes end-to-end packet delivery times allowing intrusions to be detected in a hard real-time systems, where the size of messages and time of data transmission can be bound a priori.

Time sensitive networking systems can be implemented by scheduling and traffic shaping using the IEEE standards for bridges (or switches), e.g., IEEE 802.1Qbv extensions, which could detect any abnormal flow of packets due to attacks on the network but lack any means to notify end nodes. T-Pack fills this gap as intrusion is detected on end systems, which can instantaneously transition into another mode using the Simplex architecture.

Both [Ann17] and [Nar18] detect attacks by analyzing time delays under secure clock synchronization. However, they propose to embed time information in packets measured at the time of transmission and received right at the end of the propagation using hardware timestamping (see [Ann17]), which requires such support in switches/routers thereby raising cost. Instead, T-Pack embeds time inside the packet within the kernel relying on hardware support.

The IEEE 1588 standard for a precision time protocol [IEE] calculates end-to-end time for clock synchronization by sending prior transmission time as a message payload in follow-up packets and sends burst of such packets, which — in contrast to T-Pack — would increases the number of packets and thereby reduce network bandwidth. The 1-step method of the IEEE 1588 standard reduces the packet burst duration on the network, however, unlike T-Pack, it utilizes hardware timestamping, which raises implementation cost.

2.8 Summary

This work extends the design and implementation of T-Pack, a network-level timed security method that monitors end-to-end response times of packet delivery in the Linux network stack to detect malware intrusion through time delays relative to a real-time schedule of expected packet reception times. Extending the foundation established in the author's prior MS thesis [Mit20], this work formalizes an attack model centered on early intrusion detection as follows. T-Pack raises exceptions at packet reception time or via a T-Pack specific timeout, which fires in advance of global timeouts and well before a task's deadline, leaving additional time for safe-mode transitions such as Simplex. A working UDP implementation with periodic clock synchronization, composition with IPSec under both TCP and UDP, and evaluation across client-server, Paparazzi UAV, and drone-like multi-

system scenarios demonstrate the method's applicability across transport protocols and under cryptographic integrity protection. Results indicate successful detection of malware intrusion in 95%-100% of cases during distributed denial of service attacks that induce time delays, quantified via F1 scores and an instruction-budget analysis of the undetected attack window. T-Pack incurs a small overhead relative to the range of delays for the attacks it protects against, combining efficiency and simplicity through its implementation in the Linux kernel without reliance on specialized hardware.

CHAPTER

3

T-TEX: TIMED THREADED EXECUTION FOR REAL-TIME SECURITY AND SAFETY

3.1 Introduction

The use of multi-cores has significantly increased in real-time systems to meet rapidly increasing requirements for high performance and low power consumption. In the past few years, there has been a stagnation in processor frequencies. However, this has been replaced by significant increases in the number of processing cores per chip. Sequential program performance no longer improves with newer processors, so real-time application developers must either arrange themselves with stagnating execution speeds or tackle the complexities of multi-core parallel programming [Vai10; Fer13].

OpenMP is a powerful and widely-used framework for parallel programming, integral to general-purpose applications, machine learning, and high-performance computing [Var15]. With the increasing complexity and time-critical demands of embedded and real-time systems, OpenMP has also emerged as a key framework for achieving reliable, high-performance parallelism in these domains [Sto13; Mar13; Cha09; Wan13; Bur13].

Real-time support in OpenMP is especially critical, as it extends the framework's utility to applications where timing is paramount. In these scenarios, predictable execution is essential, and parallel workloads must be managed in a way that guarantees timely task completion. Traditional OpenMP approaches using the `task` clause allow applications to define independent tasks that can run concurrently, scaling efficiently with the available cores [Wan20]. However, without real-time constraints, task performance is limited by the core count and lacks the predictability required by real-

time systems. To address these requirements, OpenMP-RT introduces a real-time scheduling layer, (1) enabling threads to operate with real-time priorities under fair core sharing and (2) supporting access to shared resources within real-time tasks and even for non-real-time tasks in a lock free and wait free manner [McD24]. This advancement is crucial, as it provides a pathway for OpenMP-based applications to meet timing constraints by controlling the execution order and priority of tasks while conforming to real-time scheduling paradigms.

This work addresses the critical issue of security in real-time systems that rely on implicit OpenMP parallelism, presenting a novel approach to intrusion detection. Real-time systems with high computational demands (e.g., video surveillance, computer vision, radar tracking, and hybrid real-time structural testing) often depend on parallel algorithms to meet stringent deadlines [Hua10]. Implicit parallelism, which enables these applications to efficiently leverage multicore processing, becomes a powerful tool in handling computationally intensive tasks under timing constraints (see Sect. 3.3). In today's technology landscape, system-level security is indispensable, protecting systems across multiple layers from attacks that target memory (e.g., buffer overflows [One96]), exploit vulnerabilities in value handling [Hor02], or disrupt network resources through denial-of-service. For real-time systems, one particularly severe threat is the delay attack, where adversaries induce timing delays in critical sections of code or network traffic associated with time-sensitive events. Such attacks can degrade system performance, and in control-based applications, the consequences of missed deadlines can range from environmental damage to life-threatening outcomes [Lou19].

The unique structure of real-time systems offers a crucial defense capability against such attacks, e.g., accurate knowledge of worst-case execution times (WCET) [Zim10], secure clock synchronization [Nar18], and protected critical infrastructure of smart grids [Ped16]. However, as real-time systems become more complex and interconnected, advanced methods are essential to detect subtle, timing-based threats, especially delay attacks.

Building on this concept, our work introduces **T-Tex**, an innovative approach that leverages timed security to detect delay attacks specifically on multi-core systems with OpenMP implicit parallelism. Unlike traditional methods, T-Tex leverages the timing consistency in real-time systems to provide fine-grained monitoring of task execution, accommodating preemption and resumption of threads within protected regions. By timing individual iterations of parallelized loops and applying a multi-phase security model, T-Tex offers robust intrusion detection that allows users to balance performance and security.

T-Tex makes several key contributions: (1) It introduces novel compilation techniques for timed instrumentation in Clang/LLVM [Cla]. (2) It employs a portable multi-phase profiling approach based on OMPT [Eic13]. (3) It modifies the Linux kernel to enable execution time monitoring per-thread, even across context switches. (4) Experimental results on a real platform demonstrate that T-Tex detects 100% of delay-based intrusions within a 60us vulnerability threshold, with an associated performance overhead of approximately 11% for less loop intensive benchmarks (parsec) and 72% for others (Daphne).

Overall, T-Tex provides time-based analysis in real-time systems as a defense mechanism against

delay attacks. What's more, T-Tex combines a pathway to enhanced, adaptive security with the ease of OpenMP parallel real-time programming, which is unprecedented.

3.2 Related Work

Previous studies have highlighted the impact of delay attacks on cyber-physical systems (CPS) that are subject to real-time constraints [Lou19]. Denial-of-service (DoS) attacks are among the most common types of attacks, which not only affect sequential code but also parallel processing on shared resources of multi-core systems [Bec22]. These attacks impact memory, process and task scheduling within a process of the system by over utilizing a shared resource, thereby delaying the execution of the processes. Past work [Bec19] demonstrates that modifications in how the resources are scheduled and utilized can prevent or mitigate the impact of these attacks.

Zimmer et al. [Zim10] developed techniques to provide micro-timings for multiple granularity levels of the application code. Techniques such as T-Rex, T-Prot and T-Axt, demonstrated an advantage of timed analysis of code execution in constraining the window of vulnerability for code injections within an application, from usually tens of millions of cycles down to tens, hundreds, or thousands of cycles, depending on the respective protection technique. T-Pack [Mit21] demonstrated how constraining the window of vulnerability for code injections allows intrusions to be detect for communication by timing the processing of individual packets.

T-Tex visions to extend these techniques to constrain the window of code execution in a multi-threaded real-time application and to time code regions executed by each thread for intrusion detection, even when threads execute in parallel on multi-cores. T-SYS [McD22] is the closest work to T-Tex, to the best of our knowledge. T-SYS analyzes execution time of code by injecting expected timeouts at the basic block level. However, T-SYS executes on a single core platform using a single-threaded model without any preemption. This eliminates the possibility of other applications running on the system or even multiple threads within the same application contending for execution on the same core. With preemption or parallel execution, this method would not work or result in looser bounds based on response time instead of WCET. Such loose bounds provide considerable slack to the attacker to mask potential code injections, which then remain undetected by response time monitoring. Subsequently, some region with a given deadline would remain unaware of on-going attacks. Basic Block level security lacks the ability to provide a smaller vulnerability window for code regions with a lower execution time than a single block, which results in a larger vulnerability window. T-Tex also provides a way to identify loop execution times by maintaining a counter for the number of iterations at run-time without splitting the loop. It increases the overhead by maintaining conditions at every iteration. However unlike T-SYS, it also provides a way to protect all types of loops instead of just simple "for loops" with trivially found inductions variable.

In summary, T-Tex implements delay attack detection in a multiprocessor using novel techniques to protect code regions within a given maximum vulnerability threshold (upper bound of the vul-

nerability window), carefully considering the above mentioned intricacies and also complementing T-SYS to secure parallel regions along with sequential regions of the code.

3.3 Assumptions and Attack Model

Prior work has shown that an over-subscription of user threads beyond the number of cores across multiple OpenMP applications may actually *decrease* the impact of co-runners and the performance variability [Ian10]. This technique can also significantly increase the system throughput. In a real-time system, where predictable execution is crucial, limiting the number of threads subscribed by OpenMP to the number of cores could be a feasible solution. However, this limits the application's ability to reap the performance benefits of oversubscription. [Zha09] demonstrates that thread migration within a real-time system incurs a non-negligible performance overhead, suggesting that binding threads to a core can help avoid non-deterministic migration times. OpenMP creates the desired number of threads at the beginning of a parallel region and maintains those threads in a pool until end of execution to reduce overhead [Had07].

Considering these characteristics of OpenMP, this work assumes a real-time OpenMP model similar to OpenMP-RT [McD24], where real-time priorities are assigned to the threads in this pool. Utilization of these threads requires setting the priority value of the underlying POSIX thread by modifying the "sched_param" structure [Nic96]. This enables OpenMP to execute each parallel region with a different thread priority, e.g., one from the real-time priority band of Linux per explicit task. Within such tasks, it supports parallel regions via implicit task executions at the same real-time priority as its surrounding parent task (see [McD24]), which real-time oblivious (regular) OpenMP implementations lack.

Our model supports oversubscription, i.e., an OpenMP application can run more threads than the number of cores and there can be more than one application running on the system, each at a different thread priority (which may or may not be protected by T-Tex, e.g., as for the green thread in Fig. 3.1). However, we assume that a scheduled thread only runs on the core it was originally bound to at the operating-system (OS) level to prevent any thread migration.

Multiple threads provide the ability to execute code regions (implicit tasks like OpenMP parallel region) in applications with different thread priorities resulting in contention of shared resources (e.g., shared caches) and preemptions, which are reflected in their response times. T-Tex implements a novel technique to identify these code regions, to break them down into much finer regions and to provide tight WCET monitoring of these regions by excluding delays due to shared resource contention or execution of other threads due to preemption.

Fig. 3.1 illustrates the idea of T-Tex. On the left, a protected application consists of red (serial) and blue (parallel) code regions, e.g., under OpenMP. T-Tex retains the objective of T-SYS [McD22] and utilizes it to protect the execution of serial code (red). What's more, T-Tex complements it to monitor the WCET of the blue code region executed by multiple threads. Yet, T-Tex eliminates the delays attributed to any third party, thereby providing tight monitoring of both red and blue code regions,

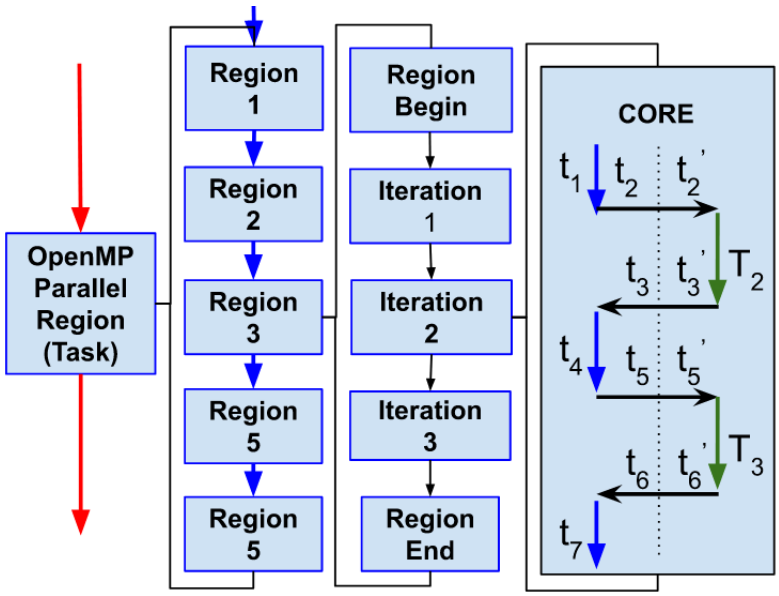


Figure 3.1 Analysis Model: Timed analysis of Parallel Region - broken into finer code regions - broken into loop iterations - analyzing context-switch time (t_1-t_7 represent the execution times of the regions they are marked alongside).

one of the capability that T-SYS was lacking. Blue code regions are further broken down to provide finer timing constraints by T-Tex using novel transformation techniques like loop monitoring. Fig. 3.1 demonstrates how Region 3 can be broken more finely into multiple iterations of a loop (assuming a loop within the code region), which provides a finer analysis for tighter security complimenting security technique in T-SYS. Contention due to other applications or threads shown in green (right side of Fig. 3.1) maintains fine-grained time inside the OS kernel between context switches. The execution time of iteration 3, broken from region 3 in this figure is evaluated as $t = \sum_{i=1}^{7} t_i$.

A potential delay attack in this scenario would be as follows: (1) A delay could occur due to context-switches when a higher priority task preempts. T-Tex maintains thread-specific times per thread, where a context switch changes time accounting within the OS kernel to a target thread and changes it back to the original thread when it resumes. Without our kernel changes, such time keeping and direct accounting of a thread's execution time in relation to WCET would not be feasible at fine granularity. In fact, longer execution of other accepted applications could further increase the response time of a protected thread, e.g., due to accessing shared data or resources resulting in extremely loose bounds on response time, which facilitates time-based attacks. In contrast, T-Tex neither restricts the number of context switches nor the priority of threads contenting with one another. (2) An attacker can schedule additional load (threads/applications) on the system delaying other scheduled work. The attacker can update the number of threads in the thread pool allowing a parallel region to execute with different number of threads than expected. However, the attacker cannot just add a new thread or remove one while executing a parallel region as threads are acquired from a thread pool at runtime by executing functions in the OpenMP runtime library. Nonetheless,

additional applications can be executed at any time. This could increase the number of context switches and thereby the response time of code regions, e.g, inflating t_2, t_3, t_5, t_6 (Fig. 3.1). Increased switches could still be detected by maintaining a counter tracking the context switch path but the impact on response time cannot be accurately determined by the OS. Increased load could also increase memory contention, thereby prolonging the latency of memory accesses without affecting the number of context switches. (3) The potential delays resulting from code injections, e.g., buffer overflow attacks, can be strategically coordinated to conceal the delay during run-time. T-Tex addresses this challenge by implementing time monitoring within the kernel, creating a secure space that remains isolated from potential attackers operating in user-space. This approach aims to mitigate the impact of code injection attacks on system performance while enhancing overall security.

We assume that the maximum workload of accepted applications on the processor is known a priori. This implies that one could bound the response time by bounding the number of context switches, even without T-Tex. But monitoring the WCET, which is tighter, only becomes feasible with T-Tex. We also assume that the attacker cannot modify the OS kernel space, interfere with scheduling or modify timers utilized for timed protection within the kernel.

3.4 Design

T-Tex is a novel method for verifying the execution times of POSIX threads in real-time OpenMP on multicore systems. It monitors specific code regions within an OpenMP application, capturing each thread's execution time in these regions and triggering a timeout if the region's WCET is exceeded. Although WCET values are experimentally determined in this work, T-Tex can also work with WCET bounds derived from static analysis tools.

3.4.1 Identifying Code Regions

3.4.1.1 OpenMP Regions

Timed analysis in a multi-core system requires identification of code regions, distinguishing them into parallel and sequential regions. Parallel regions within OpenMP are an implicit task. OpenMP-RT [McD24] provides the capability to execute these tasks via a real-time priority thread of any priority level (high or low). Parallel regions are further divided into sub-regions for a broader accepted range for a given maximum vulnerability threshold.

Past work with OpenMP has implemented various performance monitoring capabilities that make OpenMP execution events interposable, e.g., to create event tracing tools for finding inefficiencies in parallel regions. E.g., the OMPI and POMP tools [Moh02] allow programmers to specify and control instrumentation at well-defined OpenMP runtime events. OMPT, yet another tool, also provides (1) callbacks and inquiry functions that enable sampling-based performance tools to attribute application performance to complete calling contexts and (2) notifications for callback that enable construction for comprehensive monitoring [Eic13]. These OMPT callbacks are utilized

by T-Tex to identify the code regions depicted in Fig. 3.2 at runtime and also the instrumented function calls within these regions. OMPT helps identify parallel regions and threads executing it, however, runtime calls require additional information to accurately identify the sub-region/loop executed by a particular thread for isolated protection and evaluation (via LLVM passes).

All OpenMP directives within each parallel region are detected as sub-regions for the ease of use of the OpenMP profiler OMPT (see Sec. 3.5). The abstract syntax tree (AST) of the compiler is needed to verify these directives and relay the information to the compiler optimizer. The LLVM [Cla] compiler tool chain retrieves the AST via Clang and provides assistance in our code transformations to adapt region sizes and in instrumenting timer calls efficiently (LLVM pass).

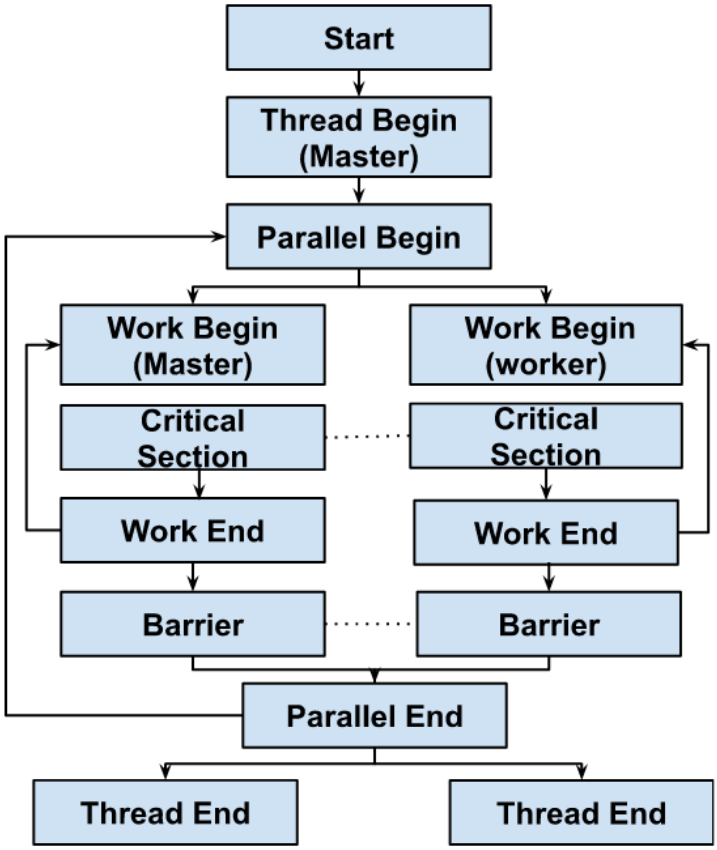


Figure 3.2 Code Regions for Implicit Parallelism in OpenMP.

3.4.1.2 Loop Detection in Parallel Regions of OpenMP

Loops can either be part of an OpenMP directive (executing sequentially or in parallel) or exist outside of these directives (executing sequentially). Loops within parallel regions are difficult to associate with specific sub-regions. T-Tex uses LLVM's loop detection techniques to identify such loops (see Sec. 3.5). Multi-processor scheduling accelerates loop iteration execution, allowing T-Tex

to combine some iterations, where timer instrumentation is provided only when needed to maintain the vulnerability window. This reduces the performance impact compared to instrumenting each iteration.

3.4.1.3 Monitoring Executing Threads

As discussed in Sec. 3.3, an attacker in the user space has the ability to modify the number of threads in the thread pool of an application. This allows the attacker to add malicious code and hide it from WCET monitoring. T-Tex evaluates the WCET of regions based on a given number of threads executing it. OpenMP also provides flexibility to users to execute code regions with variable number of threads, which would result in an incorrect timed analysis. To counter this problem, T-Tex constrains the number of threads executing a code region at the time of program analysis to a constant, which is strictly enforced during runtime as a modification to OpenMP.

3.4.2 Timers

T-Tex implements a multi-timer model (Appendix 3.8:3.8.1) to avoid synchronization overheads. Each thread's timer activates upon entering a protected code region (Sect. 3.3). T-Tex operates in multiple phases. In Phase 1, T-Tex determines WCETs for protected regions and feeds this data back to the compiler. Phase 1 uses broad bounds for code restructuring, resulting in low performance overhead but higher vulnerability. Phase 2 refines security by dividing code into finer regions based on a set threshold, creating tighter bounds. Additional restructuring iterations subsequently provide finer and more accurate time protection (Sect. 3.5). Each phase tracks execution times and maintains timers based on the WCET per region, resetting and recalibrating at a region's completion.

3.4.3 Accurate Analysis via Maintaining Timers at Context-Switch

In a multi-processor framework (see Sec. 3.3), multiple threads execute one or more applications known a priori while competing for a limited number of cores on the system.

Each user-space thread is mapped to a Linux kernel thread, and kernel scheduling may preempt a real-time application thread with another, higher-priority thread with similar protections (see Sec. 3.3). This preemption can arise from oversubscription or explicit core sharing across different priorities in multi-tasking real-time scheduling.

The execution time for each thread includes preemption time from context switches, which leads to a pessimistic WCET for identified code regions during static analysis in phase 1. T-Tex aims to perform finer execution time analysis of OpenMP code, but inflated timing without considering kernel preemptions would create extra slack, allowing attackers to mask delays and remain undetected. To address this, T-Tex modifies the Linux kernel to update timers during context switches, pausing a thread's timer when it's preempted and resuming it upon reactivation.

3.5 Implementation

Figure 4.3 shows the T-Tex workflow, which includes modifications to the Clang compiler, LLVM linker optimizations, runtime system updates (profiler and OpenMP runtime), an interpositioning framework, and T-Tex data structures. A new Linux kernel module manages timers and monitors context switches. Yellow boxes indicate code modifications while red boxes represent new modules, all contributing to the T-Tex framework. The blue and green arrows in Fig. 4.3 show T-Tex's two phases. In Phase 1, T-Tex uses code region data for automated dynamic timing analysis, determining the WCET for each region during parallel execution with prioritized threads. Timing information is collected at the region/sub-region level. For a more precise vulnerability window, the WCET values from this analysis are fed back into the T-Tex data structure for Phase 2 during linker optimization. Multiple iterations can refine the WCET values for timing analysis.

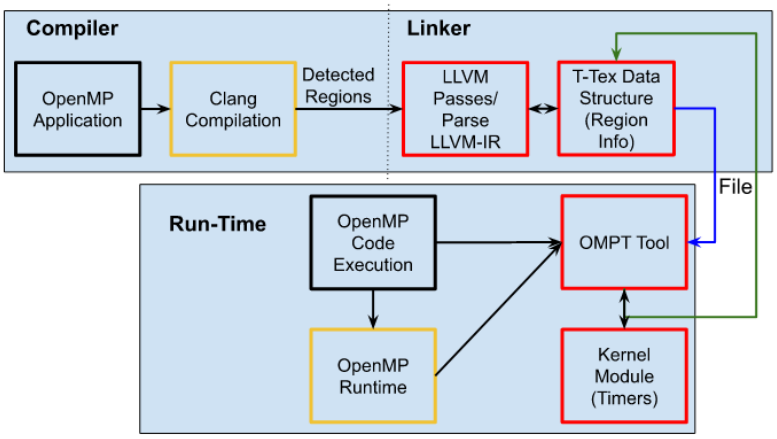


Figure 3.3 T-Tex Workflow.

3.5.1 Compiler Modifications: Clang Frontend

Clang [Cla] is a compiler front-end for C, C++ and other programming languages. It is used both for compilation and high-level language specifications, such as OpenMP. As shown in Fig. 4.3, T-Tex modifies Clang (yellow box—Clang Compilation) to (1) add an option to enable/disable T-Tex security, (2) traverse the application code to identify OpenMP regions using Clang's *RecursiveASTVisitor API*, mapping each region to a unique ID, and (3) relay this information to the compiler back-end (Detected Regions Line in Fig.4.3). T-Tex identifies all OpenMP parallel regions, using the region IDs to protect them from delay attacks. The vectors generated for all such parallel regions are stored as metadata (associated with the function created for parallel region by LLVM/Clang). Each value in the vector represents a reference ID to denote the respective OpenMP sub-regions (see Sec. 4.4). Consider Listing 3.1 (see Appendix). Clang generates two arrays, [2,-1] & [-1], to denote two paral-

lel regions starting with two **omp sections** followed by two **omp for** regions. The encoding of -1 represents an **omp for** whereas $n \geq 1$ represents n sections within the **omp section** scope. These IDs are not unique but rather reference IDs to identify the type of a sub-region. (3) Calls made to the OpenMP runtime library for these sub-regions are modified to accept an additional parameter, namely a unique ID. These IDs are passed as parameters for dynamic identification (see Sec. 4.4). In Listing 3.1, for the first parallel region, IDs 1,2 are fed to OpenMP runtime calls `Sections` and `For`, respectively. These IDs are subsequently used to identify regions within a `Parallel` section to arm timers in the respective callbacks (see Sec. 4.4 & Fig. 3.2).

Listing 3.1 OpenMP Sample Code.

```
#pragma omp parallel num_threads(4)
{
  #pragma omp sections nowait
  {
    #pragma omp section
    { code section 1; }
  #pragma omp section
    { code section 2; }
  #pragma omp parallel for num_threads(2)
    { code section 3; }
  #pragma omp for
    { code section 4; }
}
```

3.5.2 Compiler Modifications: Linker Optimization (LLVM Pass)

LLVM is a compiler toolchain used to develop front-ends and back-ends for various languages and architectures. T-Tex utilizes LLVM as the backend to process information from Clang. During the linker optimization phase, LLVM executes a pass on the global module, applying several T-Tex transformations. (1) The LLVM pass identifies all code regions and retrieves metadata for parallel regions/sub-regions initializing the T-Tex data structure. This information is stored in a 2-D vector and sent to the profiler tool. (2) OpenMP defines code regions in parallel constructs (Fig. 3.2) with two callbacks (start and end of a region). However, finer analysis is required to minimize slack time and prevent attacks from masking delays (Sec. 4.4). T-Tex refines these regions for more accurate WCET analysis by further breaking them down, including loops and sub-regions with custom callbacks. This analysis is integrated into the LLVM linker for a global view of the control flow supporting application code crossing multiple files.

3.5.2.1 Securing Loops within a Parallel Region

During the LLVM pass, T-Tex identifies parallel regions (by identifying wrapper functions) and their loops using LLVM's loop analysis pass, which does not consider nested loops. T-Tex enhances this pass to detect sub-loops up to n levels of nesting ($n \geq 1$). It also parses functions within these loops to find additional loops while skipping repeated loops (& functions) within a single parallel region, as they all adhere to the same WCET vulnerability (executed at the same priority).

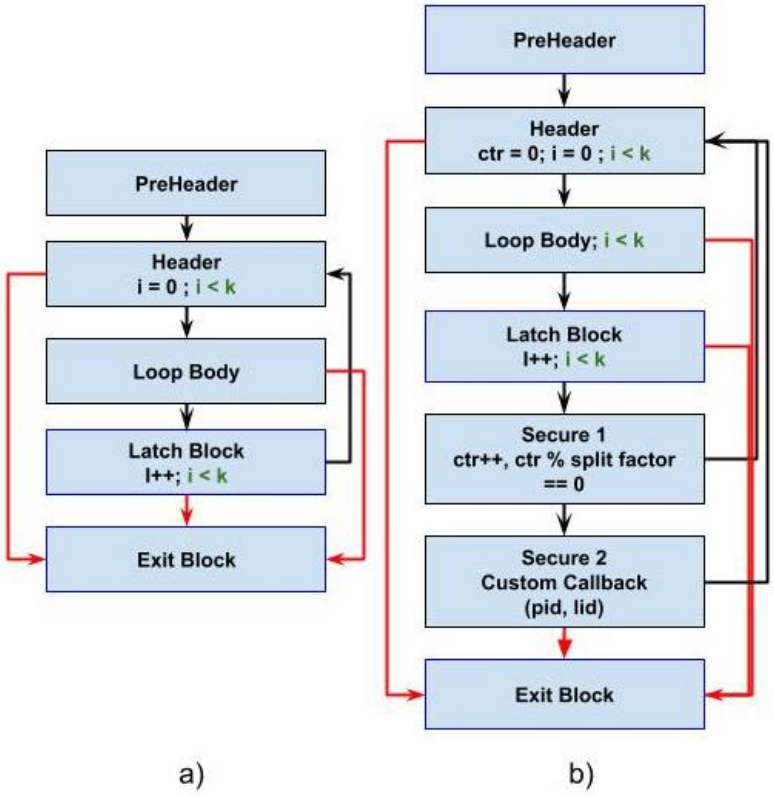


Figure 3.4 Loop Control Flow Graph: Red Arrow = possible edges, Black Arrow = 100% existing edge, Blue Box = one or more basic blocks, Green condition = may or may not be in the given block a) Loop before T-Tex; b) Loop after T-Tex.

In Phase 1, T-Tex instruments each loop with a custom callback called after every x-th iteration allowing the profiler to handle timers via the kernel module. Fig. 3.4 shows the updated loop control flow graph from a) to b). The header is updated with a phi-node (instruction used to select a value depending on the predecessor of the current basic block), which initializes a value ctr. For every incoming edge to the loop header, ctr is initialized with 0. In the latch block (where the back edge to the header originates), the conditional or unconditional jump to the header is updated to instead jump to an added block. This block retrieves the ctr value to check for x iterations (split), where on every x^{th} iteration, a custom call is executed. For nested loops, the first execution of the outermost loop is always timed to eliminate inner loop delays.

T-Tex handles all loops, regardless of optimizations, and eliminates the need for timer calls in each loop block if the iteration time is below the vulnerability threshold. If the time exceeds the threshold, x is reduced in subsequent phases. In case of a higher WCET with even a single iteration, T-Tex further secures the loop by splitting at the instruction level (instead of basic block) in later phases based on the vulnerability threshold. Similarly, other sub-regions outside the loops are identified and divided at instruction level.

3.5.3 T-Tex Data Structure and Subsequent Phases

The profiler needs information about parallel region identification, loop splitting after x iterations, and sub-region IDs. T-Tex provides this via a 2-D vector for loops and sub-regions, with each entry containing WCET, parallel region ID, sub-region/loop ID, split factor (for loops), reference number (for sub-regions), and split metrics (for finer division) (see Listing 3.1 in Appendix 3.8).

The first level of the vector lists parallel regions, while the second level tracks sub-regions or loops (each with a separate 2-D vector). After Phase 1, these vectors are initialized with high WCET values from fine-grained profiling and stored in a file. Repeated loops or sequential code are recorded separately under their respective parallel regions. Each loop/sub-region is assigned a unique ID (e.g., a unique loop ID), which differs from both loop ID and sub-region ID for identification of the region invoked that resulted in the highest WCET of all invocations. This information helps with further splitting in subsequent phases. Call paths resulting in any lower WCET will automatically be accurately split based on given maximum vulnerability, albeit in a pessimistic way (i.e., more splits in the call path are inserted than potentially needed for this shorter WCET, but they are needed for the highest WCET).

If the WCET exceeds the vulnerability threshold, regions are split with custom callbacks. Each iteration of a loop is further divided based on the sequential split factor (seq_split), determined by the maximum vulnerability threshold. This ensures that instructions in the loop body are segmented into seq_split regions. Further splitting into finer regions does not add entries to the 2-D vector as regions within the same parallel region have similar WCETs. Sub-regions are split similarly. This process helps T-Tex protect both sequential and parallel code (which compliments T-SYS). Multiple phases refine the split structure, minimizing callback overhead once the desired protection level is achieved.

3.5.4 Runtime Modifications: OpenMP & OMPT

The OpenMP runtime library functions (see Fig. 4.3) are updated to accept region IDs as parameters, which are then passed as arguments to the OMPT callback functions in the profiler. This allows correlation of execution regions with time values in the T-Tex data structure.

The T-Tex data structure initialized by the LLVM pass is first parsed to extract the execution time per region. As seen in Fig. 4.3, executions of these regions are identified by triggering a callback function. To this end, T-Tex utilizes thread-specific data to store execution information along with thread IDs during OMPT sampling. Thread begin callbacks initialize unique thread data and IDs for each thread. The thread data here is a vector list, later used to store time information of each executed code region. Each thread also arms a timer in the respective thread context. These timers are created via ioctl calls to the kernel maintaining time within the kernel (for all the threads), both for protection and for updates along context switches inside the kernel (see Sec. 4.4). At the next callback, a thread cancels its timer, evaluates the executed time and stores it as the actual execution time of the previous region. This timekeeping is implemented as high-resolution timers (HRT) within Linux, which are also stopped/started again across preemptions, i.e., they have become

thread specific due to our modifications.

At the end of execution for all the threads, OMPT evaluates the WCET based on the recorded execution time values. Recorded IDs for the executed code are utilized to identify the regions for which the time is evaluated. For executed code regions with the same parallel region IDs and same sub-region IDs or loop IDs, the value with the highest execution time is considered as the WCET of the region and stored in the 2-D T-Tex data structure.

3.5.5 Kernel Implementation of T-Tex

T-Tex timers are implemented within the Linux kernel and are crucial for *accurate per-thread time monitoring and reducing exploitable slack during context switches* (Sect. 4.4). Preemption delays, caused by higher-priority tasks or interrupts, can be detected by monitoring context-switch counts. Timer crediting helps minimize the attack vulnerability window. Prior work [Zha06] suggests that interrupt accounting enables accurate timing analysis. Real-time monitoring of interrupts for response-time analysis complicates attack detection requiring prior knowledge of all interrupts and delays. It adds further overhead of storing interrupt execution time and adding it to the timer — further requiring information to identify the timer associated to the thread. Additional overheads would result in higher vulnerability windows for the attacker.

T-Tex utilizes a loadable kernel module in Linux to implement ioctl calls using a virtual device driver to establish user-to-kernel space communication. Three ioctl calls are established: (1) A call to setup timers for each new thread resulting in a time_set call establishing a unique timer for each thread (t_id); (2) a call to reset a timer, frequently made by T-Tex monitoring after completion of a region without WCET violation, to prepare for the next region; (3) a call to receive the current time from the kernel to evaluate execution time of the completed region; and (4) a call to evaluate execution time within the kernel.

Listing 3.2 (see Appendix) depicts the code of the OMPT profiler functions invoking the respective ioctl calls. *Create_timer* registers a thread ID from the user and initializes a timer data structure. This structure is populated by setting timer values and activating the timer. A hashmap is maintained to store all active timers across all threads (irrespective of the number of applications). Without loss of generality, T-Tex assumes a MaxThread value to create a hash map associated with a given hash function. Similarly, hashmaps are searched in case of a modified timer call (*mod_timer*) to retrieve the timer data structure of the thread and to set the modified timer value to true. Other calls are handled in a similar manner as demonstrated in Listing 3.2.

Listing 3.2 ioctl calls: kernel timer implementation.

```
ioctl call(ref):
  switch(ref):
    case "create timer":
      t_id <- copy_from_user()
      t_data <- thread data structure
      t_data <- {timer_value, time_set}
        map[t_id % MaxThread] <- t_data
```

```
case "mod timer" :
  time, t_id <- copy_from_user()
  t_data <- map[t_id % MaxThread]
  t_data.timer_val <- time
  t_data.timer_mod <- true

case "delete timer":
  tid <- copy_from_user()
  t_data.timer_set <- false
  delete map[t_id % MaxThread]

case "receive timer":
  time <- ktime_get_ns()
  time -> copy_to_user()

case "retrieve time"
  time_s <- copy_from_user()
  time_f <- ktime_get_ns()
  return ktime_sub(time_f,time_s)
```

Timer calls within the kernel acquire spin locks held until completion. Updates to this timer are constrained to the context switch code itself for a given target thread, thereby avoiding possible race conditions. When T-Tex sets the *modified timer* variable for the thread (to inform the context switch of a required update of the timer), the kernel stops the timer of the task being switched out and modifies the timer of the task being switched in by assigning more slack time (as an extension of the context switch code). Once the context switch updates the timer, it sets the *switch* variable of the thread timer data structure to true. This assures that the timer of target process of the context switch has been modified, where the value was modified before the task was switched out. However, if a task was switched out in the middle of the ioctl modify call, the timer would *not* be reset, in which case the *switch* variable of the timer data structure would be false. If it is false, T-Tex executes a **schedule()** kernel call to reschedule the task so as to reset the timer when the task is reactivated upon a subsequent context switch. Similarly, the T-Tex timer data structure in the kernel also maintains a pending timer. It is updated for a switched out task right after the timer is stopped. (In Linux, the only way to stop a kernel timer is by deleting/deactivating it). If the timer is not to be modified when switching back in, T-Tex re-assigns the pending time to the timer; otherwise, an updated time is calculated using the task's reactivation time. When a thread is deleted, a *delete timer* call is issued. Since, we assume each thread is bound to a core, we can issue the del_timer call without activity checking, i.e., the timer can no longer be active on any core.

3.6 Experiments and Evaluation

3.6.1 Experiments

Real-time systems, e.g., for autonomous vehicles, rely on real-time sensors to detect nearby objects to navigate safely and avoid collisions. Object detection is often performed through nearest neighbor evaluations, with the Iterative Closest Point (ICP) algorithm helping vehicles interpret surroundings accurately. The OpenMP API is essential in optimizing performance-critical sections of autonomous driving software on embedded systems. Platforms such as Autoware [Kat15] and Apollo AI [Zhu18]

illustrate OpenMP's role in autonomous driving and embedded systems.

We test T-Tex on the Daphne benchmarks [Som19], which mirror key modules of the Autoware platform, including PointsToImage and EuclideanClustering. Our focus is on EuclideanClustering, used for object detection — a time-sensitive task crucial for vehicle safety. T-Tex is assessed on the following aspects:

(1) Does T-Tex's multi-phase approach reduce the vulnerability window for code regions below the security threshold? (2) What is T-Tex's accuracy under various attack intensities simulating buffer overflow attacks? (3) How does T-Tex compare to basic block level security of T-SYS? (4) What is the performance overhead of T-Tex?

We further evaluate T-Tex on the freqmine application in the Parsec benchmark suite [Bie08]. Parsec is widely used to assess multi-threaded applications, and freqmine's data-parallel structure with medium-level granularity offers an ideal scenario to test T-Tex (see Tab. 3.1 in Appendix 3.8).

3.6.2 Evaluation

Figure 3.5 illustrates T-Tex's method of instrumenting timer calls by dividing code regions to keep execution times below the security threshold. This example assumes a 4-threaded application running on 2 cores (oversubscribed) of an x86_64 Intel i7-9750H processor at 2.60GHz with real-time thread capabilities (see "sched_param" in Sec. 3.3).

The results show that T-Tex's multi-phase approach reduces all code regions near the 60us security threshold, further dividing regions that exceed this limit. T-Tex does not aim to remove outliers with shorter-than-threshold times but rather reduces all those above the threshold. Outliers exceeding 60us (y-axis) are brought within the threshold for the Daphne benchmark running with 4 threads. This multi-phase refinement provides dynamic timed analysis and protection for code regions, achieving a level of timed security never achieved before to the best of our knowledge.

T-Tex maintains the execution time of each thread and eliminates any time spent between context switches (see Sect. 4.4). The experimental evaluation shows that ioctl calls (see Sect. 3.5) incur a varied overhead. Hence, a security threshold high enough is set to factor in this cost. Compared to prior state-of-the-art protection methods, T-Tex provides a much finer security threshold ($>\approx 10X$ finer than the most closely related T-SYS, which provides a $\approx$ 600us threshold).

Each phase of T-Tex security evaluates an estimated division of a code region higher than the threshold. Each division of the code region incurs an added overhead of the instrumented code for time keeping. This results in an increase in the total execution time of all the regions. However, since higher time values of certain regions reduce into 2 or more halves, this narrows the box plot whiskers by reducing the outliers within the box (see Fig 3.5). The median of the distribution increases because certain code regions are divided into finer sub-regions whenever a high WCET exceeds the threshold. This division accounts for cases where some regions, such as loops with many iterations, temporarily exceed the threshold, even though in other executions they fall well below it. This finer division adds a slight overhead to each region, which impacts the time-keeping metric for loops. As the code is further divided closer to the threshold, box plot size starts decreasing so that a few more values

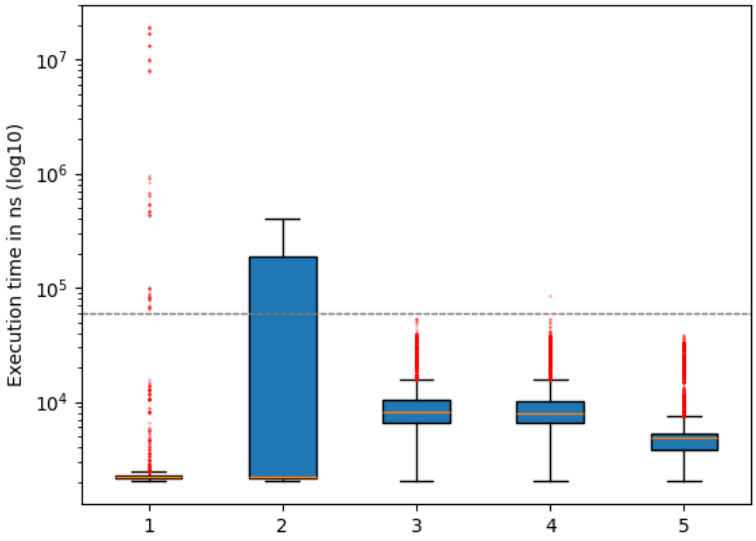


Figure 3.5 Phase 1-5 of T-Tex (further division of code regions to reduce WCET of each region) : Execution Time (ns) (black line: 60us security threshold).

become outliers. However, each of these regions still follows the security threshold for WCET. It is important to note that T-Tex ensures each region to execute within the defined threshold. However, for certain regions with consistently shorter execution times, T-Tex sets a WCET below this threshold, based on execution time evaluations performed during each phase of T-Tex security. This additional constraint further reduces the vulnerability window for potential attackers.

Fig. 3.6 assesses the accuracy and vulnerability of T-Tex to detect delay attacks within the code regions. To simulate delay attacks, spin delays are injected into each code region, which resemble a chain of delay attacks due to denial of service or buffer overflow attacks (Sect. 3.3). T-Tex secures the application by dividing it into finer code regions, detecting delays in each region independently. To provide maximum security within the given threshold, T-Tex effectively creates 100,000 monitored instances by analyzing 6 unique regions (see Sec. 4.4). To assess T-Tex's responsiveness to delays, we simulate an attack in each monitored instance. For example, a 40us delay attack would translate to a sequence of 100,000 individual 40us delays.

Fig. 3.6 shows box plots of execution time values for the Daphne benchmark protected by T-Tex and its distribution under various attack intensities. From the lower black/dotted line we can observe that 100% of the code regions protected via T-Tex will detect an attack delay of $\geq 40us$ seconds as none of the WCET values overlaps with the execution time distribution under the 40us attack intensity. Hence, even a single attack of 40us delay would be detected by T-Tex as no code region has a vulnerability threshold $\geq 40us$.

To assess the limits of T-Tex, we visually assess if an overlap exists between the execution times of T-Tex (no attack) and T-Tex under a 10us delay attack (see Fig. 3.7 in Appendix 3.8:**??**). From the prior Fig. 3.6, we observe overlapping outliers in test cases 1 with 2. Fig. 3.7 elaborates this using frequency distribution graphs (T-Tex in red and 10us delay in purple). 0.7% of the executed regions

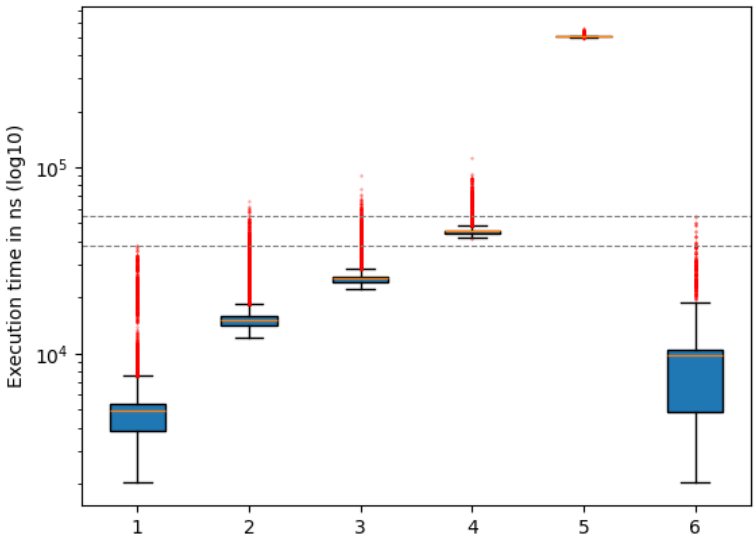


Figure 3.6 Attack Detection and Vulnerability 1) T-Tex (No Attack) 2) 10us Delay 3) 20us Delay 4) 40us Delay 5) 500us Delay 6) Basic Block Protection (T-SYS: No Attack).

by T-Tex overlap with the attack delay (see zoomed in image in Fig. 3.7: black dotted rectangle represents the zoomed region). If an attacker initiated a 10us delay within these regions, it would remain undetected. The WCET of these regions affects 22K out of the 100K regions analyzed by T-Tex (higher timer set). For an attacker to mask such a delay, one would have to predict these 22K regions to induce an attack. To take over an entire kernel, millions of instructions are typically required. We can limit an attack to $\approx$ 10K instructions here assuming, e.g., a CPU clock of 1GHz at an instruction per cycle rate of one. For an undetected intrusion under T-Tex, 100 such attacks of 10K instructions (adding up to 1M instructions) need to be executed given that the attacker can identify those 22K regions correctly. However, a subset of 22K regions may be close to the WCET, uncovering such an attack (as per-region prediction is infeasible at this fine granularity).

We compare T-Tex to the state of the art T-SYS. We partially replicate T-SYS by implementing code region distribution at a basic block granularity over instructions. However, we still identify all loops and provide iteration-based security, which T-SYS lacks. Code regions, which were cut at instruction level under T-Tex, are instead divided at basic block level for T-SYS resulting in *larger* slack for the attacker to hide the delay, i.e., allow longer injected attacks. Box plot 6 in Fig. 3.6 shows overlapping execution times for code regions protected by basic block protection compared to a 40us delay attack (upper black/dotted line). Fig. 3.7 further explains the overlapping outliers. 0.05% of the code regions in basic block protection (green), effecting the WCET of 800 regions in total, overlap with the delay attack (blue). In contrast, T-Tex (red) overlaps with the delay (blue). This demonstrates better accuracy for T-Tex resulting in 800 fewer attack vulnerability windows for same security threshold. We also observe from Figures 3.5 and 3.6 that basic block protection fails to constrain all outliers within the security threshold.

We demonstrate the effectiveness of kernel timer crediting in T-Tex by pausing the timer at each

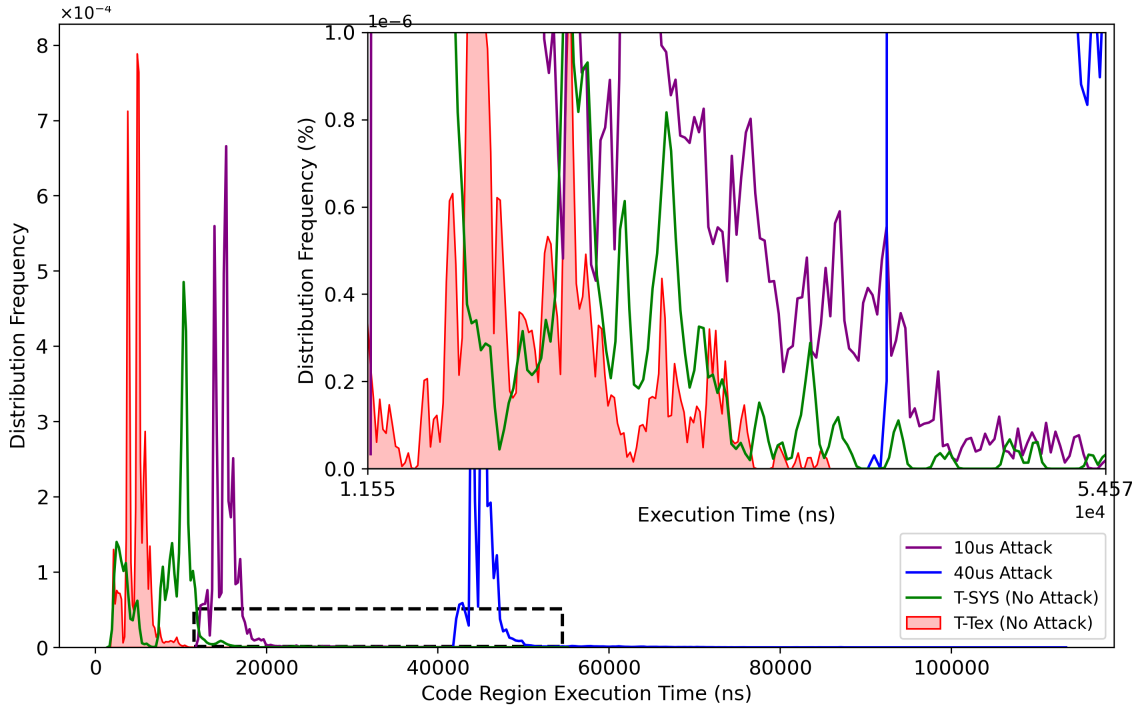


Figure 3.7 Time Distribution from Fig 3.6 — Image inside is a zoomed in image of the black rectangular box (Frequency Density of 0.001%): 1) 0.7% of red overlaps with purple 2) 0.05% of green overlaps with blue.

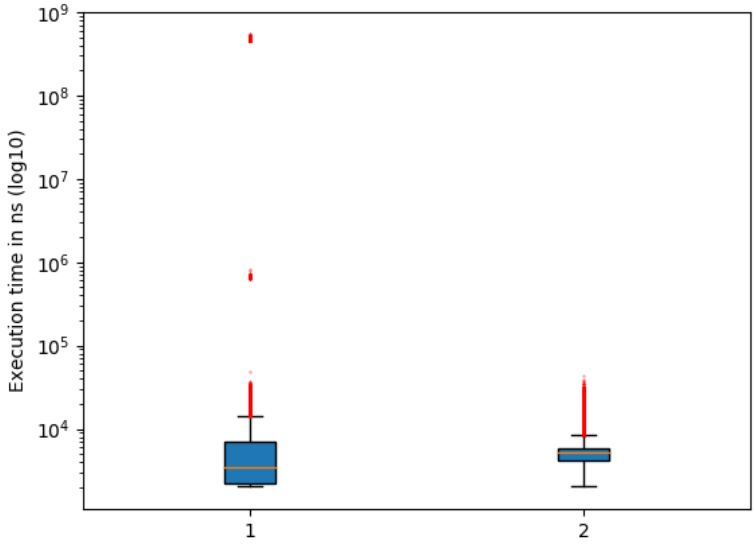


Figure 3.8 Benefit of Kernel Timer Crediting 1) T-Tex without kernel time crediting 2) T-Tex with kernel time crediting.

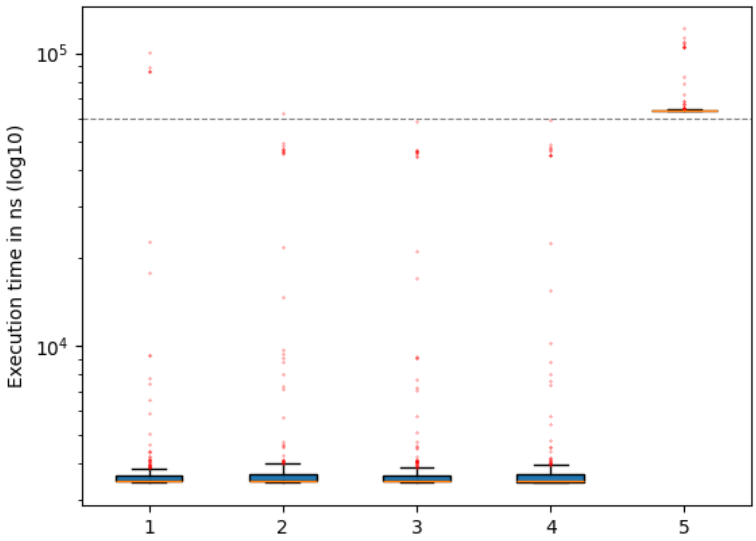


Figure 3.9 T-Tex security on parsec — frequency mining application: 1-4) Number of T-Tex iterations to reach security threshold of 60us 5) Attack: 60us Delay.

context switch. Running the Daphne benchmark alongside a high-priority task increases context switches, showcasing T-Tex's compatibility with tasks at varying priority levels. Figure 3.8 shows execution time distributions with T-Tex protection: (1) without kernel timer crediting and (2) with timer crediting. Red outliers in the non-credited case (at 10^6 and 10^9) indicate regions with higher WCETs, creating longer vulnerability windows where attackers could potentially evade detection.

We also analyze T-Tex's overhead for various security levels (Fig. 3.5). For maximum security (Phase 6), T-Tex incurs a 72% overhead but offers the option to adjust security via a compiler argument. Phase 2 security, with a 35% overhead, limits execution times to ≈ 400us per code region, creating a higher vulnerability window. Nonetheless, T-Tex detects 100% of delay attacks with a 500us delay intensity, reducing the vulnerability window by 100us compared to T-SYS.

Figure 3.9 illustrates T-Tex's compatibility with various real-time multi-threaded applications. For the Parsec frequency mining application, T-Tex identifies 220 code regions to maintain a security threshold of 60us, with fewer nested loops contributing to a lower 11% performance overhead. This setup achieves 100% detection accuracy even for a single 60us delay attack. *We conclude that T-Tex incurs only a marginal overhead for less loop intensive applications.*

3.7 Summary

This work presents the design and implementation of T-Tex, a novel timed analysis and attack detection technique for real-time multiprocessor applications using OpenMP. T-Tex leverages the LLVM compiler and OMPT profiler to implement a multi-timer approach that supports dynamic timed analysis. It successfully detects 100% of attacks by segmenting code regions to keep execution times below delay intensities of an unprecedented 40–60us, with an overhead of 11% – 72% for

benchmark applications. Additionally, a new kernel time-crediting technique reduces false positives, thereby improving accuracy.

3.8 Supplementary Information

3.8.1 Multiple Timers (T-Tex) vs. Single Timers

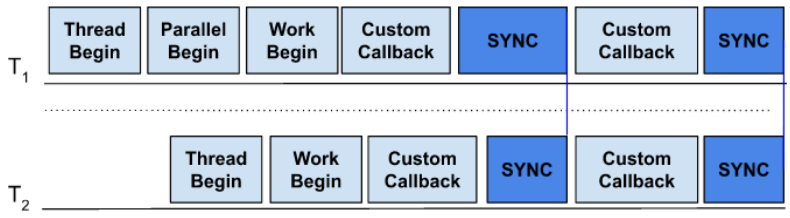


Figure 3.10 Single Timer: Custom Callbacks are instrumented timer calls for further dividing OpenMP recognized regions for finer analysis. More synchronizations needed for a smaller vulnerability window (resulting in more custom callbacks) defeating the purpose of multi-threaded model.

T-SYS features timed code executions within the kernel using a single timer for anomaly detection. A single timer in case of a uni-processor is feasible. However, implementing a single timer in a multi-processor for monitoring the execution of all the threads poses several challenges.

The implications of this approach are as follows. (1) A single timer under OpenMP and with multicores would require multiple barriers to synchronize each thread with the timer, leading to performance degradation and limiting the benefit of multi-threaded programming. As Fig. 3.10 shows, T_1 waits for T_2 to complete the Sync before executing the next code region (vertical blue line aligned at the end of the first Sync). (2) The OpenMP `nowait` clause allows threads to proceed beyond a sub-region within parallel region without a barrier. A single timer would either randomly indicate region termination for some thread or, if only the last thread resets the time, would cause the attack window of the next regions to increase (as it is not strictly timer protected) for all other threads. In contrast, multiple timers support detection of timing anomalies of OpenMP tasks for all these cases.

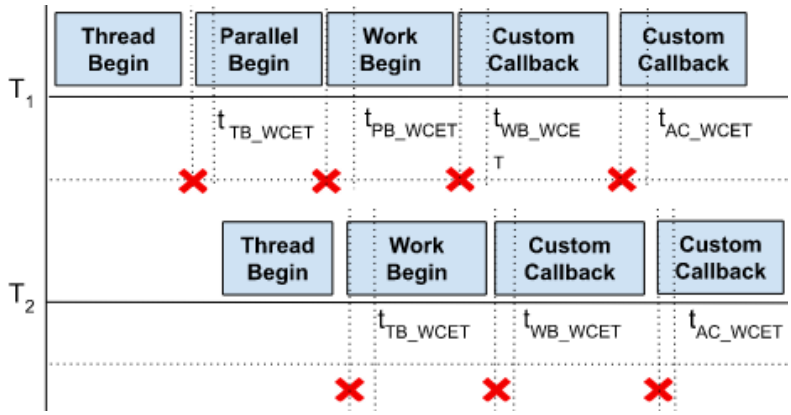


Figure 3.11 Multiple Timers: Timers armed at T_x are canceled (X) before being triggered, and a parallel region is partitioned into multiple sub-regions protected by finer-grained callbacks, which are not synchronized between threads until a final barrier.

Table 3.1 Parsec Benchmark Suite.

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

CHAPTER

4

OPENMP-RT-OFFLOAD: REAL-TIME SUPPORT FOR OPENMP GPU OFFLOADING

4.1 Introduction

The correctness of real-time systems depends not only on the input/output (I/O) correctness of computation, but also on timeliness, i.e., if results are produced by a given deadline. Such systems are expensive in development and certification as timing constraints need to be verified via extensive and detailed simulations as well as experiments. Minor changes within the system would result in another round of extensive testing. Different components within the system were extremely difficult to integrate such as multiprocessing capabilities required for computational scalability [Sta88]. However, with the rapid advancement in hardware technologies and schedulability theory, real-time systems increasingly execute on multicore/multiprocessor platforms under well-understood analysis. They can also be deployed in distributed environments using real-time middleware and deterministic networks to achieve end-to-end timing guarantees [Sor94; Liu73]. The use of multi-cores has significantly increased in real-time systems to meet rapidly increasing requirements for high performance computing (HPC) and low power consumption [Mit25a] along with the existing capabilities of timing and I/O correctness.

Given the rising demand of real-time systems for increased performance, there is a need to provide programming support and ease their integration into existing HPC development frameworks

while providing real-time guarantees. OpenMP is one such powerful and widely used framework for parallel programming, integral to general-purpose applications, machine learning and HPC [Jin11]. OpenMP `target` directives enable seamless offloading to conventional accelerators [Mit25b], such as GPUs [VL21], a cornerstone of HPC workloads and, lately, real-time systems [Ser18].

OpenMP benefits from real-time support because it extends the framework's utility to applications where timing is critical. In such settings, parallel workloads must be managed so that each thread meets execution-time guarantees. The OpenMP `task` clause allows applications to define independent, dynamically created tasks that run concurrently on available cores, reducing shared-resource contention [Wan20]. However, baseline tasking limits performance to the available core count and lacks predictability in the presence of task dependencies [Mit21]. OpenMP-RT [McD24] adds real-time capabilities to OpenMP by leveraging Linux real-time priority bands, where threads execute with appropriate priorities. It also provides lock-free and wait-free protection of critical sections, offering timing guarantees for independent threads sharing a core and for dependent tasks, respectively.

OpenMP-RT provides real-time capability on the CPU side. However, the increasing reliance on offloading computation to accelerators such as GPUs complicates timing guarantees. This is largely due to non-preemptible execution at the granularity of GPU kernels/threads. As a result, long GPU kernels and shared device resources can introduce unbounded priority inversion and interference.

To address this issue along with improving utilization of the accelerators (e.g., GPUs), vendors expose more flexible accelerator controls, including temporal preemption and spatial partitioning (e.g., NVIDIA's preemption beginning with the Pascal architecture and configurable modes on Tegra GPUs). However, preemption controls remain coarse because of limited priorities (support for two levels, high and low). Spatial partitioning, e.g., MIG (Multi-Instance GPU), offers strong spatial but no temporal isolation. It does not enable real-time scheduling within a partition and is also largely limited to data-center GPUs. Due to power and cost constraints, such GPUs are not feasible choices for embedded platforms.

Consequently, recent research has focused on software-based partitioning strategies (e.g., TimeWall [Ame21], HARD [Ni25], RTGPU [Zou23]). While effective at utilizing resources at a lower cost, these approaches share a critical flaw. They rely on a single priority queue on the CPU to manage GPU access. As we demonstrate in this work, a single queue creates a bottleneck, where a high-priority task is blocked not only by kernel execution but also by the sequential nature of memory copy operations (Host-to-Device). Even when a target partition is idle, the scheduler cannot dispatch the corresponding high-priority task if the head of the single command queue is occupied by a task stalling for a different partition.

We present **OpenMP-RT-Offload**, an application-transparent interposer that extends OpenMP-RT to support real-time GPU offloading. Unlike prior work, we introduce a **multi-queue, partition-aware scheduler** that decouples memory and kernel operations across disjoint groups of Streaming Multiprocessors (SMs).

Contributions:

- We propose a software-based spatial partitioning mechanism using NVIDIA Green Contexts (see 4.3) that works across the entire spectrum ranging from embedded to high-end GPUs without hardware modifications.
- We identify and resolve the single-queue bottleneck (see 4.4) prevalent in state-of-the-art schedulers by implementing per-partition and per-operation priority queues.
- We demonstrate that OpenMP-RT-Offload significantly reduces worst-case execution time (WCET) and jitter for high-priority real-time OpenMP tasks compared to standard schedulers.
- Task response time analysis indicates a 12-25 ms reduction in the worst-case response time for higher-priority tasks executed via OpenMP-RT-Offload compared to both the baseline and state-of-the-art approaches, demonstrating its effectiveness in curtailing unbounded delays.
- Evaluations reduces the miss rate of the highest priority task by $\approx$ 59% resulting in a 6% improvement for a weighted "schedulability metric" (designed to prioritize higher-priority task completions) for an eight-task set executing across two partitions compared to the baseline. Under the same task and partition configuration, the misses of the second-highest-priority task are completely eliminated. These improvements are driven by reduced queuing delays for higher-priority tasks and the elimination of head-of-line blocking through the use of multiple queues.
- Schedulability further increases by 10% for four tasks compared to the baseline when each task is executed in its own isolated partition while reducing the miss rate of the highest-priority task by $\approx$ 99%, which highlights the benefits of spatial isolation.

4.2 Related Work

Substantial work has been done to provide real-time guarantees for task execution within accelerator-assisted applications (HPC). TimeWall [Ame21] introduces a time-partitioning framework for multi-core and accelerator platforms. CPU cores receive periodic time slices for particular task execution, while global EDF manages access to shared resources (e.g., the accelerator) using priority-based locks, allowing only one task to execute on the accelerator at any given time. To prevent tasks from execution overrun on a non-preemptible accelerator, TimeWall introduces forbidden zones. If the execution time of a delayed task would overrun its time slice while executing on the accelerator, then that task is deferred to be executed in the next time slice. First, this approach is application intrusive requiring source code changes, Second, it creates a very pessimistic style of execution. Even though it provides real-time guarantees on the response time of the task on the accelerator by allowing a single task to execute on it at any given time, **it significantly affects performance, which is also a major contributing factor within HPC applications.** Deferring time critical task within embedded system could also be fatal in certain scenarios, e.g., for object detection in autonomous driving.

HARD [Ni25] utilizes partition-enabled GPUs to provide a more flexible strategy than TimeWall. It employs a similar technique of dividing task execution into time slots, prioritizing tasks with fewer remaining slots. GPU execution is realized spatially by assigning tasks to dedicated partitions, ensuring that no two tasks share the same partition to eliminate resource contention. HARD also provides critical proofs for WCET, effectively eliminating the scenarios that would otherwise require forbidden zones. However, HARD's reliance on hardware-supported partitioning (e.g., NVIDIA MIG) limits its applicability to high-end data center GPUs, making it unsuitable for cost-constrained embedded platforms. RTGPU [Zou23] addresses these limitations as they introduce partitioning via SM IDs and interleaved kernel execution using persistent threads. This permits spatial partitions on any GPU instead of limitations to only certain GPU models. RTGPU groups SMs into virtual SM groups (partitions) and executes each kernel on a separate group. However, if the utilization of a group is less than one, i.e., if an SM is not fully utilized, it executes another kernel within the same group but on a different SM. This results in higher utilization while avoiding resource contention.

Works like TimeWall, HARD, and RTGPU rely on a single priority queue to manage execution. As detailed in Section 4.1, a single queue introduces blocking due to memory calls and head-of-line blocking, where a high-priority task destined for an idle partition cannot be dispatched if the queue is stalled by a task waiting for a different, busy partition or by the kernel and memory execution of a lower-priority task. In contrast, OpenMP-RT-Offload introduces *Green Contexts* to enable software-defined partitioning on any NVIDIA GPU and utilizes a multi-queue dispatcher to eliminate blocking.

Our work focuses on OpenMP *tasks* as mentioned in Sect 4.1. OpenMP is one of the most widely used parallel programming paradigms. Substantial prior work has provided real-time guarantees for OpenMP task execution [Sun17]. McDonald et al. [McD24] introduced the **rt_tasks** directive, enabling flexible creation and execution of real-time tasks in OpenMP with timing guarantees on CPUs. However, OpenMP-RT does not extend these guarantees to the OpenMP *target* directive. OpenMP-RT-Offload fills this gap. To our knowledge, Cetre et al. [Cet24] is the only prior work addressing real-time capabilities for OpenMP offloading via event-based execution. Their approach modifies task execution to proceed via events. Events, however, are triggered only when dependencies are cleared, which can defer work indefinitely unless the dependency pattern is favorable. While this can reduce GPU response time, it may increase end-to-end completion time, which is a mismatch for time-critical workloads such as object detection in autonomous vehicles. OpenMP-RT-Offload is, to the best of our knowledge, the first work to provide fine-grained real-time execution and timing guarantees for OpenMP tasks offloaded to any NVIDIA GPUs via the *target* directive without any driver modifications or source code instrumentation.

4.3 Background

4.3.1 Executing Tasks with OpenMP-RT

OpenMP-RT-Offload builds on OpenMP-RT to execute real-time tasks. Listing 4.1 illustrates how periodic real-time tasks are executed with OpenMP-RT. The *rt_task* directive is added to the LLVM OpenMP specification and wraps its body into an OpenMP task, analogous to the explicit *task* directive. However, it executes the task on a periodically running real-time POSIX thread created to reside in the Linux real-time priority band. A user-provided configuration file specifies the task name and parameters (period, deadline, priority, etc.), which are used to assign these properties. A *name* clause is added to the *rt_task* directive. At compile time, it refers to the corresponding entry in the configuration file, from which the runtime retrieves the properties and schedules the task accordingly. OpenMP-RT demonstrates real-time guarantees for such tasks executing on multicore CPUs. However, it does not extend this capability to the OpenMP *target* directives. It is precisely this gap that OpenMP-RT-Offload fills.

Listing 4.1 OpenMP-RT.

```
#pragma omp rt_task name(A)
{
    execute_cpu_load();
    #pragma omp target
        execute_gpu_load();
}

#pragma omp rt_task name(B)
{
    execute_cpu_load();
    #pragma omp target
        execute_gpu_load();
}

#pragma omp rt_task name(C)
{
    execute_cpu_load();
    #pragma omp target
        execute_gpu_load();
}
```

4.3.2 NVIDIA GPUs in Embedded Systems

NVIDIA has played a significant role in bringing efficient CUDA-capable GPUs to embedded systems. Jetson/Tegra SoCs (and, for higher-end edge use cases, embedded RTX modules) deliver the performance needed for UAVs, autonomous ground vehicles, and service robots. These platforms support the standard CUDA software stack (CUDA Runtime/Driver APIs), cuBLAS/cuDNN/TensorRT, and expose practical power/thermal controls (e.g., DVFS and nvpmodel on Jetson) required for real-time, energy-constrained deployments. They are widely used in perception, localization, and planning pipelines, where accelerator offload is essential and timing/power budgets are tight.

4.3.2.1 Primary Context/ Streams

NVIDIA CUDA includes a device-side scheduler that arbitrates work across processes/contexts (e.g., multiple applications). Since OpenMP-RT-Offload targets a single OpenMP-RT application, it does not typically interact with inter-application scheduling. Within one application (i.e., one primary CUDA context), CUDA uses streams (per-context queues) in which operations are issued via an in-order policy. Operations in different streams may execute concurrently when resources allow. Historically, streams were priority-agnostic, but NVIDIA later added stream priorities (e.g., on embedded Jetson/Tegra platforms). Here, a higher-priority stream is preferred when the scheduler chooses among ready work, though running operations are not preempted. For utilization, CUDA can overlap compute and copies by using different streams that map to the compute engine and up to two copy engines, one for host-to device (H2D) and another for device-to-host (D2H) data movement. Scheduling within a given stream remains strictly in-order (first-in, first-out), and priority does not reorder already enqueued work. To change priority, new work must be submitted to a stream with the desired priority.

4.3.2.2 CUDA Green Context

NVIDIA CUDA, when used with the default primary context and streams, does not support explicit partitioning of GPU resources (e.g., across SMs). From HARD and RTGPU (Sect. 4.2), we know that GPU partitioning is valuable for real-time execution and efficient utilization. However, hardware partitioning such as MIG is generally unsuitable for embedded devices due to the power and cost implications of high-end devices supporting MIG. RTGPU suggests a software technique that leverages SM IDs to steer kernel execution to specific SMs, enabling effective partitioning in principle. That said, pinning work to a particular SM (or too narrow a SM subset) risks starvation/underutilization if that SM is not available, and it typically requires kernel instrumentation (e.g., SM-ID checks) and persistent-thread execution. These requirements are at odds with OpenMP offloading, where the *target* directive launches non-persistent kernels, and where the execution (selected via target teams distribute parallel for) automatically partitions work across teams and threads on a GPU. That compiler-driven distribution makes SM-ID steering even harder (and non-portable), since the mapping of iterations to specific SMs is neither exposed nor controllable by the programmer. The OpenMP model provides no standard mechanism for persistent kernels or SM-ID–based admission control.

To address this, we use NVIDIA green contexts, a lightweight alternative to traditional CUDA contexts that can be instantiated with an explicit set of device resources. This lets us define software spatial partitions of the GPU and target them using the standard CUDA programming abstractions (streams, kernel launches, etc.) [NVI]. In our flow, (i) we obtain an initial resource set (where SM resources are supported via cuDeviceGetDevResource), (ii) we subdivide that set using a partitioning API (e.g., cuDevSmResourceSplitByCount), (iii) we produce a resource descriptor with cuDevResourceGenerateDesc, and (iv) we provision the resources and create a green context with

cuGreenCtxCreate [NVI].

4.3.3 Dynamic Interpositioning of GPU Calls

On ELF-based systems, the dynamic linker honors the LD_PRELOAD environment to preload one or more shared objects before resolving symbols across an executable and its standard dynamic libraries in the library paths. If a preloaded library defines a function with the same symbol as a target API (e.g., cuLaunchKernel, cudaMemcpyAsync), the loader binds calls to the interposed version first. Inside the wrapper, we can embed our logic (e.g., enqueue into a partition queue, adjust priorities, record timestamps) and then forward to the real implementation via redirection to the original call (using the dynamic symbol resolution `dlsym(RTLD_NEXT, "symbol")`). This approach overrides behavior without a need to modify/recompile applications or the OpenMP runtime and works across the CUDA Driver/Runtime APIs.

4.3.4 RT-Mutex Design and Queuing

The Linux real-time patch from 2005, part of the Linux kernel since 6.12 versions (but generally back-patched to 5.10 for years), introduced priority-based locking [Ros06], commonly utilized for shared resources accessed by real-time tasks executing at different priority levels. Priority inheritance is employed to eliminate unbounded delays or the starvation of higher-priority tasks caused by waiting for a resource locked by a lower-priority task. To manage contention, Linux maintains a wait queue for each lock via a priority-sorted doubly linked list (plist). Unlike a standard FIFO queue, this structure ensures that the highest-priority task is always selected next, irrespective of the order of resource requests. We utilize this strict ordering property (see Sect. 4.4) to design our multi-queue approach.

4.4 Design

4.4.1 Task and Platform Model

We consider a set of periodic real-time tasks τ, where τ_i is the i-th task with period P_i, relative deadline D_i (we assume $D_i \leq T_i$), and worst-case execution times (WCETs) for distinct stages:

$$W_i: \quad \left(t_{i,1}^{\text{cpu}},\ t_i^{\text{h2d}},\ t_{i,2}^{\text{cpu}},\ t_{i,p}^{\text{gpu}},\ t_{i,3}^{\text{cpu}},\ t_i^{\text{d2h}},\ t_{i,4}^{\text{cpu}}\right)$$

where t_i^{h2d} and t_i^{d2h} are host⟷device copy WCETs, $t_{i,p}^{\text{gpu}}$ is the device-compute WCET of τ_i when executed on GPU partition p (modeled as non-preemptive stages), See Listing 4.2 (H2D, GPU_P, D2H). Tasks are scheduled on the CPU using OpenMP-RT for real-time threads with fixed-priority (POSIX FIFO) or deadline-based scheduled (EDF). Device and copy stages are scheduled by the queuing policy described next.

The variables $t_{i,1}^{\text{cpu}}$, $t_{i,2}^{\text{cpu}}$, $t_{i,3}^{\text{cpu}}$, and $t_{i,4}^{\text{cpu}}$ denote the *preemptible* host-side CPU execution segments demonstrated as CPU1, CPU2, CPU3 & CPU4 in listing 4.2 (e.g., initial data preparation, OpenMP/CUDA API overheads, post-processing and other overheads). Between these active CPU segments, the task offloads work to the accelerator stages (t_i^{h2d}, $t_{i,p}^{\text{gpu}}$, t_i^{d2h}), which are modeled as *non-preemptive* resources. During these accelerator stages, the host thread *suspends* (yields the CPU core). When an accelerator stage completes, the host thread becomes ready for executing CPU segments but may experience interference from higher-priority preemptive tasks. We analytically bound this interference by the host-side response time interference equation detailed below.

The GPU is spatially partitioned into P disjoint partitions $\{p = 1, \ldots, P\}$, with a fixed number of streaming multiprocessors (SMs) per partition S_p. In our canonical configuration (see Fig. 4.1), $P = 4$ with $\{S_p\} = \{4, 4, 8, 8\}$. Each task partition can execute more than one task (τ_i), unlike HARD [Ni25].

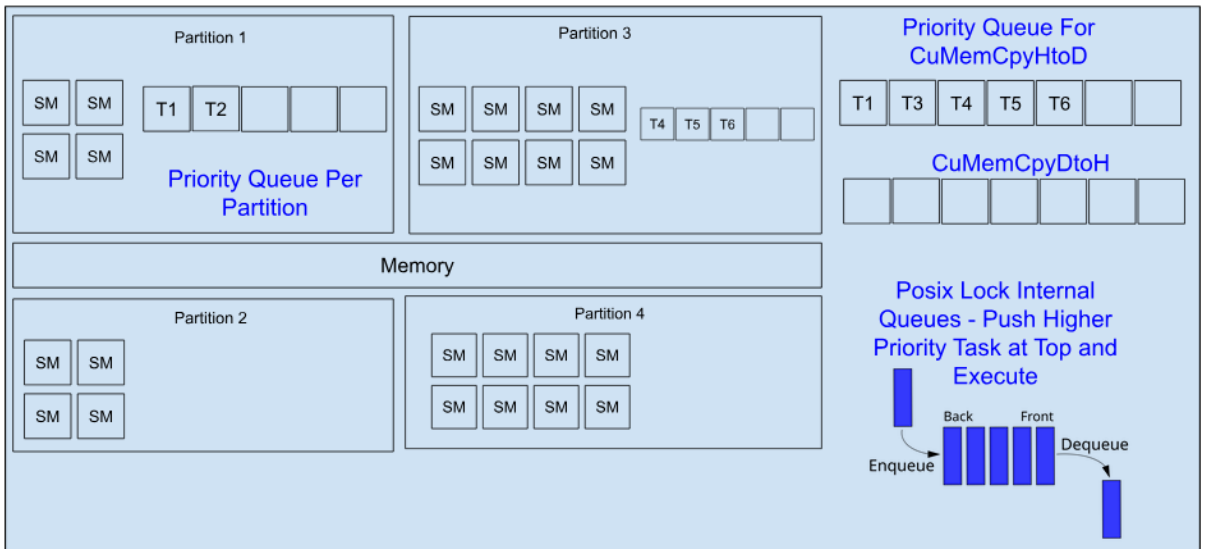


Figure 4.1 Partitioned GPU with per-partition queues and separate memory queues for H2D/D2H. Example shown: partitions with 4/4/8/8 SMs.

4.4.2 Queuing Abstraction

We maintain independent priority queues:

- one queue Q^{h2d} for H2D transfers,
- one queue Q^{d2h} for D2H transfers, and
- one queue Q^{p} per GPU partition p for device-compute work.

Each queue manages a series of *non-preemptive* jobs of tasks waiting for specific GPU resources (memory copies or kernel execution). Conceptually, dynamically reordering these jobs by priority

could be achieved using a standard heap data structure. This would bound the worst-case execution time for enqueue and dequeue operations to $O(\log n)$, where n is the number of waiting jobs.

Note: There is a need to utilize the native priority-ordered wait queue of a `pthread_mutex` configured with the `PTHREAD_PRIO_INHERIT` protocol. Since it is strictly necessary to prevent unbounded priority inversion when accessing shared GPU state, we use a priority-inheritance (PI) mutex (see Sect 4.3), where the underlying Linux kernel manages the prioritized arbitration under inheritance of blocked threads.

Service Policy and Key Property: Within any GPU queue (memory or partition), execution is non-preemptive. Hence, a higher-priority job can be delayed by at most *one* lower-priority job that was already executing when it arrived. Because each resource is modeled as an independent entity (no dependencies) with a strict priority queue and per-call reordering, no scenarios persist where multiple lower-priority jobs are at the head of the queue.

Lower Priority Blocking: For each queue $Q \in \{Q^{\mathrm{h2d}}, Q^{\mathrm{d2h}}, Q^{\mathrm{p}}\}$, a job of task τ_i can incur at most one blocking in a queue by any lower priority task than τ_i ($\mathrm{lp}_Q(i)$), irrespective of its priority. Hence, the blocking term is:

$$B_{i,k} = max_{j \in \mathrm{lp}_Q(i)}(t_j^{Q^k}) \tag{4.1}$$

4.4.3 Timing Analysis (RM)

Let $\mathrm{hp}_Q(i)$ denote the set of tasks (static priorities) with higher priority than τ_i that use queue Q.

Let the execution time of task τ_i be the sum of its CPU and GPU (the three H2D, GPU, D2H components)

$$C_i = \sum_{k \in \{1..4\}} t_{i,k}^{\mathrm{cpu}} + \sum_{k \in \{1..3\}} t_{i,k}^{\mathrm{gpu}}. \tag{4.2}$$

Let its blocking time of the same components and its respective queues be

$$B_i = \sum_{k \in \{1..3\}} B_{i,k}^{\mathrm{gpu}} \tag{4.3}$$

A task's response time R_i is then given by

$$R_i = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{4.4}$$

A task set is schedulable under fixed-priority scheduling if $R_i \leq D_i$ for all i.

Fig. 4.2 illustrates an execution timeline based on the task configuration of Table 4.1. This specific alignment is constructed to represent a near worst-case scenario for the analyzed task, τ_i, wherein almost every execution segment is subjected to severe preemption or blocking delays. The purpose of this trace is to empirically demonstrate how our proposed response time analysis (Eq. 4.4) safely upper-bounds such complex, compounding interference.

Table 4.1 Task Set Configuration - Multiple Interference.

Task	Prio	T	Release	Execution Time Arrays
τ_j (High)	99	28	2	$\mathbf{C} = [3,3,3,3]$ $\mathbf{G} = [4,6,4]$
τ_m (High)	85	52	12	$\mathbf{C} = [1,1,1,1]$ $\mathbf{G} = [1,5,1]$
τ_l (High)	75	76	0	$\mathbf{C} = [3,3,3,3]$ $\mathbf{G} = [3,5,3]$
τ_i (Observed)	30	T_i	0	$\mathbf{C} = [5,5,5,5]$ $\mathbf{G} = [6,12,6]$
τ_k (Low)	10	200	35	$\mathbf{C} = [1,1,1,1]$ $\mathbf{G} = [1,9,1]$
τ_{k2} (Low)	5	200	65	$\mathbf{C} = [1,1,1,1]$ $\mathbf{G} = [1,1,7]$

Note: Times in ms. $R_i = \sum C_i + \sum G_i + I_i + B_i = 20+24+41+5 = 90$ ms. I_i is the interference by high priority task on both CPU and GPU. C & G are the CPU and GPU execution times.

As shown in the timeline, all four CPU segments of τ_i (C_1 through C_4) suffer multiple direct preemptions from higher-priority tasks. Eq. 4.4 safely accounts for this scenario by aggregating the worst-case CPU demand of all higher-priority tasks that can be released within the response time window.

Furthermore, the timeline highlights the complex nature of GPU interference. The GPU kernel execution of τ_i suffers from a *transitive delay* (highlighted in light yellow): It is delayed by the high-priority task τ_j whose own kernel execution was previously blocked by the low-priority task τ_k. During this interval, τ_i is ready to execute but must wait for the sequential completion of both τ_k and τ_j. Our analysis explicitly captures this chained interference. It incorporates the execution delay from all higher-priority tasks. In addition, it bounding the maximum direct blocking caused by lower-priority tasks for each specific GPU segment (i.e. H2D, Kernel, and D2H) as defined in Eq. 4.1. Consequently, the direct blocking delay observed during τ_i's D2H segment (highlighted in dark yellow) — caused by the low-priority task τ_{k2} — is also successfully captured by the blocking term. Notice: Indirect/chain blocking is impossible since a GPU queue is protected by priority-ordered waits, where only one segment executes at a time, and nesting GPU calls (from within a GPU kernel) and not supported in OpenMP offloads.

Applying Eq. 4.4 to the configuration in Table 4.1 yields a theoretical worst-case response time for τ_i. This is calculated as the sum of its intrinsic execution time (44ms), the maximum per-segment GPU blocking (17ms), and the upper bound on higher-priority interference (60ms). The resulting analytical bound of 121ms safely encompasses the observed 90ms response time in the highly congested trace, demonstrating the safety and soundness of the proposed equation.

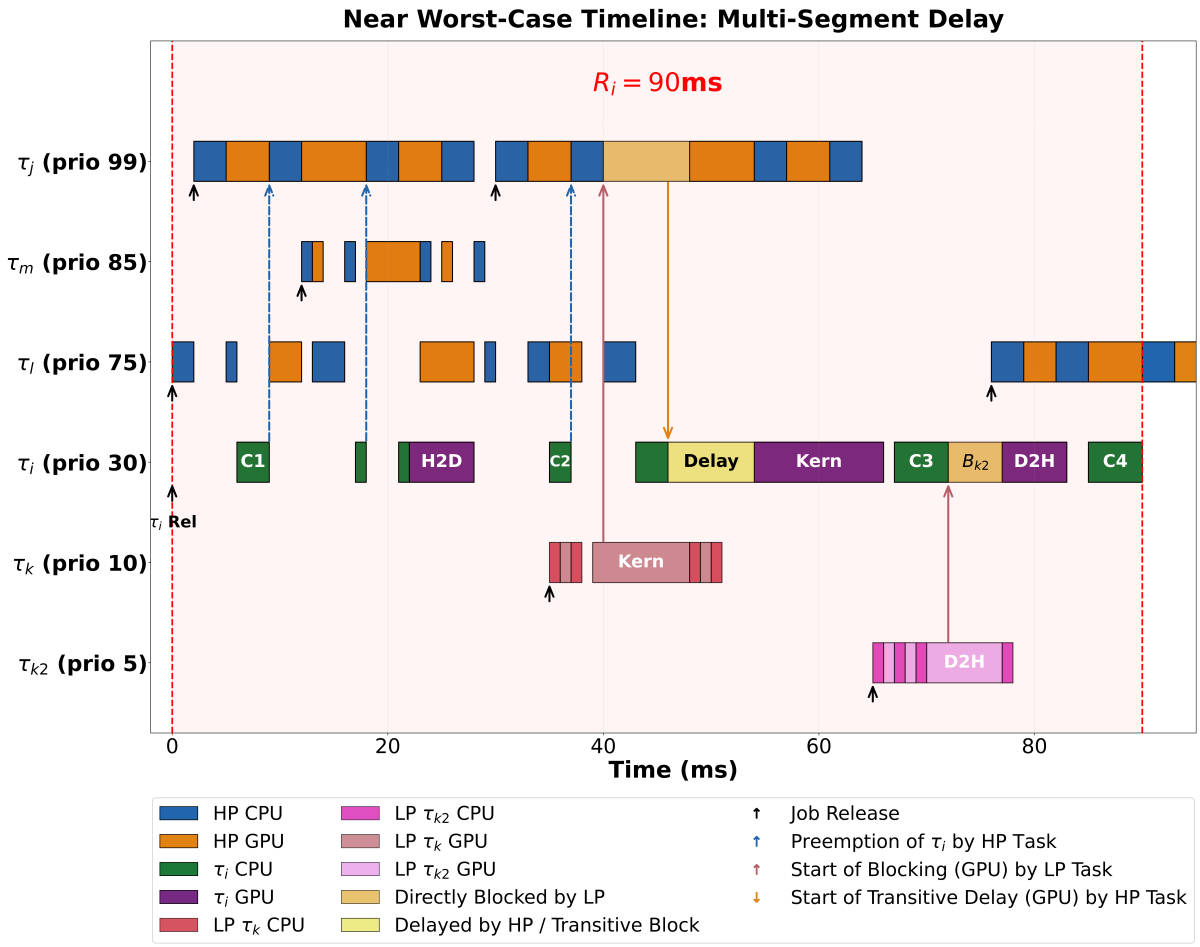


Figure 4.2 Execution Timeline for τ_i Along with HP (higher priority) and LP (lower priority) Tasks on single CPU and GPU (Different Queues For H2D, Kern, D2H). Each task runs a sequence of CPU-GPU as described in the task model. Legends in the image show what each colored box represents.

4.4.4 Timing Analysis (EDF)

Similar to the stack resource policy (SRP) protocol - Indirect/chain blocking is impossible in our task model. Hence, we analyze the schedulability of our model under EDF using the density bound with a blocking term [Zha13]. Given a taskset $\tau = \{\tau_1, \ldots, \tau_n\}$ ordered by non-decreasing period with implicit deadlines ($D_k = P_k$), the taskset is schedulable under EDF if for each $\tau_i \in \tau$ if

$$\sum_{k=1}^{i} \frac{C_k}{P_k} + \frac{B_i}{P_i} \leq 1, \tag{4.5}$$

where C_k is the total CPU execution time of task τ_k (Eq. 4.2), and B_i is the worst-case blocking time from non-preemptive GPU segments of lower-priority tasks (Eq. 4.3).

4.4.5 Per Resource Queue

The per-resource queue structure presented above directly enables the independent, per-stage timing bounds of the equations for the GPU suspension. Because each GPU partition and memory engine is managed by its own priority queue, interference terms in each equation are scoped to only those tasks sharing the same resource. This separation also extends to memory operations by maintaining distinct queues for Host-Device and Device-Host transfers independent of compute queues. A task waiting for a memory transfer does not block compute dispatch, and vice versa. In contrast, a single shared queue couples all tasks across all resources, allowing a task blocked on one partition to delay tasks targeting an entirely different and otherwise idle partition, i.e., a head-of-line blocking effect that our multi-queue design eliminates entirely. We quantify this cross-partition interference in Sect 4.7.

4.4.6 NVIDIA Write Lock Considerations

Listing 4.2 OpenMP-RT: Memory Decoupling.

```
#pragma omp parallel num_threads(4)
{
    #pragma omp sections
    {
        #pragma omp section
        {
            malloc(Input);
            malloc(Result);
            #pragma omp target enter data map(to: Input) map(alloc: Result)
            #pragma omp rt_task name(A)
            {
                execute_cpu_load(); //CPU1
                #pragma omp target data map(always, to: Input) map(always, from: Result) //H2D
                { //CPU2
                    #pragma omp target
                        execute_gpu_load(); //GPU_P
                    //CPU3
                } //D2H

                //CPU4
            }
            wait_task_exit_signal();
            #pragma omp target exit data map(release: Input, Result)
            free(Input);
            free(Result);
        }

        #pragma omp section
        {
            //launch another rt_task in parallel
        }
    }

}
```

NVIDIA's CUDA runtime acquires an internal write lock during memory allocation and de-allocation operations (`cudaMalloc`, `cudaFree`), which serializes all concurrent CUDA operations across the entire device. To avoid this serialization during periodic execution, OpenMP-RT-Offload requires that all device memory be pre-allocated before periodic task execution begins. Each task instance re-populates existing device buffers via asynchronous memory copies, which acquire only a lighter-weight read lock and can proceed concurrently. This constraint is enforced through

OpenMP's `target enter data` and `target exit data` directives, separating allocation from periodic execution.

Listing 4.2 illustrates this pattern. Device memory is allocated and mapped once via `target enter data` before the `rt_task` begins its periodic execution. Within each period, the `always` modifier on the `target data` directive forces re-transfer of input data and retrieval of results using the pre-allocated buffers, avoiding any write-lock acquisition. Device memory is released only after the task terminates via `target exit data`. This separates the allocation from periodic execution, ensuring that the write lock is never contended during steady-state operation.

4.5 Implementation

OpenMP-RT-Offload is implemented via dynamic interpositioning using LD_PRELOAD over the CUDA Driver and Runtime APIs. We do not patch, replace, or modify the CUDA driver. All intercepted calls are eventually forwarded to the vendor implementation via dlsym(RTLD_NEXT, ...) after our scheduling decisions.

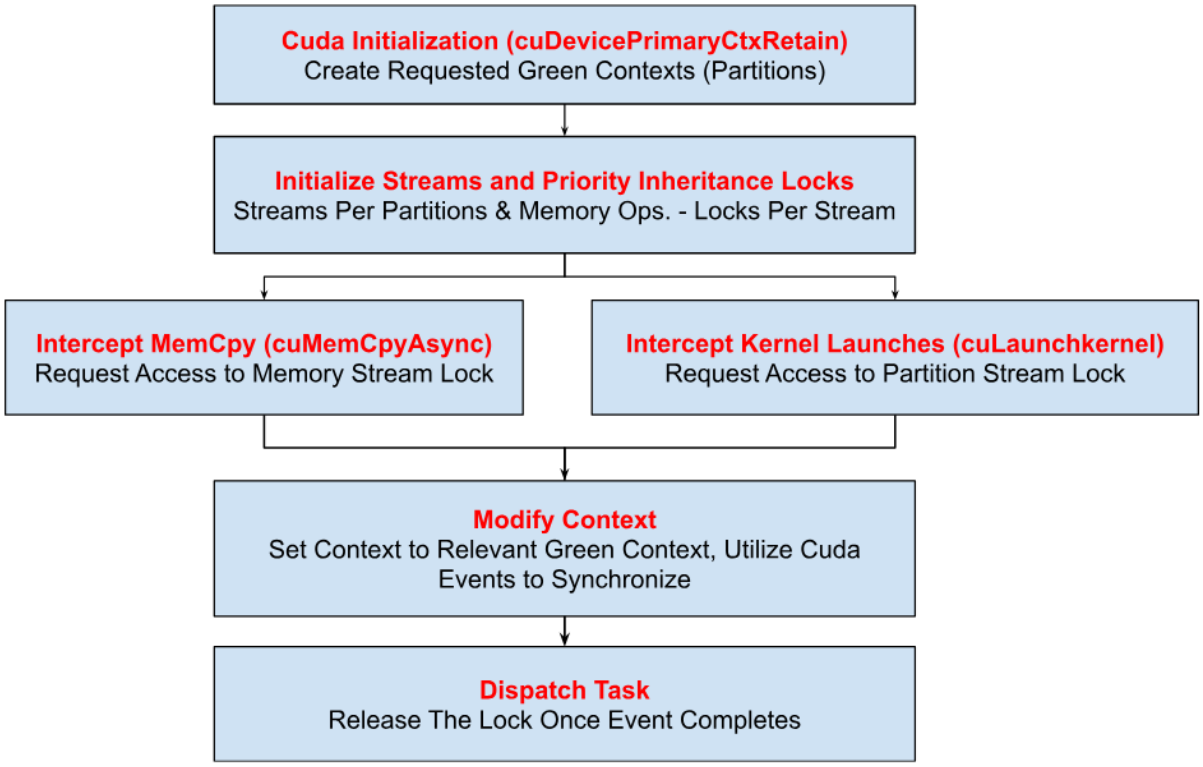


Figure 4.3 OpenMP-RT-Offload Pipeline.

Figure 4.3 depicts the pipeline of our OpenMP-RT-Offload implementation, where CUDA calls

are overridden with custom handlers that enforce priority-ordered scheduling before forwarding to the original CUDA API.

4.5.1 Initialization and Partitioning

Before CUDA initialization, we interpose on `cuDevicePrimaryCtxRetain` and, via `std::once`, load the user's task configuration (see Listing 4.3). The configuration specifies the GPU partitions to create, their SM allocations, and the mapping of tasks to partitions.

Based on this configuration, we form SM groups from the device's total SM count using the `cuDevSmResourceSplitByCount` API to obtain individual SM resource handles. These handles are combined according to the `sm` field in the configuration and used to create NVIDIA green contexts via `cuGreenCtxCreate`. For each green context, we create three dedicated streams using `cuGreenCtxStreamCreate`: one for kernel execution, one for host-to-device (H2D) transfers, and one for device-to-host (D2H) transfers. This mirrors the per-resource queue structure described in Section 4.4 and enables independent scheduling of compute and memory operations within each partition.

We then initialize the priority queues associated with each partition and memory resource. Each queue is implemented as a POSIX mutex configured with the `PTHREAD_PRIO_INHERIT` protocol, leveraging the kernel's priority-sorted wait queue to enforce strict priority ordering among contending tasks (Section 4.3).

Listing 4.3 Configuration File.

```
{
    "A" : {
        "priority": 99,
        "sm": 16,
        "period": 12,
        "deadline": 12,
        "phase": 0,
        "edf": 0,
        "green_ctx_id": 0
    },
    "B" : {
        "priority": 75,
        "sm": 20,
        "period": 25,
        "deadline": 25,
        "phase": 0,
        "edf": 0,
        "green_ctx_id": 1
    }
}
```

4.5.2 Task Identification

To route CUDA operations to the correct partition and priority queue, we must identify which OpenMP `rt_task` issued each intercepted CUDA API call. We implement this using a lightweight hook in the OpenMP-RT runtime. A function `unique_task()` is declared with the *weak* attribute and is invoked by the runtime immediately before a task begins its periodic execution. Because the symbol is weak, the OpenMP runtime calls it only when a definition is present. Our LD_PRELOAD

interposer provides that definition. When invoked, it receives the task's name (from the `name` clause in the `rt_task` directive, see Listing 4.1) and records a mapping from the calling thread's unique ID (obtained via the `gettid` system call) to the task's name and configuration in a hash map. Subsequently, when any intercepted CUDA API call is invoked, we retrieve the caller's thread ID, look up the corresponding task record, and obtain its priority and assigned green context. The call is then placed into the appropriate partition's priority queue, where ordering is enforced by the priority-inheritance mutex.

4.5.3 Interception and Dispatch

CUDA memory-copy and kernel-launch calls are intercepted to enforce priority-ordered scheduling. When an intercepted call arrives, the dispatcher identifies the calling task (via the thread ID mapping), determines the target green context and resource queue, and proceeds as follows:

1. The calling thread acquires the priority-inheritance mutex for the target resource (partition compute queue Q^{p}, H2D queue Q^{h2d}, or D2H queue Q^{d2h}). The kernel's priority-sorted wait queue ensures that the highest-priority waiting task is granted the lock next.

2. The current CUDA context is saved via `cuCtxGetCurrent` and switched to the target partition's green context using `cuCtxSetCurrent`.

3. A start event is recorded on the appropriate green context stream, the CUDA operation is submitted to that stream by forwarding to the original API via `dlsym(RTLD_NEXT, ...)`, and a stop event is recorded.

4. The thread synchronizes on the stop event using `cudaEventSynchronize`. Synchronizing on the specific event rather than the stream ensures we wait only for the submitted operation to complete without draining unrelated work on the same stream.

5. The mutex is released, allowing the next highest-priority waiting task to proceed. The original CUDA context is restored via `cuCtxSetCurrent`.

Because each partition and memory engine has its own mutex and stream, tasks on different partitions never block each other. A task waiting for Partition 1's compute queue does not prevent a task on Partition 2 from dispatching, eliminating the cross-partition head-of-line blocking inherent in single queue designs.

4.5.4 Single Queue Implementation For Comparison

For comparison, we also implement a single queue variant where all tasks across all partitions share a global priority queue implemented as a max-heap. Tasks acquire a global queue lock to insert into the heap, then attempt to acquire the target stream's mutex. If the stream is occupied, the task releases the queue lock, sleeps for a fixed control granularity interval, and retries — a polling

approach similar to HARD [Ni25]. Upon acquiring the stream mutex, the highest-priority task is extracted from the heap and dispatched. This design serves as a state-of-the-art implementation to demonstrate the cross-partition interference that our multi-queue approach eliminates.

4.6 Experimental Framework

We evaluate our system using a synthetic workload based on matrix multiplication, configured to replicate the workload characteristics of an autonomous driving system. Such systems typically comprise tasks with diverse criticality levels and execution profiles: a high-priority, short-duration task such as LiDAR-based distance estimation for collision avoidance, which demands the tightest response time guarantees, a medium-priority task such as object detection or tracking and a longer-running, lower-priority tasks such as point cloud processing or map updates, which operate over larger data sets but are less frequent.

We utilize OpenMP offloading with the OpenMP-RT task [McD24] to execute these tasks as periodic real-time GPU workloads. NVIDIA's internal write locks (discussed in Section 4.4) prevent concurrent CUDA memory allocation and deallocation with kernel execution, requiring us to decouple these phases. Memory is pre-allocated before the periodic execution begins, and each task instance re-populates the existing device buffers with new input data. This pattern is consistent with real sensor-driven pipelines, where periodic tasks repeatedly process new frames or point clouds using fixed-size buffers.

We map these workload characteristics to four synthetic tasks (A to D) with decreasing priority and increasing computational demand, as detailed in Table 4.2. Each task performs dense matrix multiplication of varying dimensions to produce execution times representative of the corresponding workload class.

Table 4.2 details the task set configuration. Tasks A and C share green-context, comprising of (16 SMs), while Tasks B and D share the other green-context isolating 20 SMs, enabling evaluation of both intra-partition priority scheduling and inter-partition isolation. To isolate GPU scheduling behavior, we execute on a host system with sufficient CPU cores to ensure no CPU-side resource contention. Our analysis therefore focuses exclusively on GPU resource management and execution dynamics. All tasks are scheduled under the POSIX `SCHED_FIFO` policy on the host, with static priorities as listed.

Table 4.2 Task Set Configurations for OpenMP-RT Synthetic Benchmarks.

Task	Priority	SMs	Period (ms)	Deadline (ms)	Phase	Ctx ID
A	99	16	12	12	0	1
B	75	20	25	25	0	2
C	50	16	50	50	0	1
D	10	20	50	50	0	2

The primary objective of these experiments is to evaluate the efficacy of the multi-queue technique within OpenMP-RT-Offload, leveraging NVIDIA Green Contexts for partition-based execution.

4.7 Results

4.7.1 Evaluating Isolated Execution

Table 4.3 reports isolated executions times for tasks A-D (See Table 4.2) under 3 execution models: the default GPU execution (Baseline), the single priority queue approach as demonstrated in HARD [Ni25] (Sect 4.2), and our multi-queue approach (Sect 4.4& 4.5). Each task is executed in isolation with no competing workloads, so these times reflect pure computation and scheduling overhead without any queuing delay.

As expected, execution time increases with matrix size across all configurations. The baseline, which utilizes all 36 SMs without partitioning, achieves the lowest execution times for every task. The partitioned configurations (16 and 20 SMs) exhibit moderately higher execution times due to the reduced SM count, with the scheduling overhead of both approaches contributing only 0.2-2 ms beyond the baseline (for given the matrix size).

The single queue and multi queue approaches show marginally different execution times. Our multi-queue approach avoids the explicit priority queue required by the single queue implementation, instead relying on the priority-ordered execution of POSIX mutexes configured with the `PTHREAD_PRIO_INHERIT` protocol. While the single queue's max-heap operations have $O(\log n)$ complexity, in isolation ($n = 1$) this overhead is negligible, explaining the near-identical performance of both approaches in the absence of contention.

Table 4.3 Isolated Task Execution Times (ms), No Contention.

Execution Model / Matrix	800^2	1000^2	1200^2	1400^2
Default (36 SMs)	3.601	6.879	11.439	14.635
Single (16 SMs)	3.893	–	12.396	–
Single (20 SMs)	–	7.543	–	16.115
Multi (16 SMs)	3.822	–	12.565	–
Multi (20 SMs)	–	7.348	–	16.793

Tasks A and C execute on 16 SM partitions (Ctx 1). Tasks B and D execute on 20 SM partitions (Ctx 2). Dashes indicate configurations not applicable to the given task-partition mapping. Matrix sizes for Tasks A, B, C, and D are 800^2, 1000^2, 1200^2, and 1400^2, respectively.

4.7.2 Response Time Analysis

We utilize these isolated execution times to design a real-time execution scenario (Table 4.2) with sufficient contention to expose the interference and unpredictability inherent in non-partitioned

default GPU execution. We compute the per-partition utilization to characterize the expected contention under each approach.

For the partitioned configurations, Ctx1 (Tasks A and C on 16 SMs), has a utilization of $U_1 = \frac{3.82}{12} + \frac{12.57}{50} \approx 0.57$, and Ctx 2 (Tasks B and D on 20 SMs) has $U_2 = \frac{7.35}{25} + \frac{16.79}{50} \approx 0.63$. Both partitions are comfortably below 1.0, ensuring schedulability while providing sufficient overlap for contention to occur in approximately 57-63% of scheduling windows. The single queue approach operates on the same SM partitions and therefore exhibits comparable per-partition utilization.

In contrast, under the default baseline where all four tasks share 36 SMs, the aggregate utilization is $U = \frac{3.60}{12} + \frac{6.88}{25} + \frac{11.44}{50} + \frac{14.64}{50} \approx 1.10$. Unlike CPU scheduling where utilization above 1.0 implies deadline misses, GPU execution co-schedules thread blocks from concurrent kernels across shared SMs. However, this co-scheduling causes mutual interference. High-priority tasks such as Task A will experience inflated and unpredictable execution times as their thread blocks compete for SM resources with lower-priority tasks. This motivates the need for SM partitioning to provide bounded, predictable execution for high-priority tasks.

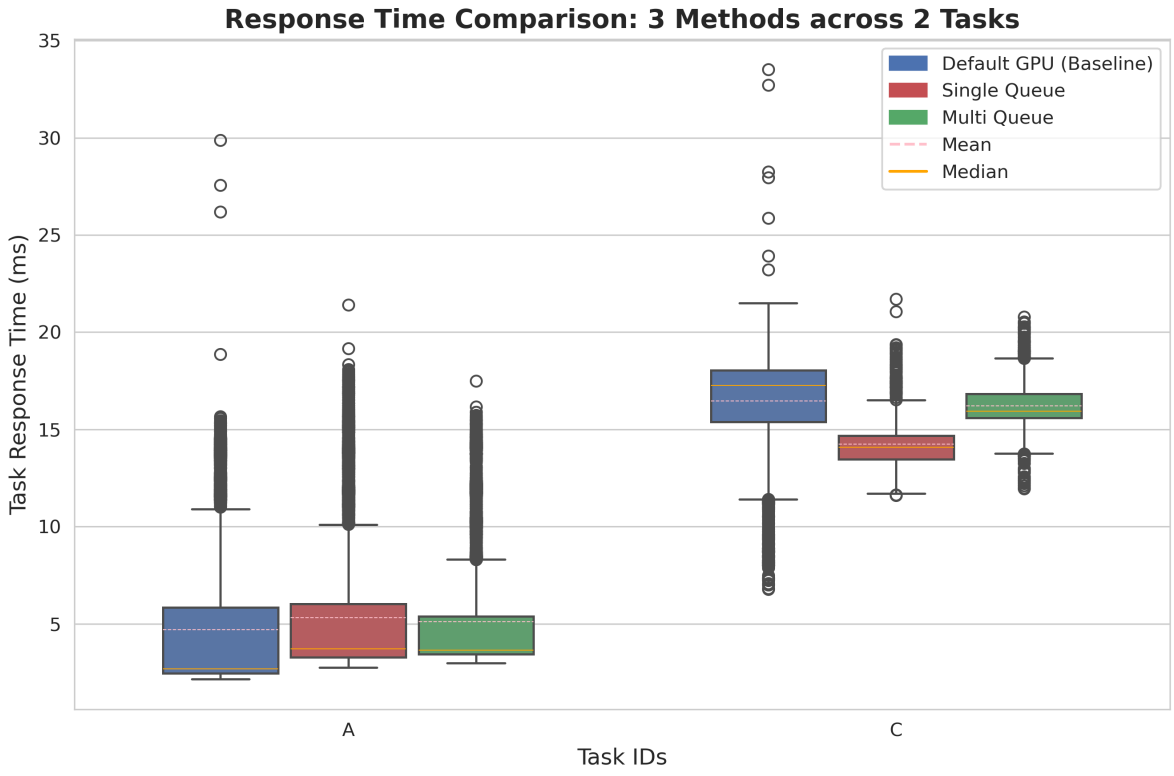


Figure 4.4 Comparison of Response times for Tasks A & C under the default baseline vs State-of-the-art (Single Queue) vs. OpenMP-RT Offload (Multi-Queue).

Figures 4.4 and 4.5 show task response times under the three execution models for the configuration in Table 4.2.

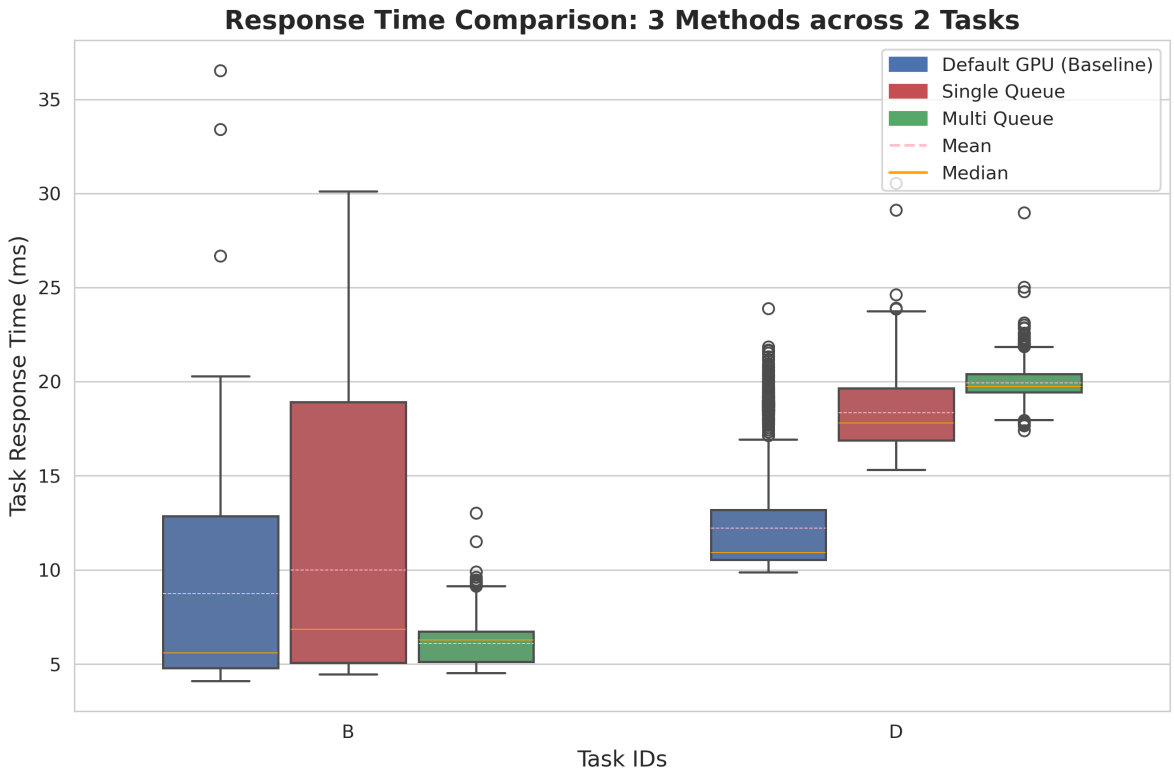


Figure 4.5 Comparison of Response times for Tasks B & D under the default baseline vs. State-of-the-art (Single Queue) vs. OpenMP-RT Offload (Multi-Queue).

4.7.2.0.1 Baseline Interference

As expected, the highest-priority Task A suffers significant execution time variability under baseline execution (blue box plot, Fig. 4.4). Despite an isolated execution time of ≈ 3.6ms (Table 4.3), the mean rises to ≈ 4.5ms under contention, with outliers reaching ≈ 30ms. These higher delays correspond to Task A competing for SM resources with concurrent lower-priority kernels. Given the matrix sizes and block dimensions used (256 threads per block), each task generates between 2,500 and 7,744 thread blocks. Since each SM can hold at most 6 resident blocks (limited by 1,536 threads per SM), even a single task requires multiple execution waves to complete. When all four tasks execute concurrently under the baseline, their thread blocks collectively saturate the 36 available SMs, causing mutual interference as blocks from different tasks compete for SM execution slots. The largest outliers of ≈ 30ms are consistent with the interference combination of Task C and Task B or Task C and Task D (see mean values for Task B, Task C and Task D in Fig 4.4&4.5). The smaller outliers reflect interference from a single co-running task. Since the baseline provides no priority enforcement on the GPU, these delays are *unbounded*, i.e., a high-priority task has no guarantee of preferential access to SM resources.

4.7.2.0.2 Single queue improvement and limitations

Both the single queue and multi queue approaches significantly reduce the response time variability of Task A (red and green box plots in Fig. 4.4), with response times bounded (by the multi-queue approach) as analyzed in Sect 4.4. However, the single queue approach exhibits poor performance for Task B (red box plot in Fig 4.5). Since all the partitions share a global priority queue, Task B cannot be dequeued while higher-priority Task A is pending in the ready queue, even when its SM partitions is independent of A and is available for execution. When Task A is blocked by Task C on the same partition, Task B in the single queue approach is delayed. Observed higher values of $\approx$ 30ms are a result of delay caused by pending Task A blocked by Task C. Since both partitioned approaches serialize kernel execution within each partition, the uncontested execution time for any task corresponds to its isolated execution time (Table 4.3) plus scheduling overhead, unlike the baseline where co-scheduling inflates all task's execution times. Isolated execution time of Task C of $\approx$ 12.4ms plus scheduling delays of Task A is the expected difference between average response time of Task B ($\approx$ 11ms) and the given outliers (red box plot for B in Fig 4.5).

4.7.2.0.3 Multi-queue isolation

Our multi-queue approach eliminates cross-partition interference entirely. Task B's outliers are significantly reduced compared to both baseline and single queue, since Task B only contends with a single run of Task D on its own partition and is never delayed by Task A or Task C. For Task A (green box plot in Fig 4.4), the worst-case response time of $\approx$ 17.5ms is $\approx$ 12ms above the mean. This is consistent with a single blocking period from Task C, whose isolated execution time on 16 SM's is $\approx$ 12.6ms (see Table 4.3). This confirms that under multi queue scheduling, the maximum interference experienced by a high-priority task is bounded by the worst-case execution time of the longest lower-priority task sharing the same partition, as formalized in Sect 4.4.

4.7.3 Weighted Schedulability Score

We define a weighted schedulability score to quantitatively compare the three execution models across varying system configurations. The weight of each task is defined as

$$w_i = \frac{p_i}{\sum_{\tau_j \in \tau} p_j}, \tag{4.6}$$

where p_i is the priority of task τ_i and p_j is the priority of all the task τ_j in the set of all scheduled tasks (τ). This ensures that deadline misses by higher-priority tasks incur a proportionally greater penalty in the overall score. The schedulability score for a given task set is then

$$SC = \sum_{\tau_i \in \tau} w_i \cdot \left(1 - \frac{m_i}{n_i}\right), \tag{4.7}$$

where m_i is the number of deadline misses for task τ_i across n_i total invocations. A perfect score

of 1.0 indicates that no task misses any deadline. The score degrades as miss rates increase, with misses by high-priority tasks penalized most heavily.

We evaluate the schedulability score for different different scheduling models (Multi-Q, Single-Q and Baseline (Default))

Table 4.4 Task Set Configurations for Weight Score Evaluation.

Task	Priority	Period (ms)	Deadline (ms)	Phase	Matrix Size
A	99	10	10	0	800^2
B	75	20	20	0	1000^2
C	50	35	35	0	1200^2
D	10	40	40	0	1400^2
E	30	40	40	0	900^2
F	5	65	65	0	1100^2
G	20	50	50	0	1100^2
H	3	80	80	0	1300^2

4.7.3.1 Varying Partition Count

Table 4.5 Deadline Miss Rates (%) – Varying Partition Count (4 Tasks).

Partitions	Model	Task A D=10ms	Task B D=20ms	Task C D=35ms	Task D D=40ms
1 (36 SMs)	Baseline	24.44	0.07	0.00	0.00
	Single Queue	6.2	2.1	0.00	0.00
	Multi Queue	0.89	0.00	0.00	0.00
2 (16+20 SMs)	Baseline	24.44	0.07	0.00	0.00
	Single Queue	5.05	4.42	0.00	0.00
	Multi Queue	3.78	0.08	0.00	0.00
4 (8+10 SMs)	Baseline	24.44	0.07	0.00	0.00
	Single Queue	0.01	0.00	0.00	0.00
	Multi Queue	0.01	0.00	0.00	0.00

Miss rate is computed as m_i/n_i where m_i is the number of invocations exceeding deadline D_i. Baseline data is identical across partition configurations as it does not utilize GPU partitioning.

Figure 4.6 shows the schedulability score for four tasks (A to D) as the number of GPU partitions increases from 1 to 4. This score is evaluated using the miss rates projected in Table 4.5. Task periods are set tighter than the configuration in Table 4.2 to increase GPU utilization and induce sufficient queuing contention for the deadline miss rate metric to differentiate between the three approaches. The tighter periods result in a baseline GPU utilization exceeding 1.0, ensuring that interference

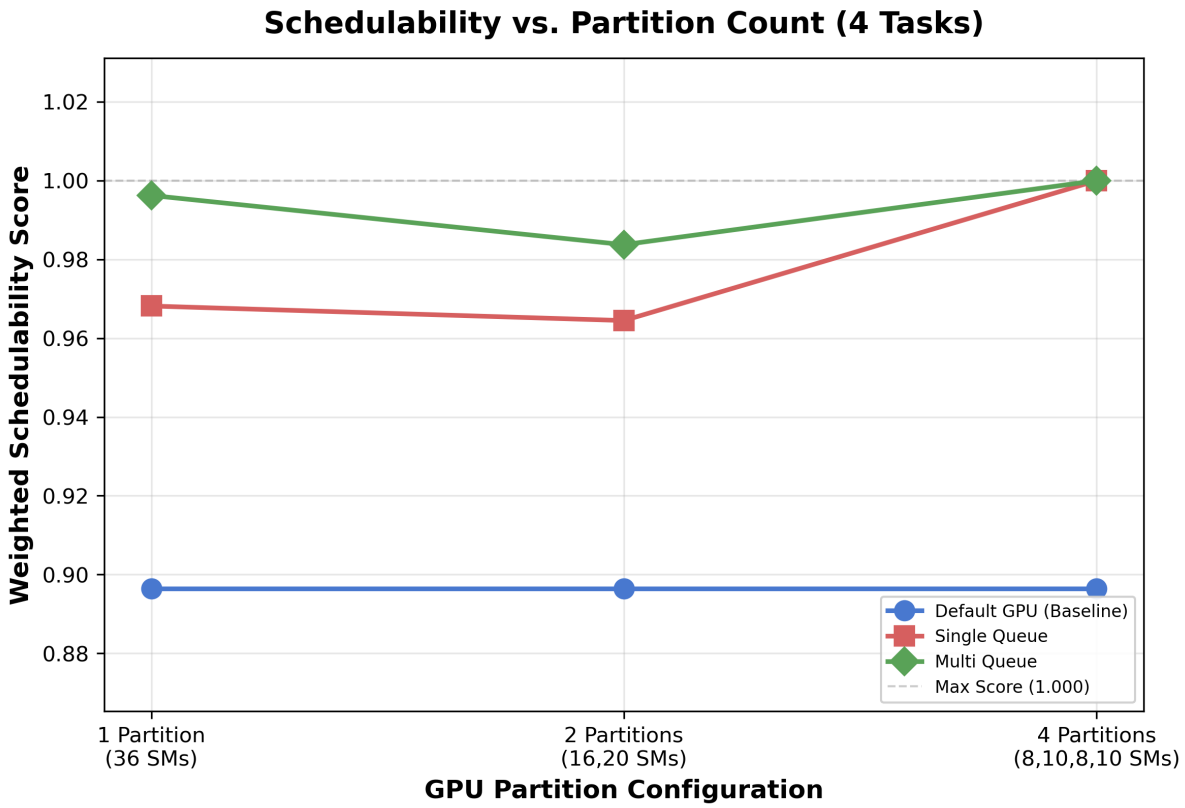


Figure 4.6 Weighted schedulability score for 4 tasks under varying GPU partition configurations: default baseline, single queue, and multi queue.

among tasks is frequent enough to expose scheduling quality differences. Task are scheduled using static priorities using the modified task configuration specified in Table 4.4.

4.7.3.1.1 1 Partition (36 SMs)

With a single partition, all tasks share all 36 SMs. In the baseline, there is no priority enforcement on the GPU, so a lower-priority task may execute while a higher-priority task is ready, leading to deadline misses for Tasks A and B. Both the single queue and multi queue approaches enforce priority-based scheduling, ensuring that a higher-priority task is blocked by at most one lower-priority task that is currently executing. With only one partition, both approaches behave similarly for kernel scheduling. However, single queue scores slightly lower than multi queue because memory operations share the same global priority queue as kernel launches. A lower-priority memory transfer (e.g., from Task C) can delay Task A's memory operation, which in turn blocks Task B from being dequeued for kernel execution, even though SMs are available and Task B's kernel could execute immediately. Multi queue maintains separate queues for memory and compute operations, avoiding such interference. Despite this difference, both approaches significantly outperform the baseline, where high-priority tasks experience unbounded interference from multiple concurrent lower-priority kernels competing for

SM resources without any priority enforcement.

4.7.3.1.2 2 Partitions (16 + 20 SMs)

With two partitions, Tasks A and C share Partition 1 (16 SMs) while Tasks B and D share Partition 2 (20 SMs). The baseline score remains unchanged, as it does not utilize green context partitioning. For the partitioned approaches, the interference pattern for the highest-priority task remains similar to the single-partition case: Task A can still be blocked by one lower-priority task (now only Task C) on its partition. However, we observe a slight drop in both single queue and multi queue scores compared to the single-partition configuration. This is because the reduced SM count per partition increases kernel execution times while the task periods remain unchanged, resulting in a higher fraction of invocations missing their deadlines. Multi queue continues to outperform single queue, as Task B benefits from partition isolation, i.e., it is never delayed by Task A's scheduling, unlike in the single queue where cross-partition head-of-line blocking persists. As shown in Table 4.5, Task B's miss rate reflects this blocking, where it is consistently higher in single queue while is remains near zero in multi queue. Task A's miss rate in single queue also exceeds that of multi queue. However, there is no fixed pattern to it as Task A does not suffer from head-of-line blocking but incurs additional scheduling overhead from the shared queue (control granularity and heap operations are some additional bounded delays compared to multi-queue). Despite these differences, both approaches significantly outperform the baseline, where high-priority tasks experience unbounded interference from multiple concurrent kernels competing for SM resources.

4.7.3.1.3 4 Partitions (8 + 10 SMs)

With four partitions, each task executes on its own dedicated partition, eliminating all inter-task kernel contention. Both single queue and multi queue achieve near-perfect scores, as the only remaining source of interference is the shared memory bottleneck (a single H2D and D2H copy engine). Since the memory transfer times for the matrix sizes used are small relative to the task periods, this overhead has a significantly low impact on deadline compliance. Although the reduced SM count per partition increases isolated execution times compared to the 2-partition configuration, this increase is marginal ($\approx$ 1ms). The elimination of inter-task blocking more than compensates for this overhead, resulting in higher scores than the 2-partition case. This confirms that the scheduling delays from shared-partition contention are a far greater source of deadline misses than the modest execution time increase from fewer SMs. The baseline remains unchanged at its lowest score, as it continues to execute without partitioning. This result demonstrates the value of increasing partition granularity. When each task is fully isolated, both scheduling approaches converge to near-ideal performance, while the unpartitioned baseline offers no such guarantee.

Table 4.6 Deadline Miss Rates (%) – Varying Task Count (2 Partitions).

Tasks	Model	A	B	C	D	E	F	G	H
2 (A, B)	Baseline	0	0	–	–	–	–	–	–
	Single-Q	0	0	–	–	–	–	–	–
	Multi-Q	0	0	–	–	–	–	–	–
4 (A–D)	Baseline	21.04	0	0	0	–	–	–	–
	Single-Q	4.93	1.6	0	0	–	–	–	–
	Multi-Q	5.05	0	0	0	–	–	–	–
6 (A–F)	Baseline	24.34	0	0	0	0	0	–	–
	Single-Q	8.83	2.1	0	0	0	0	–	–
	Multi-Q	6.42	0	0	0	0	0	–	–
8 (A–H)	Baseline	46.4	3	0	0	0	0	0	0
	Single-Q	39	14.4	0	2	0	0	0	0
	Multi-Q	19.2	0	0	2	0	0	0	0

Dashes indicate tasks not active in that configuration. Tasks A, C, E, G on Partition 1; Tasks B, D, F, H on Partition 2. The 1-task configuration is omitted (0% misses for all).

4.7.3.2 Varying Task Count

Figure 4.7 shows the schedulability score as the number of tasks increases from 1 to 8, with a fixed 2-partition configuration (16 + 20 SMs). Scores here are evaluated using the miss rate projected in Table 4.6.

4.7.3.2.1 1 and 2 Tasks

With a single task, all three models achieve a perfect score of 1.0 as there is no contention. With two tasks distributed across two partitions (Task A on Partition 1, Task B on Partition 2), contention remains negligible (single memory engine for H2D and D2H) for all approaches. Each task has exclusive access to its partition, and even the baseline experiences minimal interference between two moderately-sized kernels on 36 SMs.

4.7.3.2.2 4 Tasks

At four tasks (A+C on Partition 1, B+D on Partition 2), the results mirror the 2-partition configuration in Figure 4.6. The baseline drops sharply as four concurrent tasks without priority enforcement cause significant interference for high-priority Tasks A and B. Single queue outperforms the baseline by enforcing priority ordering, though Task B still suffers from head-of-line blocking when Task A is pending in the shared global queue. Multi queue achieves the highest score by eliminating cross-partition interference entirely.

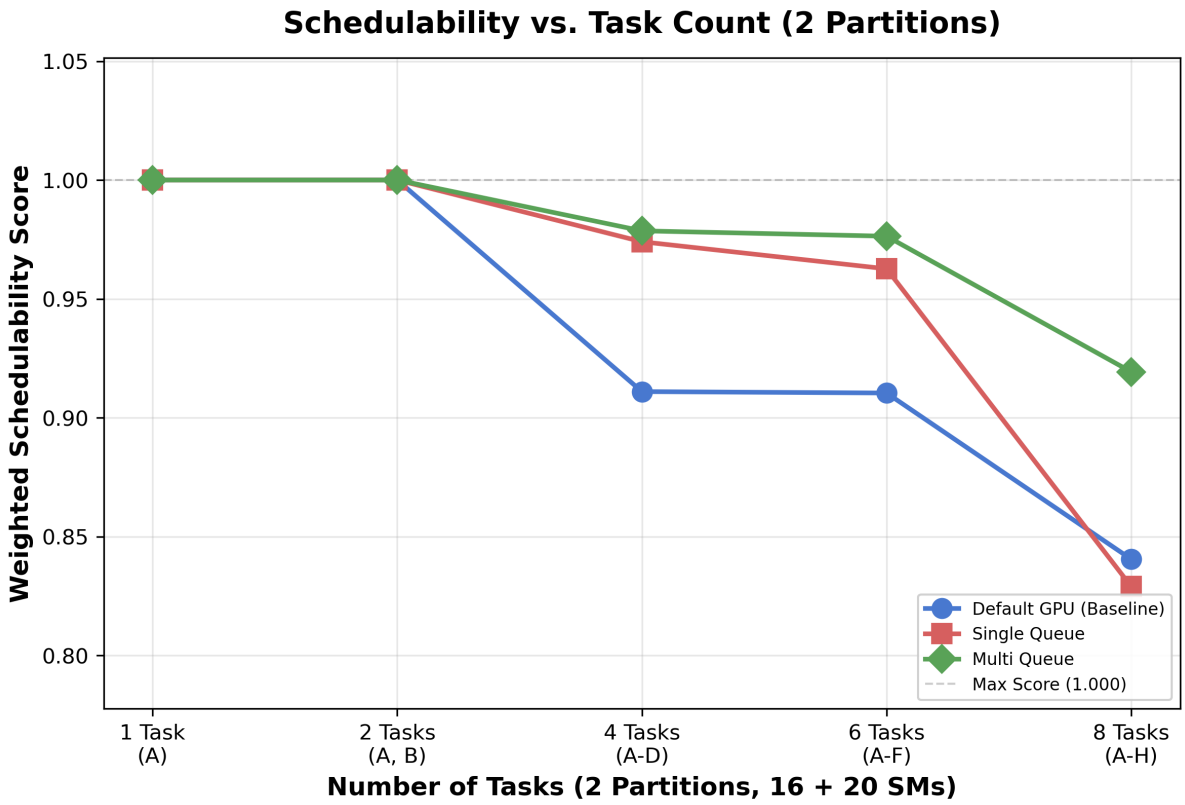


Figure 4.7 Weighted schedulability score for 2 partitions under varying task counts: default baseline, single queue, and multi queue.

4.7.3.2.3 6 Tasks

With six tasks (3 per partition), all three approaches experience a slight degradation. Multi queue remains the most resilient. The maximum interference for any high-priority task is still bounded by blocking of a single lower-priority task, but the increased number of tasks per partition raises the probability that a lower-priority task is executing when a higher-priority task arrives, resulting in a modest increase in deadline misses. The baseline shows similar scores at 4 and 6 tasks. However, this does not indicate a fundamental limit on degradation. Table 4.6, for baseline, demonstrates the expected increase in deadline misses for Task A, with the increasing number of tasks ($\approx$ 3% more). Since the baseline provides no priority enforcement, its performance is entirely dependent on the arrival patterns and execution overlap of tasks, which vary between runs. With different task configurations or timing, the baseline could degrade further at 6 tasks. Single queue continues to degrade as the global queue grows longer, increasing the likelihood of cross-partition head-of-line blocking. Task B is likely to be more frequently affected by head-of-line blocking as Task A's memory or kernel operations are blocked more often with increased system activity. Additional delays due to heap operations also increase as the number of tasks grow (with O(log(n)) complexity, where n is the number of tasks) resulting in more degradation compared to multi-queue.

4.7.3.2.4 8 Tasks

With 8 tasks (4 per partition), all the three approaches as expected experience a degradation. Miss rate for Task A (See Table 4.6) for baseline exceed both the approaches (significantly increased compared to 3 task per partition) reflecting massive queuing delays. However, schedulability score for single queue approach (Fig 4.7) drops below baseline because of a 11.5% increase in the miss rate for Task B. This demonstrates a significant negative impact of head-of-line blocking for all the tasks below the highest priority task. Due to a higher increase in queuing delay for the lower priority tasks we also see a non-zero miss rate for task D for both single and multi-queue approaches even with high deadlines.

4.8 Summary

This work presents OpenMP-RT-Offload, a novel multi-queue, priority-aware scheduling mechanism for OpenMP-RT tasks offloading workloads onto NVIDIA GPUs. OpenMP-RT-Offload leverages NVIDIA Green Contexts to configure partitions and `LD_PRELOAD` for the dynamic inter-positioning of CUDA calls to implement priority-aware queues. Results demonstrate reduced worst-case task response times for higher-priority tasks and a significant reduction in the miss rate, which ultimately improves the schedulability score. These results are achieved using only software-based techniques executed on actual hardware with synthetic benchmarks representing a real-time system.

By utilizing multiple queues for priority-aware scheduling and eliminating head-of-line blocking, OpenMP-RT-Offload reduces the worst-case response time of the two highest-priority tasks by approximately 12-25 ms compared to baseline (default execution). This strict reduction in worst-case behavior is critical for predictable execution time analysis in real-time systems. Miss rates evaluated over increasing partitions and task sets further demonstrate the efficacy of this design. For eight tasks distributed over two partitions, the multi-queue approach lowers the miss rate of the highest-priority task by $\approx 59\%$ (dropping from 46% under the baseline to 19%). Furthermore, by avoiding head-of-line blocking and the queue overheads associated with additional heap and sleep operations, the multi-queue approach outperforms the single-queue priority model, yielding a $\approx$ 50% reduction in the highest-priority task's miss rate and complete elimination of misses for the second-highest-priority task. When increasing the number of partitions in a four-task configuration such that each task executes in spatial isolation, both priority-aware scheduling approaches achieve a $\approx 99\%$ improvement in the miss rate. Overall, OpenMP-RT-Offload identified the challenges of GPU execution for real-time systems under default OpenMP offloads and demonstrated significant reductions in deadline miss rates without specialized hardware or driver support.

CHAPTER

5

CONCLUSION

Real-time systems built on modern high-performance platforms face a fundamental issue: They increasingly rely on distributed networked nodes, multi-core CPUs, and GPU accelerators to meet the architectural and computational demands of modern workloads, and each of which introduce unbounded or unobservable execution-time interference. Adversarial attacks can inject network delays that slip past conventional timeouts, contention between co-located threads can cause unbounded time dilation, and GPU accelerators serialize kernels in first-come-first-served order without awareness of task priority or real-time deadlines. In each case, the timing guarantees on which safety-critical workloads depend are threatened. This dissertation examined the hypothesis that software mechanisms providing fine-grained timing observability and priority-based scheduling are necessary to preserve timing guarantees and detect timing anomalies across these execution domains, specifically by contributing three such software mechanisms.

First, conventional network timeouts catch packets that fail to arrive but miss packets that arrive late — within the timeout window but past the real-time schedule's expected end-to-end time — leaving a class of timing attacks undetected until deadlines are already violated. We addressed this by developing T-Pack, which extends the Linux network stack to embed and monitor timing information within packets for both TCP and UDP communication. T-Pack was able to detect distributed denial-of-service delay attacks with high accuracy at packet reception time, well in advance of task deadlines and while composing cleanly with IPSec for cryptographic integrity, demonstrating that timing integrity can be preserved in distributed real-time systems against network-based adversarial interference.

Second, task-level deadlines provide only coarse-grained timing observability. By the time a task misses its deadline, the attack or anomaly has already caused damage, and the specific code region

responsible is not identifiable. We addressed this by developing T-Tex, which extends OpenMP with timed monitoring at code-region granularity through compiler-based instrumentation and OpenMP tracing, combined with Linux-kernel-level support for tracking execution time across context switches. By decomposing task-level timing into per-region timing bounds, T-Tex detects delay-based interference as soon as a region exceeds its local bound — well in advance of task's deadline expiration — demonstrating that timing integrity can be preserved in multi-threaded real-time applications at a granularity that task-level monitoring cannot achieve.

Finally, GPU accelerators execute kernels in first-come-first-served order without priority awareness, leading to unbounded delays for high-priority tasks under contention from lower-priority workloads sharing the device. We addressed this by developing OpenMP-RT- Offload, a transparent CUDA interposer that partitions streaming multiprocessors into isolated groups and dispatches OpenMP target regions through a priority-aware scheduler. OpenMP-RT-Offload was able to enforce predictable kernel execution within each partition while spatial isolation eliminated cross-task interference between partitions, demonstrating that timing predictability can be preserved in accelerator-based execution without modifying application code or the OpenMP runtime.

Together, these contributions demonstrate that the hypothesis holds. Software mechanisms operating at the network, thread, and accelerator layers of the real-time software stack can observe timing behavior at a granularity finer than task-level deadlines and schedule execution to preserve timing integrity against malicious interference and timing predictability under contention, without requiring specialized hardware or application-level modifications.

REFERENCES

[Ame21] Amert, T. et al. “Timewall: Enabling time partitioning for real-time multicore+ accelerator platforms”. *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2021, pp. 455–468.

[Ann17] Annessi, R. et al. “SecureTime: Secure multicast time synchronization”. *arXiv preprint arXiv:1705.10669* (2017).

[Bak09] Bak, S. et al. “The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety”. *IEEE Real-Time Embedded Technology and Applications Symposium*. 2009, pp. 99–107.

[Bec19] Bechtel, M. & Yun, H. “Denial-of-service attacks on shared cache in multicore: Analysis and prevention”. *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2019, pp. 357–367.

[Bec22] Bechtel, M. & Yun, H. “Denial-of-Service Attacks on Shared Resources in Intel’s Integrated CPU-GPU Platforms”. *2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)*. IEEE. 2022, pp. 1–9.

[Bie08] Bienia, C. et al. “The PARSEC benchmark suite: Characterization and architectural implications”. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008, pp. 72–81.

[Bri06] Brisset, P. et al. “The paparazzi solution”. *2nd US-European Competition and Workshop on Micro Air Vehicles*. 2006.

[Bru05] Brumley, D. & Boneh, D. “Remote timing attacks are practical”. *Computer Networks* **48**.5 (2005), pp. 701–716.

[Buf99] Buff, R. & Goldberg, A. “Web Servers Should Turn Off Nagle to Avoid Unnecessary 200 ms Delays” (1999).

[Bur13] Burgio, P. et al. “Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters”. *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1504–1509.

[Cal09] Callegati, F. et al. “Man-in-the-Middle Attack to the HTTPS Protocol”. *IEEE Security & Privacy* **7**.1 (2009), pp. 78–81.

[Cer20] Cerrolaza, J. P. et al. “Multi-core devices for safety-critical systems: A survey”. *ACM Computing Surveys (CSUR)* **53**.4 (2020), pp. 1–38.

[Cet24] Cetre, C. et al. “Event-based OpenMP tasks for time-sensitive GPU-accelerated systems”. *International Workshop on OpenMP*. Springer. 2024, pp. 31–45.

[Cha09] Chapman, B. et al. “Implementing OpenMP on a high performance embedded multicore MPSoC”. *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–8.

[Che11] Checkoway, S. et al. “Comprehensive experimental analyses of automotive attack surfaces.” *USENIX Security Symposium*. Vol. 4. San Francisco. 2011, pp. 447–462.

[Che18] Chen, C.-Y. et al. “Securing real-time internet-of-things”. *Sensors* **18**.12 (2018), p. 4356.

[Cla] *Clang: Compiler Front-end*.

[Rfc] *Congestion Control in IP/TCP Internetworks*. RFC 896. 1984.

[Con99] Conover, M. *w00w00 on heap overflows*. 1999.

[Cre07] Crenshaw, T. et al. “The Simplex Reference Model: Limiting Fault-Propagation Due to Unreliable Components in Cyber-Physical System Architectures”. *IEEE Real-Time Systems Symposium*. 2007, pp. 400–412.

[Cro03] Crosby, S. A. & Wallach, D. S. “Denial of Service via Algorithmic Complexity Attacks.” *USENIX Security Symposium*. 2003, pp. 29–44.

[Dhe98] Dhem, J.-F. et al. “A practical implementation of the timing attack”. *International Conference on Smart Card Research and Advanced Applications*. Springer. 1998, pp. 167–182.

[Eic13] Eichenberger, A. E. et al. “OMPT: An OpenMP tools application programming interface for performance analysis”. *International Workshop on OpenMP*. Springer. 2013, pp. 171–185.

[Fer13] Ferry, D. et al. “A real-time scheduling service for parallel tasks”. *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2013, pp. 261–272.

[Fra11a] Francillon, A. et al. “Relay attacks on passive keyless entry and start systems in modern cars”. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Eidgenössische Technische Hochschule Zürich, Department of Computer Science. 2011.

[Fra11b] Francis, L. et al. “Practical Relay Attack on Contactless Transactions by Using NFC Mobile Phones.” *IACR Cryptology ePrint Archive* **2011** (2011), p. 618.

[Gor04] Gorrieri, R. et al. “Automated analysis of timed security: a case study on web privacy”. *International Journal of Information Security* **2**.3-4 (2004), pp. 168–186.

[Had07] Hadjidoukas, P. E. & Dimakopoulos, V. V. “Nested parallelism in the ompi openmp/c compiler”. *Euro-Par 2007 Parallel Processing: 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007. Proceedings 13*. Springer. 2007, pp. 662–671.

[Hor02] Horovitz, O. “Big loop integer protection”. *Phrack Inc., Dec* (2002).

[Cud] *http://www.nvidia.com/object/cuda_home.html*.

[Hua10] Huang, H.-M. et al. “Cyber-physical systems for real-time hybrid structural testing: a case study”. *Proceedings of the 1st ACM/IEEE international conference on cyber-physical systems*. 2010, pp. 69–78.

[Ian10] Iancu, C. et al. “Oversubscription on multicore processors”. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–11.

[IEE] IEEE. *1588-2019 - IEEE Approved Draft Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. `https://standards.ieee.org/content/ieee-standards/en/standard/1588-2019.html`, month = feb, year = 2020,

[Jin11] Jin, H. et al. "High performance computing using MPI and OpenMP on multi-core parallel systems". *Parallel Computing* **37**.9 (2011), pp. 562–575.

[Kat15] Kato, S. et al. "An open approach to autonomous vehicles". *IEEE Micro* **35**.6 (2015), pp. 60–68.

[Koc96] Kocher, P. C. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". *Annual International Cryptology Conference*. Springer. 1996, pp. 104–113.

[Kop08] Kopetz, H. "The Rationale for Time-Triggered Ethernet". *2008 Real-Time Systems Symposium*. 2008, pp. 3–11.

[Kos10] Koscher, K. et al. "Experimental security analysis of a modern automobile". *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 447–462.

[Led18] Ledinot, E. *Workshop on Analysis Tools and Methodology for Embedded Systems, The 2018 Industrial Challenge*. `https://w3.onera.fr/waters2018/node/13`. Dassault Aviation, 2018.

[Lee08] Lee, E. A. "Cyber physical systems: Design challenges". *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*. IEEE. 2008, pp. 363–369.

[Lee16] Lee, E. A. & Seshia, S. A. *Introduction to embedded systems: A cyber-physical systems approach*. MIT press, 2016.

[Liu73] Liu, C. L. & Layland, J. W. "Scheduling algorithms for multiprogramming in a hard-real-time environment". *Journal of the ACM (JACM)* **20**.1 (1973), pp. 46–61.

[Lou19] Lou, X. et al. "Assessing and mitigating impact of time delay attack: a case study for power grid frequency control". *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. ACM. 2019, pp. 207–216.

[Mar13] Marongiu, A. et al. "Improving the programmability of STHORM-based heterogeneous systems with offload-enabled OpenMP". *Proceedings of the First International Workshop on Many-core Embedded Systems*. 2013, pp. 1–8.

[Mar15] Mary, C. "Shellshock attack on linux systems–bash". *International Research Journal of Engineering and Technology* **2**.8 (2015), pp. 1322–1325.

[McD22] McDonald, B. & Mueller, F. "T-SYS: Timed-Based System Security for Real-Time Kernels". *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2022, pp. 247–258.

[McD24] McDonald, B. & Mueller, F. “OpenMP-RT: Native Pragma Support for Real-Time Tasks and Synchronization with LLVM under Linux”. *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 2024, pp. 119–130.

[Mil91] Mills, D. L. “Internet time synchronization: the network time protocol”. *Communications, IEEE Transactions on* **39**.10 (1991), pp. 1482–1493.

[Mir04] Mirkovic, J. & Reiher, P. “A taxonomy of DDoS attack and DDoS defense mechanisms”. *ACM SIGCOMM Computer Communication Review* **34**.2 (2004), pp. 39–53.

[Mit20] Mittal, S. “T-Pack, Timed Network Security In Real Time System”. MA thesis. Raleigh, NC: North Carlina State University, 2020.

[Mit21] Mittal, S. & Mueller, F. “T-Pack: Timed Network Security for Real Time Systems”. *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE. 2021, pp. 20–28.

[Mit25a] Mittal, S. & Mueller, F. “T-Tex: Timed Threaded Execution for Real-time Security and Safety”. *Proceedings of the ACM/IEEE 16th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2025)*. 2025, pp. 1–11.

[Mit25b] Mittal, S. et al. “OpenMP-Q: Quantum Task Offloading in OpenMP”. *International Workshop on OpenMP*. Springer. 2025, pp. 81–95.

[Moh02] Mohr, B. et al. “Design and prototype of a performance tool interface for OpenMP”. *The Journal of Supercomputing* **23**.1 (2002), pp. 105–128.

[Nar18] Narula, L. & Humphreys, T. E. “Requirements for secure clock synchronization”. *IEEE Journal of Selected Topics in Signal Processing* **12**.4 (2018), pp. 749–762.

[Ni25] Ni, Y. et al. “HARD: Hardening Real-Time Scheduling and Analysis for Accelerator Enabled Computing”. *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2025, pp. 389–401.

[Nic96] Nichols, B. et al. *Pthreads programming: A POSIX standard for better multiprocessing*. " O’Reilly Media, Inc.", 1996.

[NVI] NVIDIA. *CUDA Driver API: Green Contexts*. `https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__GREEN__CONTEXTS.html`. Accessed: Nov. 9, 2025.

[One96] One, A. “Smashing the stack for fun and profit”. *Phrack magazine* **7**.49 (1996), pp. 14–16.

[Ped16] Pedroza, G. et al. “Timed-model-based Method for Security Analysis and Testing of Smart Grid Systems”. *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE. 2016, pp. 35–42.

[Pel10] Pellizzoni, R. et al. “Worst case delay analysis for memory interference in multicore systems”. *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE. 2010, pp. 741–746.

[Qia17] Qian, T. et al. “A Linux Real-Time Packet Scheduler for Reliable Static SDN Routing”. *Euromicro Conference on Real-Time Systems*. 2017, 25:1–25:22.

[Reg19] Reghenzani, F. et al. “The real-time linux kernel: A survey on preempt_rt”. *ACM Computing Surveys (CSUR)* **52**.1 (2019), pp. 1–36.

[Ros06] Rostedt, S. *RT-Mutex implementation design*. 2006.

[Sar15] Sargolzaei, A. et al. “A novel technique for detection of time delay switch attack on load frequency control”. *Intelligent Control and Automation* **6**.04 (2015), p. 205.

[Sch00] Schindler, W. “A timing attack against RSA with the chinese remainder theorem”. *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2000, pp. 109–124.

[Ser18] Serrano, M. A. et al. “Towards an OpenMP specification for critical real-time systems”. *International Workshop on OpenMP*. Springer. 2018, pp. 143–159.

[SHA18] SHAAR, F. & EFE, A. “DDoS attacks and impacts on various cloud computing components”. *Int. Journal of Information Security Science* **7** (2018), pp. 26–48.

[Som19] Sommer, L. et al. “DAPHNE-An automotive benchmark suite for parallel programming models on embedded heterogeneous platforms: work-in-progress”. *Proceedings of the International Conference on Embedded Software Companion*. 2019, pp. 1–2.

[Sor94] Sorel, Y. “Massively parallel computing systems with real time constraints: the" Algorithm Architecture Adequation" methodology”. *Proceedings of the First International Conference on Massively Parallel Computing Systems (MPCS) The Challenges of General-Purpose and Special-Purpose Computing*. IEEE. 1994, pp. 44–53.

[Sta04] Stankovic, J. A. & Rajkumar, R. “Real-time operating systems”. *Real-Time Systems* **28**.2-3 (2004), pp. 237–253.

[Sta88] Stankovic, J. A. et al. *Real-time computing systems: The next generation*. University of Massachusetts at Amherst. Computer and Information Science [COINS], 1988.

[Sto13] Stotzer, E. et al. “Openmp on the low-power ti keystone ii arm/dsp system-on-chip”. *International Workshop on OpenMP*. Springer. 2013, pp. 114–127.

[Sun17] Sun, J. et al. “Real-time scheduling and analysis of OpenMP task systems with tied tasks”. *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2017, pp. 92–103.

[Vai10] Vaidehi, M & Nair, T. “Multicore applications in real time systems”. *arXiv preprint arXiv:1001.3539* (2010).

[VL21] Valero-Lara, P. et al. “Openmp target task: Tasking and target offloading on heterogeneous systems”. *European Conference on Parallel Processing*. Springer. 2021, pp. 445–455.

[Var15] Vargas, R. et al. “OpenMP and timing predictability: A possible union?” *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 617–620.

[Wan13] Wang, C. et al. “libEOMP: a portable OpenMP runtime library based on MCA APIs for embedded systems”. *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*. 2013, pp. 83–92.

[Wan20] Wang, Y. et al. “Partitioning-Based Scheduling of OpenMP Task Systems With Tied Tasks”. *IEEE Transactions on Parallel and Distributed Systems* **32**.6 (2020), pp. 1322–1339.

[Wik] Wikipedia. *Time-Sensitive Networking*. `https://en.wikipedia.org/wiki/Time-Sensitive_Networking`.

[Wil07] Wilhelm, R. et al. *The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools*. Tech. rep. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-209/2007-1-SE. Maelardalen Real-Time Research Centre, Maelardalen University, 2007.

[Wil08] Wilhelm, R. et al. “The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools”. *ACM Transactions on Embedded Computing Systems* **7**.3 (2008), pp. 1–53.

[Yan18] Yang, M. “Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems”. *Proceedings of the 30th Euromicro Conference on Real-Time Systems*. 2018.

[Zha13] Zhang, F. & Burns, A. “Schedulability analysis of EDF-scheduled embedded real-time systems with resource sharing”. *ACM Transactions on Embedded Computing Systems (TECS)* **12**.3 (2013), pp. 1–19.

[Zha09] Zhang, Y. et al. “Real-time performance and middleware for multiprocessor and multicore linux platforms”. *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2009, pp. 437–446.

[Zha06] Zhang, Y. & West, R. “Process-aware interrupt scheduling and accounting”. *2006 27th IEEE International Real-Time Systems Symposium (RTSS’06)*. IEEE. 2006, pp. 191–201.

[Zhu18] Zhu, F. et al. “Baidu apollo auto-calibration system-an industry-level data-driven and learning based vehicle longitude dynamic calibrating algorithm”. *arXiv preprint arXiv:1808.10134* (2018).

[Zim10] Zimmer, C. et al. “Time-Based Intrusion Dectection in Cyber-Physical Systems”. *International Conference on Cyber-Physical Systems*. 2010, pp. 109–118.

[Zim12] Zimmer, C. & Mueller, F. “Fault resilient real-time design for noc architectures”. *2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*. IEEE. 2012, pp. 75–84.

[Zou23] Zou, A. et al. “RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization”. *IEEE Transactions on Parallel and Distributed Systems* **34**.5 (2023), pp. 1450–1465.