T-Tex: Timed Threaded Execution for Real-time Security and Safety

Swastik Mittal
smittal6@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Frank Mueller
fmueller@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Abstract

Task scheduling and resource management are increasingly subject to attacks exposing system vulnerabilities, particularly on multi-core processors with an attack surface crossing cores and tasks with different privileges. Meanwhile, modern real-time systems utilize multi-core environments, where delay attacks can force deadline misses.

This work proposes "Timed Threaded Execution" (T-Tex), a method to detect such security attacks based on monitoring time dilation induced by unexplained delays in general, and more specifically for OpenMP. T-Tex extends OpenMP by exposing it to timed monitoring of code execution. It contributes novel compilation techniques for timed instrumentation exemplified for LLVM via multi-phase profiling using OpenMP tracing (OMPT) capabilities. T-Tex also contributes Linux kernel modifications to monitor thread-level execution time across context switches between threads.

Experiments on a real platform demonstrate that T-Tex can detect 100% of delay-based intrusions constrained by timer granularity to an unprecedented 60us vulnerability threshold at a performance overhead of 11% – 72% for Parsec and Daphne benchmarks.

ACM Reference Format:
Swastik Mittal and Frank Mueller. 2025. T-Tex: Timed Threaded Execution for Real-time Security and Safety. In *ACM/IEEE 16th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2025) (ICCPS '25), May 6–9, 2025, Irvine, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3716550.3722016

1 Introduction

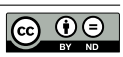
The use of multi-cores has significantly increased in real-time systems to meet rapidly increasing requirements for high performance and low power consumption. In the past few years, there has been a stagnation in processor frequencies. However, this has been replaced by significant increases in the number of processing cores per chip.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 [8, 26].

OpenMP is a powerful and widely-used framework for parallel programming, integral to general-purpose applications, machine learning, and high-performance computing [27]. 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 [5, 6, 15, 25, 28].

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 [29]. 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 [17]. 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 paper 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 [11]. 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). In today's technology landscape, system-level security is indispensable, protecting systems across multiple layers from attacks that target memory (e.g., buffer overflows [22]), exploit vulnerabilities in value handling [10], 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 [14].

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) [33], secure clock synchronization [20], and protected critical infrastructure of smart grids [23].



ICCPS '25, May 6–9, 2025, Irvine, CA, USA

ACM ISBN 979-8-4007-1498-6/2025/05
https://doi.org/10.1145/3716550.3722016

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 [1]. (2) It employs a portable multi-phase profiling approach based on OMPT [7]. (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.

2 Related Work

Previous studies have highlighted the impact of delay attacks on cyber-physical systems (CPS) that are subject to real-time constraints [14]. 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 [3]. 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 [2] demonstrates that modifications in how the resources are scheduled and utilized can prevent or mitigate the impact of these attacks.

Zimmer et al. [33] 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 [18] 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 [16] 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 multi-processor using novel techniques to protect code regions within a given maximum vulnerability threshold (upper bound of the vulnerability window), carefully considering the above mentioned intricacies and also complementing T-SYS to secure parallel regions along with sequential regions of the code.

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 [12]. 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. [30] 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 [9].

Considering these characteristics of OpenMP, this work assumes a real-time OpenMP model similar to OpenMP-RT [17], 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 [21]. 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 [17]), 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. 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.

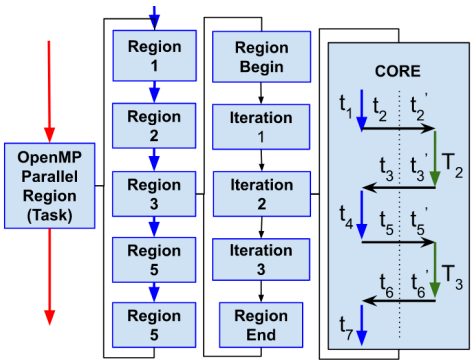


Figure 1: Analysis Model: Timed analysis of Parallel Region - broken into finer code regions - broken into loop iterations - analyzing context-switch time (t_1-t_7 represent the execution times of the regions they are marked alongside)

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. 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 [16] 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, 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. 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. 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. 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.

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.

4.1 Identifying Code Regions

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 [17] 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 [19] 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 [7]. These OMPT callbacks are utilized by T-Tex to identify the code regions depicted in Fig. 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. 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 [1] 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).

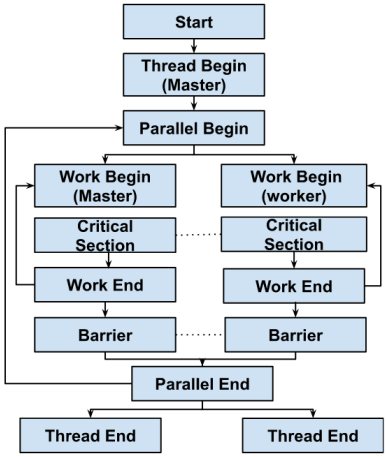


Figure 2: Code Regions for Implicit Parallelism in OpenMP

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. 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.

4.1.3 ***Monitoring Executing Threads.*** As discussed in Sec. 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.

4.2 Timers

T-Tex implements a multi-timer model (Appendix A:A.1) to avoid synchronization overheads. Each thread's timer activates upon entering a protected code region (Sect. 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. 5). Each phase tracks execution times and maintains timers based on the WCET per region, resetting and recalibrating at a region's completion.

4.3 Accurate Analysis via Maintaining Timers at Context-Switch

In a multi-processor framework (see Sec. 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). 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.

5 Implementation

Figure 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. 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.

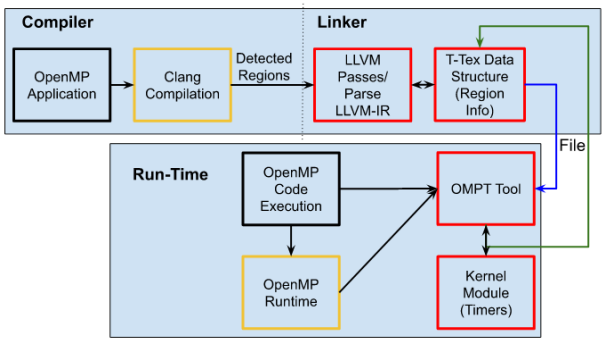


Figure 3: T-Tex Workflow

5.1 Compiler Modifications: Clang Frontend

Clang [1] 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. 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.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). Consider Listing 1 (see Appendix). Clang generates two arrays, [2,-1] & [-1], to denote two parallel 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). In Listing 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 & Fig. 2).

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. 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

Listing 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; }
}
```

(Sec. 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.

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).

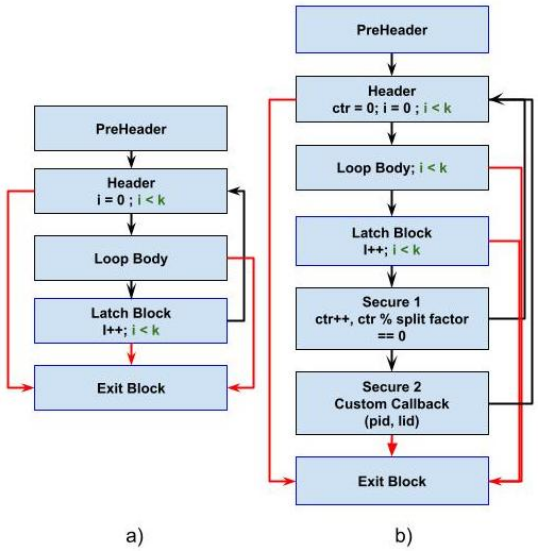


Figure 4: Loop Control Flow Graph: Red Arrow = possible edges, Black Arrow = 100% existing edge, Blue Box = one or more basic blocks, Green condition = may or may not be in the given block a) Loop before T-Tex; b) Loop after T-Tex

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. 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.

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 1 in Appendix A).

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.

5.4 Runtime Modifications: OpenMP & OMPT

The OpenMP runtime library functions (see Fig. 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. 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). 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.

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). 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 [31] 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 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 2.

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.

Listing 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)
```

6 Experiments and Evaluation

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 [13] and Apollo AI [32] illustrate OpenMP's role in autonomous driving and embedded systems.

We test T-Tex on the Daphne benchmarks [24], 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 [4]. 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. 1 in Appendix A).

6.2 Evaluation

Figure 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).

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). The experimental evaluation shows that ioctl calls (see Sect. 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

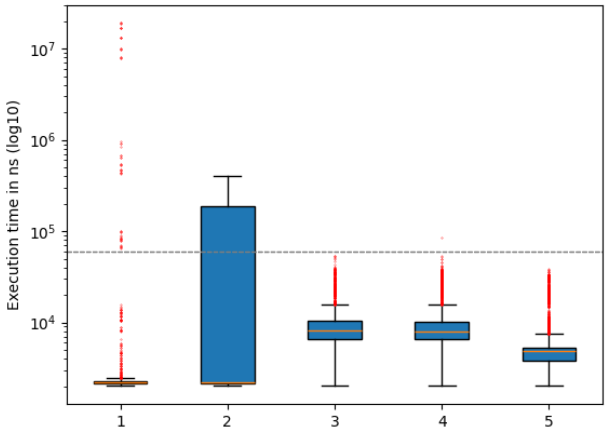


Figure 5: Phase 1-5 of T-Tex (further division of code regions to reduce WCET of each region) : Execution Time (ns) (black line: 60us security threshold)

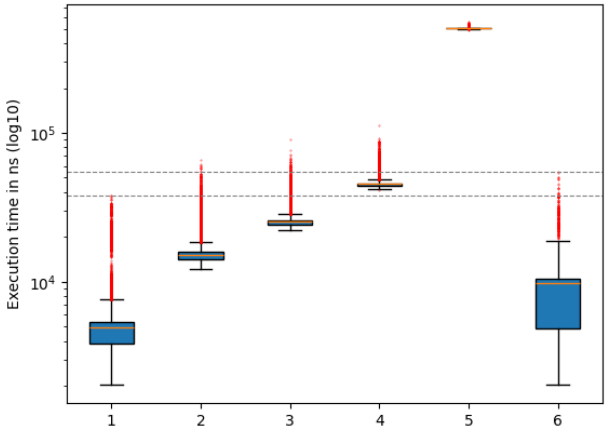


Figure 6: Attack Detection and Vulnerability 1) T-Tex (No Attack) 2) 10us Delay 3) 20us Delay 4) 40us Delay 5) 500us Delay 6) Basic Block Protection (T-SYS: No Attack)

regions reduce into 2 or more halves, this narrows the box plot whiskers by reducing the outliers within the box (see Fig 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 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. 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). 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). 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. 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. 7 in Appendix A:**??**). From the prior Fig. 6, we observe overlapping outliers in test cases 1 with 2. Fig. 7 elaborates this using frequency distribution graphs (T-Tex in red and 10us delay in purple). 0.7% of the executed regions by T-Tex overlap with the attack delay (see zoomed in image in Fig. 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. 6 shows overlapping execution times for code regions protected by basic block protection compared to a 40us delay attack (upper black/dotted line). Fig. 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 5 and 6 that basic block protection fails to constrain all outliers within the security threshold.

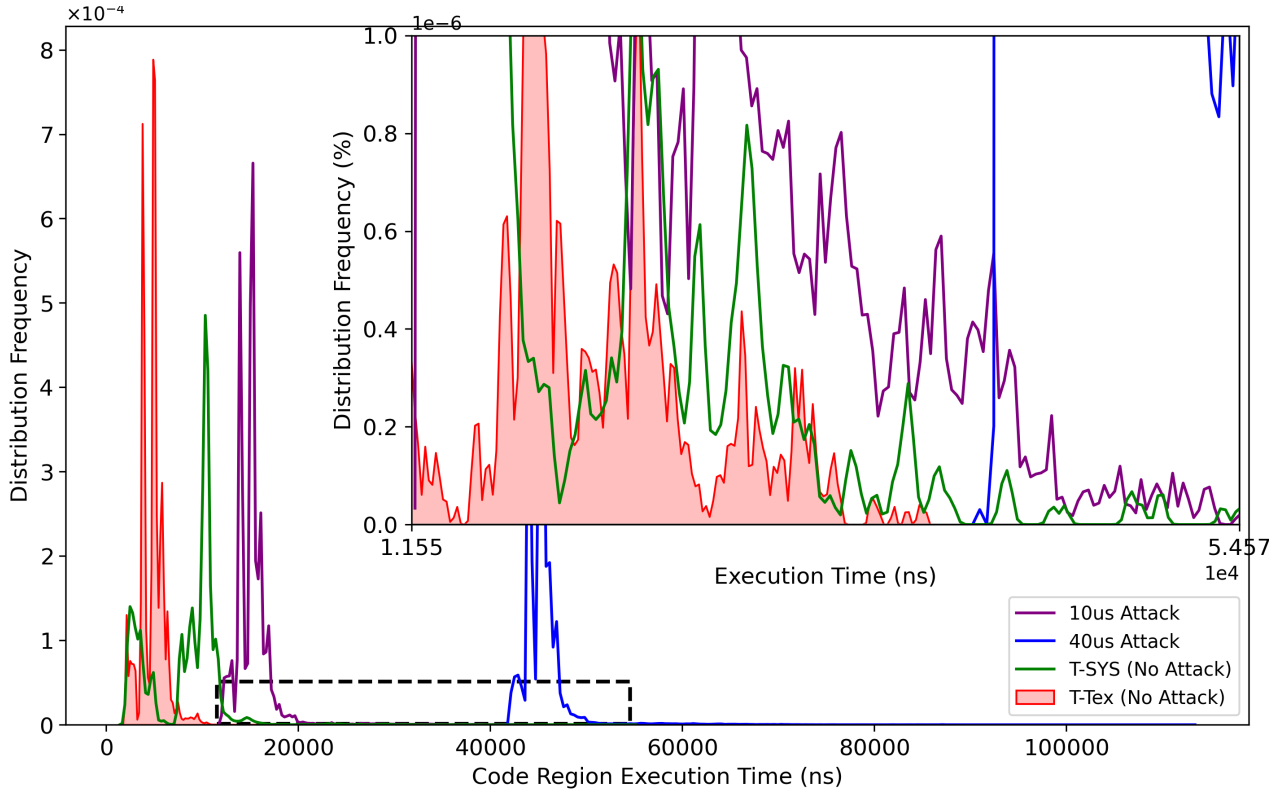


Figure 7: Time Distribution from Fig 6 — Image inside is a zoomed in image of the black rectangular box (Frequency Density of 0.001%): 1) 0.7% of red overlaps with purple 2) 0.05% of green overlaps with blue.

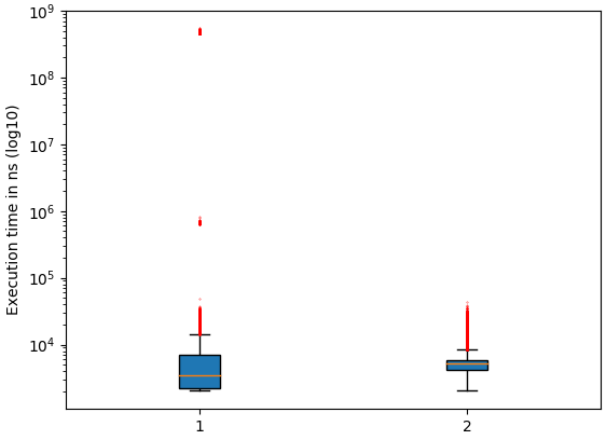


Figure 8: Benefit of Kernel Timer Crediting 1) T-Tex without kernel time crediting 2) T-Tex with kernel time crediting

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

7 Conclusion

This work presents the design and implementation of T-Tex, a novel timed analysis and attack detection technique for real-time multi-processor 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%

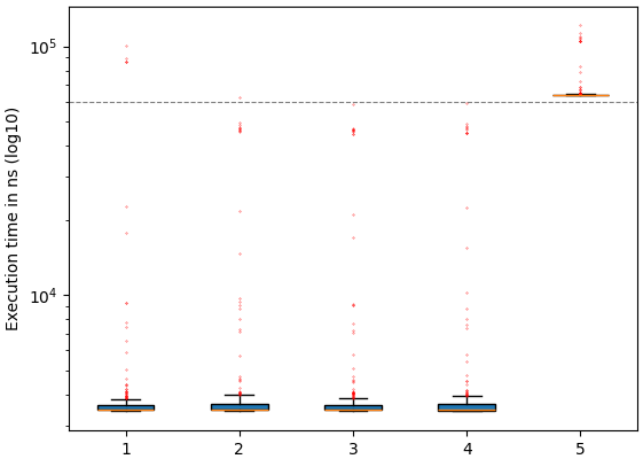


Figure 9: T-Tex security on parsec — frequency mining application: 1-4) Number of T-Tex iterations to reach security threshold of 60us 5) Attack: 60us Delay

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.

Acknowledgment

This work was supported in part by NSF grants 2316201, 2217020, 1813004 and 1747555.

References

[1] [n. d.]. Clang: Compiler Front-end. https://clang.llvm.org/
[2] Michael Bechtel and Heechul Yun. 2019. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 357–367.
[3] Michael Bechtel and Heechul Yun. 2022. Denial-of-Service Attacks on Shared Resources in Intel's Integrated CPU-GPU Platforms. In *2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)*. IEEE, 1–9.
[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.
[5] Paolo Burgio, Giuseppe Tagliavini, Andrea Marongiu, and Luca Benini. 2013. Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1504–1509.
[6] Barbara Chapman, Lei Huang, Eric Biscondi, Eric Stotzer, Ashish Shrivastava, and Alan Gatherer. 2009. Implementing OpenMP on a high performance embedded multicore MPSoC. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–8.
[7] Alexandre E Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. 2013. OMPT: An OpenMP tools application programming interface for performance analysis. In *International Workshop on OpenMP*. Springer, 171–185.
[8] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. 2013. A real-time scheduling service for parallel tasks. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 261–272.
[9] Panagiotis E Hadjidoukas and Vassilios V Dimakopoulos. 2007. Nested parallelism in the ompi openmp/c compiler. In *Euro-Par 2007 Parallel Processing: 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007. Proceedings 13*. Springer, 662–671.
[10] Oded Horovitz. 2002. Big loop integer protection. *Phrack Inc., Dec* (2002).
[11] Huang-Ming Huang, Terry Tidwell, Christopher Gill, Chenyang Lu, Xiuyu Gao, and Shirley Dyke. 2010. Cyber-physical systems for real-time hybrid structural testing: a case study. In *Proceedings of the 1st ACM/IEEE international conference on cyber-physical systems*. 69–78.
[12] Costin Iancu, Steven Hofmeyr, Filip Blagojević, and Yili Zheng. 2010. Oversubscription on multicore processors. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–11.
[13] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. 2015. An open approach to autonomous vehicles. *IEEE Micro* 35, 6 (2015), 60–68.
[14] Xin Lou, Cuong Tran, Rui Tan, David KY Yau, and Zbigniew T Kalbarczyk. 2019. Assessing and mitigating impact of time delay attack: a case study for power grid frequency control. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. ACM, 207–216.
[15] Andrea Marongiu, Alessandro Capotondi, Giuseppe Tagliavini, and Luca Benini. 2013. Improving the programmability of STHORM-based heterogeneous systems with offload-enabled OpenMP. In *Proceedings of the First International Workshop on Many-core Embedded Systems*. 1–8.
[16] Brayden McDonald and Frank Mueller. 2022. T-SYS: Timed-Based System Security for Real-Time Kernels. In *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 247–258.
[17] Brayden McDonald and Frank Mueller. 2024. OpenMP-RT: Native Pragma Support for Real-Time Tasks and Synchronization with LLVM under Linux. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 119–130.
[18] Swastik Mittal and Frank Mueller. 2021. T-Pack: Timed Network Security for Real Time Systems. In *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 20–28.
[19] Bernd Mohr, Allen D Malony, Sameer Shende, and Felix Wolf. 2002. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing* 23, 1 (2002), 105–128.
[20] Lakshay Narula and Todd E Humphreys. 2018. Requirements for secure clock synchronization. *IEEE Journal of Selected Topics in Signal Processing* 12, 4 (2018), 749–762.
[21] Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. 1996. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.".
[22] Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.
[23] Gabriel Pedroza, Pascale Le Gall, Christophe Gaston, and Fabrice Bersey. 2016. Timed-model-based Method for Security Analysis and Testing of Smart Grid Systems. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 35–42.
[24] Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. 2019. DAPHNE-An automotive benchmark suite for parallel programming models on embedded heterogeneous platforms: work-in-progress. In *Proceedings of the International Conference on Embedded Software Companion*. 1–2.
[25] Eric Stotzer, Ajay Jayaraj, Murtaza Ali, Arnon Friedmann, Gaurav Mitra, Alistair P Rendell, and Ian Lintault. 2013. Openmp on the low-power ti keystone ii arm/dsp system-on-chip. In *International Workshop on OpenMP*. Springer, 114–127.
[26] M Vaidehi and TR Nair. 2010. Multicore applications in real time systems. *arXiv preprint arXiv:1001.3539* (2010).
[27] Roberto Vargas, Eduardo Quinones, and Andrea Marongiu. 2015. OpenMP and timing predictability: A possible union?. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 617–620.
[28] Cheng Wang, Sunita Chandrasekaran, Barbara Chapman, and Jim Holt. 2013. libEOMP: a portable OpenMP runtime library based on MCA APIs for embedded systems. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*. 83–92.
[29] Yang Wang, Xu Jiang, Nan Guan, Zhishan Guo, Xue Liu, and Wang Yi. 2020. Partitioning-Based Scheduling of OpenMP Task Systems With Tied Tasks. *IEEE Transactions on Parallel and Distributed Systems* 32, 6 (2020), 1322–1339.
[30] Yuanfang Zhang, Christopher Gill, and Chenyang Lu. 2009. Real-time performance and middleware for multiprocessor and multicore linux platforms. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 437–446.
[31] Yuting Zhang and Richard West. 2006. Process-aware interrupt scheduling and accounting. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 191–201.
[32] Fan Zhu, Lin Ma, Xin Xu, Dingfeng Guo, Xiao Cui, and Qi Kong. 2018. Baidu apollo auto-calibration system-an industry-level data-driven and learning based vehicle longitude dynamic calibrating algorithm. *arXiv preprint arXiv:1808.10134* (2018).
[33] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan. 2010. Time-Based Intrusion Dectection in Cyber-Physical Systems. In *International Conference on Cyber-Physical Systems*. 109–118.

A Appendix

A.1 Multiple Timers (T-Tex) vs. Single Timers

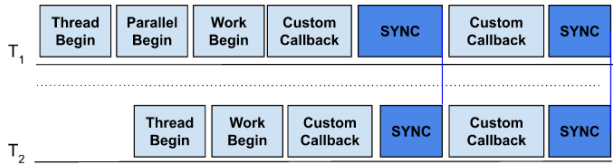


Figure 10: Single Timer: Custom Callbacks are instrumented timer calls for further dividing OpenMP recognized regions for finer analysis. More synchronizations needed for a smaller vulnerability window (resulting in more custom callbacks) defeating the purpose of multi-threaded model

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. 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.

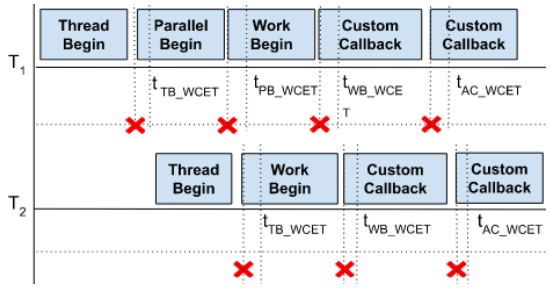


Figure 11: Multiple Timers: Timers armed at T_x are canceled (X) before being triggered, and a parallel region is partitioned into multiple sub-regions protected by finer-grained callbacks, which are not synchronized between threads until a final barrier.

Table 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