Intrusion Detection for CPS Real-Time Controllers [*]

Christopher Zimmer, Balasubramany Bhat, Frank Mueller, Sibin Mohan

Abstract
Security in CPS-based real-time embedded systems controlling the power grid has been an afterthought, but it is becoming a critical issue as CPS systems are networked and inter-dependent. This work presents a set of mechanisms for time-based intrusion detection, i.e., the execution of unauthorized instructions in real-time CPS environments. The novelty is the utilization of information obtained by static timing analysis for intrusion detection. Real-time CPS systems are unique in that timing bounds on code sections are readily available since they are required for schedulability analysis. We demonstrate how micro-timings can be exploited for multiple granularity levels of application code to track execution progress. Through bounds checking of these micro-timings, we develop techniques to detect intrusions (1) in a self-checking manner by the application and (2) through the operating system scheduler, which are novel contributions to the real-time/embedded systems domain to the best of our knowledge.

1 Introduction

The presence of embedded systems is altering today's life in many facets, and often in a subtle way that may go unnoticed — unless system failure impacts our lives. Examples range from non-critical systems (televisions, toasters), moderately critical systems (HVAC control systems, PHEV charging stations, traffic lights) to highly

Christopher Zimmer, Balasubramany Bhat, Frank Mueller
North Carolina State University, Raleigh, NC 27695-8206, mueller@cs.ncsu.edu

Sibin Mohan
University of Illinois at Urbana-Champaign, Urbana IL 61801, sibin@illinois.edu

[*] This work was supported in part by NSF grants 1329780, 1239246, 0812121 and U.S. Army Research Office (ARO) grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI). This is an extended version of a prior conference paper [50].

critical ones (power grid control, anti-lock brakes, and flight control systems). The latter two categories are examples of cyber-physical systems (CPS) where system control affects human lives or interacts with the environment. Most of these systems have real-time constraints, and ensuring that such systems are secure from intrusion and tampering is a design challenge of utmost importance. Securing CPSs dramatically deviates from security in general-purpose computing systems. In the latter, attacks may result in slower response or no execution at all. Imminent system failures, if detected, can be mitigated by rebooting or re-installation with a temporary lapse of services to users.

In safety critical real-time systems, in contrast, slower response or failure could result in significant environmental damage or even in loss of life. System restarts often cannot be instant due to an unstable physical system state, e.g., when an aircraft is in flight or a car is subject to slick roads requiring break control.

In practice, real-time software may have stringent requirements for CPS control. However, this still leaves vulnerabilities exposed by libraries and specific embedded domain device software. Attackers may exploit these by eventually executing arbitrary code that they have injected. Such code injection attacks have been common for several years in the general-purpose domain. As more embedded applications utilize networks they become more susceptible to such attacks, a problem particularly for CPS applications due to their increasing network connectivity.

One critical observation for this work lies in how embedded real-time systems are designed today. Their unique requirements lend themselves well to security methodologies that simply do not apply to general-purpose computing. The key idea of this work is to rely on static analysis of application code that yields detailed timing bounds, which can subsequently be exploited to raise the protection of CPS systems in terms of cyber security.

During system design, timing analysis of embedded real-time tasks provides so-called worst case execution time (WCET) bounds. These bounds lend themselves naturally to security analysis. As WCET safely bounds the upper execution times for specific code sections, execution times above these bounds provide indications of a system compromise due to intrusion. We have designed a technique for embedded real-time systems where general-purpose domain protection may prove ineffective: Techniques such as address-space layout randomization [37] and StackGuard [13], designed for a 64-bit address space, can be defeated more easily in embedded 8/16/32-bit processors with brute-force attacks. Instruction Set Randomization [19] and other hardware enhancements [38, 20] impose high-overhead due to binary rewriting or require additional hardware (with limitations given their static buffer constraints), the cost and overhead of which simply cannot be accommodated in lower-end CPS platforms.

Contributions: This work contributes three mechanisms utilizing both instrumentation of and analysis from within real-time applications to detect timing perturbations resulting from the execution of unauthorized code. The approaches are demonstrated to be effective both under simulation and on a hardware platform. Using timing metrics and comparing them with worst-case bounds allows the detection of security breaches due to system intrusion. In addition, prior to an actual deadline miss, one can detect that an application is about to exceed its timing requirements, which allows one to still trigger appropriate actions in a timely manner before the deadline. The three mechanisms are:

1. We first introduce T-Rex, which utilizes timing bounds to detect intrusion at a fine-grained level through instrumentation of return paths. This method allows the detection of code injections due to smallest timing dilations, i.e., depending on system parameters as small as 5-22 cycles.
2. The second method, T-ProT, validates intra-task checkpoints via synchronous scheduler invocations to uncover coarser-grain injections between 9 and 5k cycles.
3. The third approach, T-AxT, exploits asynchronous scheduler-triggered timing validations of application code sections. It does so without requiring the application code to be instrumented.

These security checks can be strategically scheduled to utilize otherwise idle time in the schedule. By offering different levels of granularity through these schemes, sufficient time is given to transition to a fail-safe state after intrusion detection. If properly designed, evasive actions can still be accommodated within real-time deadlines.

2 Attack Model and Scenario

In this section, we discuss the attack and adversary models that are the premise for our contributions. We then demonstrate a sample attack under these constraints.

There are a number of scenarios for attacks on embedded systems with or without real-time constraints. Past security work predominantly focused on wireless networks in the domain of embedded systems, such as [45]. Models range from passive packet sniffing to various active attacks, such as network traffic disruption (*e.g.*, jamming, spoofing) and packet data tampering/rewriting. Our approach complements network-centric protection with application-level intrusion detection.

We assume that one or more network nodes have been compromised or an attacker has successfully authenticated a node under our adversary model. Node authentication may provide adversaries with control to the local (wired, wireless or ad-hoc) network. Such nodes can be embedded or general-purpose systems, they may be mobile or stationary. We assume that hardware parameters are not modified during an attack, *i.e.*, memory latencies and processor frequencies are not modified by the initial attack code. In contrast to network-level security, we take an application-centric approach for protection. While past work has focused on the application-layer network interface for providing protection [48, 49, 47], we focus on application-intrinsic protection, which does not compete but rather complements the above schemes. This is based on the premise that attacks originate from applications before the operating system is compromised. Our work focuses on early intrusion detection at the application level before other system or hardware parameters can be manipulated, *i.e.*, on the detection of intrusion on uncompromised nodes *via* code injection. Data injection attacks are beyond the scope of this work. We assume that the user data space is unsafe (partially or fully compromised) at the time of detection but the operating system space is still trusted as it has not been penetrated (yet).

In this work, we seek to protect embedded control software by enhancing it with sanity checks to uncover execution of unauthorized code in addition to regular application code. Consider the example in Figure 1 that obtains input data (via fscanf) from an array of input sensors (*e.g.*, temperatures) that are aggregated and later analyzed to drive feedback-control of an actuator valve. In our attack scenario, a network packet supplies the sensor data from a spoofed or compromised node, which we implemented on a MIPS ISA platform.

```
void Sum() {
        char localcpy[MAXSIZE];
        fscanf(input,"%s\n",&localcpy);
        for (i = 0; i < MAXSIZE; i++) {
                // Search for data, increment counter, ...
        }
        // Checkpoint 1 instr. in assembly
}
void read_data() {
        input = fopen("SomeNetworkDevice","r+");
        Sum();
        // Checkpoint 2 instr. in assembly
}
```

Fig. 1 Sample Code Vulnerability

A buffer overflow is caused by supplying an initial input string that exceeds the bound of the localcpy array. It overwrites both frame pointer and return address. When returning from the function after the loop, control is subsequently transferred to the first instruction in the Sum function (see Figure 2). Upon the second execution of Sum, a second input corrects both frame pointer and return address to resume execution as normal. Without ever causing a program fault, this attack results in `2` $\times$ `MAXSIZE` aggregations of legitimate sensor data within thresholds, yet the result would be averaged incorrectly over just `MAXSIZE` elements (code omitted). This may lead to an incorrect overall value that would usually go undetected.

In embedded systems, general-purpose and network-level protection methods are insufficient for such attacks for a number of reasons.

1. While this attack exploited a common library routine to trigger a buffer overflow, constraining analysis to a subset of vulnerable routines is insufficient in embedded systems where custom hardware devices expose non-standard input routines beyond POSIX library routines that may have exploits.

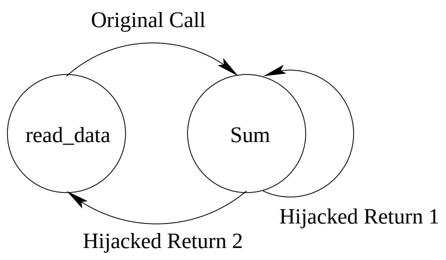

Fig. 2 Diverted Control Flow

2. Statistical detection methods [22] can be defeated in such a scenario by adaptively changing sensory input over time, which requires multiple repetitions of attacks if they can be detected at all.
3. Signature-based methods can be defeated through spoofing as embedded systems have limited computational capabilities that allow only symmetric signatures/encryption to be employed. Stronger public/private key pair signatures or encryption typically cannot be accommodated in given utilization bounds of lower-end embedded real-time systems [41].

Since our focus is on real-time systems, we follow an approach that differs significantly. In real-time systems, statically analyzable timing bounds are calculated at multiple granularity levels. We exploit time-bound checking as means to detect intrusions. For the attack in Figure 1, the time from the initially diverted return to the second return from Sum accounts for 14K additional cycles on the MIPS ISA. We have developed a number of application-centric techniques that can detect timing dilations as small as 5-22 cycles. With only minimal runtime overhead in the order of 1% of the application's execution time, the above intrusion was instantly detected. Our method detects not only this injection attack but also a variety of others. The approach is orthogonal to methods that protect against other attacks, such as data injection, timing, and denial of service attacks. Each of these attacks may require separate approaches for prevention or detection, *i. e.*, it is not realistic to expect a *single* method to secure against all of types of adversary approaches. Overall, time-based security can *complement* other security mechanisms. While it does not categorically prevent all attacks, it will raise the bar for code injection attacks.

3 Establishing Execution Time Bounds

In hard real-time systems, a priori determination of execution time bounds is a strict requirement. After all, a missed deadline may render the entire application incorrect. Timing analysis determines an application's best-case and worst-case execution time bound (BCET and WCET). This allows verification if a task's deadline can always be met. Timing analysis can be performed via dynamic [8, 42], static techniques [44, 31] or hybrids of them [5, 30, 43].

Dynamic timing analysis determines the effect of different inputs on execution time to approximate the WCET, *e.g.*, to determine that an inversely sorted list maximizes bubblesort's computational complexity. Static analysis bounds aggregate costs of instructions in blocks and then compound the costs of paths throughout the program taking architectural timing effects into account to derive a safe WCET bound at compile time. Static timing analysis has been shown to provide *safe* WCET bounds [42], much in contrast to the dynamic approach.

There are two reasons for deficiencies of dynamic analysis. (1) Due to explosion of the input space for just moderately complex software, it quickly becomes infeasible to determine worst-case inputs or exhaust all inputs during testing. (2) Even if worst-case inputs were known, hardware complexity no longer guarantees that worst-case timing occurs for the algorithmic worst-case input but may rather occur on other inputs, e.g., cache misses or branch mispredictions.

In this work, we utilize WCET bounds obtained from static timing analysis. While the objective of traditional timing analysis is to determine WCET bounds along the *longest* execution path, our work capitalizes on the ability to exploit timing results along *arbitrary* paths. Our work relies on WCET bounds for such paths but for *security* reasons and not for schedulability. We utilize the tool chain [18, 34, 32] depicted in Figure 3 to conduct our study. This enables us to accurately gauge the WCET bounds of an application (macro view) as well as small groups of instructions (micro view). A compiler translates the application to annotated Portable Instruction Set Architecture (PISA) assembly, which is a MIPS-like ISA [10]. This intermediate code along with loop bounds information is then fed into a control-flow analysis tool. Subsequently, control-flow analysis and static cache analysis are performed. The respective outputs are then consumed by a timing analyzer. It derives safe WCET bounds based on the annotated assembly and loop bounds.

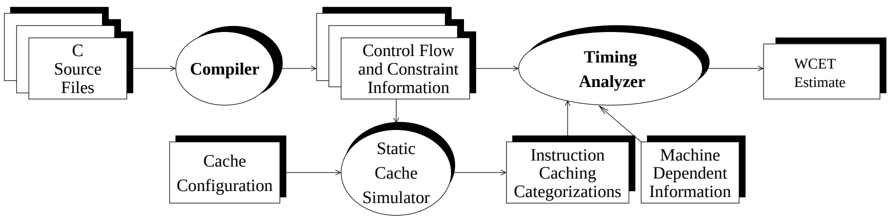
C Source Files
Compiler
Control Flow and Constraint Information
Timing Analyzer
WCET Estimate
Cache Configuration
Static Cache Simulator
Instruction Caching Categorizations
Machine Dependent Information

Fig. 3 Timing Analysis Tools

To support real-time security, we modified the timing analysis toolset in Figure 3. The original toolset provided timing feedback at the functional and loop level. We enhanced the toolset to supply timing bounds for a series of smaller ranges during the same analysis run including aggregate values of WCET bounds for sequential instructions plus the cost of branch mispredictions. The resulting bounds are tight and enable us to determine, within a reasonable margin, if a security breach has occurred, *e.g.*, through code injection.

4 Time-Based Intrusion Detection

There may be a variety of motivations for attackers to intrude systems, ranging from changing data for personal benefit to causing potentially catastrophic damage to the CPS environment, *e.g.*, to overload a power transformer by changing safety bounds data resulting in irreversible physical damage requiring components to be replaced (e.g., costly substation transformers). The common idea of our approach is not to prevent but rather detect intrusions, namely by verifying timing bounds at checkpoints during application execution. Our approach is generalized by a common methodology and systematic placement of checks within multiple system components as described in the following. We distinguish two checkpoint placement strategies, one that instruments the application and one where the real-time scheduler triggers checks called T-AxT. For application-side checkpoints, we promote what we term *macro* and *micro* checks of timing bounds. T-ProT competes with scheduler-triggered T-AxT checking at the macro level while T-Rex complements the other two schemes at the micro level.

Checkpoints are realized as synchronous system calls for application instrumentation or reside in the scheduler at preemptions. It is necessary to use system calls because user space provides insufficient data protection. Thus, we are using the real-time operating system as our trusted computing base. Critical security data, such as timing bounds, reside in a different address space than application code to decrease their vulnerability due to tampering.

Overall, the primary goal of this work is to design and assess methodologies that provide real-time CPS applications with an intrusion detection security mechanism. We next present several novel methods that work independently of one another or in a concerted fashion to provide elevated levels of protection within CPS applications.

4.1 Timed Return Execution (T-Rex)

Our first method, T-Rex, employs application-level checkpoints to detect code injections resulting in buffer overflow attacks. Typically, such attacks overwrite the return address of a routine whose frames are stored on the stack. Upon return from a function, control is transferred to the location indicated by the overwritten return address. Attackers often divert execution to hand-written instructions intentionally placed in global/stack variables, or they may spawn new programs. T-Rex detects the former while T-AxT (see below) addresses the latter.

Our T-Rex method employs a pair of checkpoints to compare WCET timing bounds with actually elapsed wall-clock time along a return (from subroutine) path. Figure 4 depicts this scenario.

The first checkpoint sets a timer equal to the WCET, and the second checkpoint cancels this timer. Failure to cancel this timer (due to time overrun) would result in an interrupt indicating a compromised system. Notice that T-Rex is equally ap-

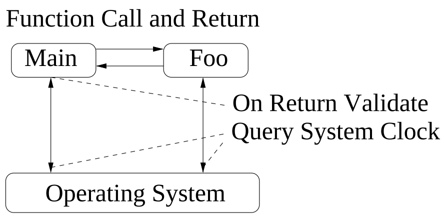

Fig. 4 Timed Return Execution (T-Rex)

plicable to arbitrary control transfers, such as function pointers or large conditional switch/case statements resulting in indirect jumps.

In general, T-Rex stipulated that if the dynamically observed wall-clock delta between checkpoints exceeds the WCET bound then an excess amount of instructions must have been executed. Such a bound violation provides an indication of a security breach. In contrast to coarser code sections with conditional control flow, static timing analysis on these straight-line execution regions yields tight WCET bounds. Return-from-subroutine code comprises a series of loads and stores to restore prior processor state and unwind the stack. Figure 4 depicts the communication structure of this method. It shows the application interfacing with the system twice to obtain values from the system clocks before checking the time-stamp delta against WCET bounds. A region that exceeds the path-based WCET bound may not necessarily cause the entire program to exceed its overall WCET bound. This is due to conditional execution where shorter paths may be taken during the remaining of execution, which compensates for the injection overhead.

As such, T-Rex is well suited for detecting attacks that could not easily be detected at task-level granularity due to deadline misses. This is because violation of micro-path WCET bounds is a necessary but not a sufficient condition for violation of a task's deadline or WCET bound.

The design of T-Rex integrates a state machine into the operating system. T-Rex requires the use of two separate calls whose order is tracked. In the motivating example, the attack would cause the timer initiated at the first checkpoint to never be canceled as the second checkpoint is skipped. A potential system intrusion is indicated by the corresponding timeout interrupt.

We also check the addresses of the checkpoint to insure that they fall within the address range of instructions as part of the state machine. Thus, any attack would have to return back to the application code to shut off the timer using the second checkpoint. For tight WCET bounds, even the simple code from the attack to jump to the second checkpoint would be detected. An attacker could potentially disrupt the control flow of the application by jumping to a non-corresponding second checkpoint if slack was available. However, such illegal control flow transitions would be detected with the T-ProT technique described in the following section.

4.2 Timed Progress Tracking (T-ProT)

Our second mechanism, *Timed Progress Tracking* (T-ProT), is depicted in Figure 5. T-ProT utilizes synchronous calls at security checkpoints to the scheduler and validates WCET bounds of longer code sections than T-Rex. The scheduler assumes the job of checking these bounds against actual elapsed time to provide separation between protected application and corresponding timing data as the latter resides within the operating system, *i.e.*, at a higher privilege level and in an address domain disjoint from the application's domain. Hence, our timed security does not rely on data / knowledge embedded within an application. Since such data can potentially be compromised, separation is a critical design decision.

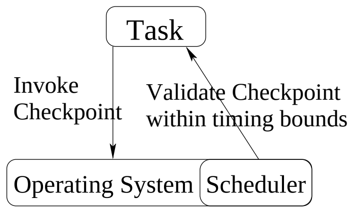

Fig. 5 Timed Progress Tracking (T-ProT)

Consider a scenario where the program diverts from the expected control flow. T-ProT detects several such intrusion scenarios, such as large sections of application code that are skipped or failure to return control to the base application, *e.g.*, by replacing the executable of a real-time task (through "exec" system calls). Upon encountering a timing checkpoint, instrumentation forces a synchronous scheduler call. The scheduler subsequently checks timing bounds for the code section between the previous and this checkpoint. It then activates a timeout equal to the WCET distance until the next checkpoint. An intrusion is flagged if no checkpoint is encountered before this timer elapses, i.e., when the respective timer interrupt is triggered (instead of being canceled when encountering the next checkpoint). Assuming that the application was not aborted prematurely due to an attack, we ensure that these checkpoints are always traversed when a job completes or its deadline expires. This is ensured by placing checkpoints in control-flow blocks guaranteed to be traversed during execution (*e.g.*, using post-dominator information [1]).

We controls the sensitivity of protection by determining the instrumentation points (checkpoints). In some algorithms, the best-case execution time may deviate significantly from the worst-case execution time. For instance, the insertion sort algorithm has a best-/worst-case complexities of O(n) and O(n^2), respectively. The difference between these bounds provides a substantial margin to orchestrate code injection. To overcome this problem, checkpoints need to be inserted such that time distribution is divided in a (uniform) manner to minimize the time between two consecutive checkpoints. An example of this would be checkpoints within the loops of the insertion sort that fire every k iterations. Here, the choice of k determines the strength in protection. We assure that there is sufficient slack in the task schedule to accommodate the timing checks, where scheduler invocations provide a call-back interface to trigger these checks. This also meshes well with code obfuscation techniques employing multi-version binaries: One can instrument at disjoint points for otherwise functionally equivalent binaries of the same application. Any attempt to systematically defeat our timed security approach becomes increasingly more difficulty for attackers by doing so.

A combined approach that uses these two methods bears additional benefits. By themselves, each approach can detect certain types of attacks. In combination, they become far stronger. T-Rex provides more fine-grained views of the internals of application timings thus allowing for targeted detection thresholds for code sections. However, this fine-grained approach has the shortcoming that detection is constrained to localized code sections. An attack may remain undetected by method one if the compromised code never returns to the original application code at all or only returns to locations that bypass these checks. In another approach to counter detection, WCET bounds stored within the application could be tampered with. In such cases, checks would fail to indicate bounds violations. This is precisely where T-ProT complements T-Rex.

T-ProT provides an ”outside-of-the-application” mechanism to ensure that specific security checks, placed strategically on the critical path of the application, are actually executed. These checks occur either when a job has completed or when a deadline expires, whichever happens first.

Checks allow the operating system to determine if injected attack code caused a job to bypass our checks or if a return never reverted back to the job's code at all. T-ProT, though operating within looser timing bounds at the macro level, uses timing data in a safer manner. It is protected from application-side buffer overflows because the data is stored inside the operating system scheduler, an address space in a different protection domain than that of the application. In combining the benefit of the two methods into a single system, we enable a more secure real-time environment suitable for the CPS domain.

4.3 Timed Address Execution Tracking (T-AxT)

Our approaches so far, T-Rex and T-ProT, both require application instrumentation for checkpoint placement. An attacker could exploit this fact through application-specific checkpoint bypass techniques, even though such bypasses are non-trivial to construct within given timeout bounds. To overcome this weakness, we designed T-AxT as an asynchronous checkpoint technique coexisting with unmodified application code. T-AxT exclusively utilizes the scheduler and timing bounds information provided at system start to maintain timed security. In T-AxT, the scheduler preempts the application upon timeouts. It then probes the PC value of the preempted application and compares execution progress to WCET bounds associated with the code section between the previous and current PC values of consecutive preemptions.

As T-AxT operates without synchronous calls, it presents an alternative to T-ProT. But with this technique, bounding the WCET of loops presents a challenge. As PC values are agnostic towards the progress of loops, the current iteration point within nests of loops needs to be known. We probe actual values of induction variables whose locations (registers/memory) are obtained via static analysis (offline, prior to system start). The scheduler dynamically evaluates polynomial functions parametrized by actual iteration points to determine if the WCET bound of a code section has been exceeded. Such sections may span multiple loop nests and iterations. Codes are systematically supplemented during static code analysis with an induction variable should any loops lack induction variables altogether.

We determine the WCET comparison bounds in either absolute or relative time in our experiments. We utilize WCET bounds *relative* to task activation when multiple execution paths exist. This allows us to eliminate path-aggregate over-estimations of WCET bounds due to conservative static timing analysis. In contrast, we utilize absolute WCET bounds for sequential straight-line code for finer granularity of timings. This duality is tailored to tighten WCET bounds checks in loops since scheduler preemption tends to occur in hot code regions, *i.e.*, predominantly within loop execution.

In practice, we mostly rely on checks of WCET bounds between two checkpoints at the highest nesting level. This interaction is depicted in Figure 6. The first check in the loop is calculated as an absolute checkpoint since no previous checkpoints exist. The second checkpoint is measured as a delta from the previous checkpoint. This strengthens timed security as a means of intrusion detection as bounds are tightened by this method.

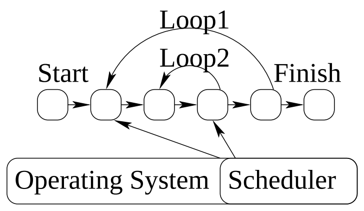

Fig. 6 Timed Address Execution Tracking (T-AxT)

Since we utilize application instrumentation for two of the timed security techniques, the overall real-time task set has to be reanalyzed after instrumentation. This ensures that WCET bounds include the instrumentation code. Timing checks by the scheduler have to be accounted for as well before the real-time schedulability is reassessed. To avoid that such overheads becomes excessive, which might render task sets infeasible in terms of real-time scheduling, checkpoints are selected based on profiled frequencies that are representative task executions in our experiments. Any detected timing bounds violation indicating intrusion further needs to result in evasive actions, such as transitioning to fail-safe states, *e.g.*, through a mode change that replaces all existing tasks with a new task set governing a shut-down sequence and network isolation. Such evasive actions are beyond the scope of this work, i.e., the focus of this work is on time-based intrusion detection.

Summary: Table 1 presents a high-level comparison between our novel techniques. T-Rex protects against buffer overflows commonly exercised on the return path of function calls, which requires fine-grained, cycle-level checks in conjunction with tight bounds on this return path. The overhead of such checks can be high if functions are called very frequently in tight loops, but could be lower when code is inlined instead of calling functions in these loops. Since protected code sections are sequential and bounds are tight, virtually all buffer overflows can be detected, and no source code changes are required. In contrast, T-Prot requires source code changes to insert checkpoint calls in between which code sections are timed at a medium grain. Due to variable loop bounds and conditionals in these code sections, bounds are moderately tight, and so is the overhead assuming that calls are inserted judiciously. T-AxT has comparable, if not lower, overhead than T-Prot. It tracks progress using the program counter and loop invariants, which allows coarser bounds checking, yet without requiring source code changes as checks are integrated into the scheduler at preemption points with low overhead.

Property	T-Rex	T-Prot	T-AxT
Timing Method	return path	add checkpoints	at scheduling points
Progress Tracking	cycles/instr.	time of a task	inspect loop counters
Granularity	fine: instructions	medium: blocks	coarse: interrupts
Bounds Tightness	very tight	moderate	loose
Cost/Overhead	high	moderate	low
Intrusion Detection	very strong	moderate/high	moderate/lower
Source code changes	none	insert calls	none

Table 1 Comparison of Intrusion Detection Techniques

5 Implementation

We implemented the mechanisms of T-Rex, T-ProT and T-AxT in two different experimental frameworks. The first one that combines static timing analysis with architectural simulation yields simulation results in experiments detailed later. The second realizes dynamic timing analysis on a concrete embedded system hardware platform, where we subsequently obtain runtime results. We tested both of our implementations using a set of C-Lab benchmarks [11].

C Benchmark	Function
adpcm	Adaptive Differential Pulse Code Modulation
cnt	Sum and count of positive and negative numbers in an array
lms	An LMS adaptive signal enhancement
srt	Bubble Sort
fft	Fast Fourier Transform

Table 2 C-Lab Benchmarks

5.1 Simulation Framework

Figure 7 depicts the overall experimental framework. We enhanced a static analysis framework as discussed in Section 3 to support check-pointing instructions. These check-pointing instructions allow us to determine the worst case cycle time at which a particular instruction finishes execution. This information is essential to determining the WCETs between two consecutive checkpoints under T-ProT. We further utilize a custom SimpleScalar processor simulator [9] enhanced to support multitasking and scheduler threads / tasks, which we exploited to implement earliest deadline first (EDF) scheduling [31]. The instruction set architecture for this simulator is PISA. This matches the input assembly utilized by our timing analysis tools. For the purpose of this work, we assess benchmark results in SimpleScalar that match the configurations of the static timing analysis tools.

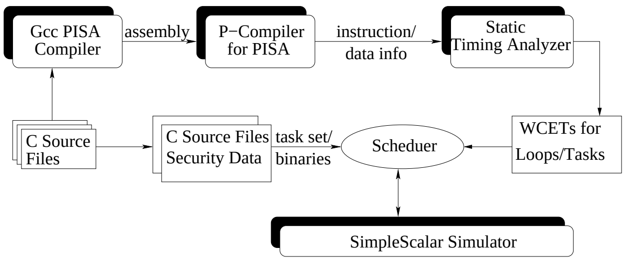

Fig. 7 Framework

As discussed before, these configurations provide a lower bound on the amount of code injection that may remain undetected. If we were to relax our configuration constraints, WCET bounds obtained by static analysis would become less tight implying that an attacker could potentially execute more instructions prior to being detected. To assess this trade-off, we also deployed T-Rex and T-ProT on a concrete hardware platform (see below).

The scheduler in the SimpleScalar framework supports multiple preemptive and non-preemptive scheduling algorithms. For the course of this work, we used a preemptive EDF schedule to most accurately show the side effect of our mechanisms on the scheduler itself. The scheduler is customized to support relative time for each thread aggregated during preemptions and at security checks of a task to most accurately track execution progress.

The cache configuration for both the static cache simulator and the timing analyzer were configured without data caches and with perfect instruction caches, *i.e.*, with an I-cache capacity exceeding that of our largest program sizes so that we only had to account for cold misses. The choice of the cache configuration parameters was intentional as our objective here was to assess a bound on detectable code injections. In other words, given the tightest possible timings on application code, we wanted to determine the largest number of cycles that would remain undetected by our security-enhancing mechanisms. For such a metric, the smaller this threshold, then stronger the protection will be by our mechanisms.

For T-Rex, SimpleScalar enhancements include two system calls to query timing information (a) before a return from a function / method, and (b) at the destination of a function / method return and compare the difference to static bounds. We utilize a timer and also verify correct sequential ordering of these calls. If call one was issued without the other, a control-flow violation (intrusion) is detected, that would result from a buffer overflow attack that returns control flow past the second call. Subsequently, a system-defined action, such as transitioning into a fail-safe state, can be initiated. In effect, the imposed call ordering represents a security side-check that provides the means to detect certain attacks missed if only execution cycles were checked. For example, if an attacker were to execute injected code and then transfer control to the instructions past our second system call in an attempt to bypass our imposed security, the absence of the second system call would be detected at the next return from a function when another instance of the first system call is issued. Call sites are identified by their call stack / PC and frame pointer signature so that calls from injected attack code are easily identified.

5.2 Embedded Hardware Framework

We also experimented with an actual embedded hardware platform, namely the DSK6713 kit from Spectrum Digital. These experiments combine dynamic timing analysis with implementations of T-Rex and T-ProT. The experimental board has a Texas Instruments C6 (TMS320C6713) DSP chip running at 150MHz featuring a 32-bit processor with Very Long Instruction Word (VLIW) architecture, eight independent functional units that can execute up to eight instructions per cycle, fixed and floating point arithmetic, 2 levels of caching and up to 256KB of on-chip SRAM programmed under Code Composer Studio v3.1. All programs were written in C and assembly.

This board is also utilized in a CPS project for controlling power devices (solid state transformers) in a renewable energy project (solar and wind power generation in microgrids). There, the TI DSP controls silicon-based solid state transformers during the DC/DC conversion from low to high voltage levels. These transformers represent the link between micro-grids and the regional power grid backbone. The project focuses on controlling renewable energy sources locally and feeding their power into the regional grid without disruptions. Due to the decentralized nature of micro-grids, software security is deemed critical in power grids as malicious attacks could potentially damage equipment upstream. Such damage would impact at a minimum entire suburbs and require manual maintenance. Hence, our work and the choice of platform are very much motivated by a concrete CPS scenario.

The effectiveness of our mechanisms depends on how accurately we can determine the WCET bounds and how tight these bounds are relative to average execution times. The objective of this study is to assess the lower bounds on tightness. In the experiments on the embedded platform, WCET bounds are determined by dynamically timing execution paths under worst-case scenarios while running the program on a cycle-accurate simulator from Texas Instruments that simulates the C6713 processor along with its on-chip peripherals. Executing the actual code segment repeatedly on this simulator using worst-case inputs and hardware settings provides the observed maximum number of CPU clock cycles for a given code segment. We then convert these dynamically determined WCET cycles into microseconds by considering the CPU clock speed. In addition, we tried to reduce the effect of any factors that adversely influence tightness of WCET bounds.

The following is a list of such factors on the given hardware platform configured for maximum predictability:

Caches: The TMS320C6713 has separate L1 instruction/data caches and a unified L2 cache. We chose to disable all caches resulting in tight WCET bounds relative to average timings. Enabling caches would considerably alter the WCET bounds to deviate more significantly from their average case, yet still preserve the safety and validity of upper WCET bounds.

SRAM *vs.* DRAM: In our experiments, the program code and data reside in static, non-volatile memory (SRAM), *i.e.*, we do not utilize dynamic, volatile memory (DRAM) at all. The TMS320C6713 processor has 256KB of on-chip SRAM. If, in contrast, DRAM were used, we would need to account for periodic self-refresh cycles. The DRAM controller refreshes row data in different banks of the DRAM in a row-cyclic manner. This issue is common to all embedded platforms utilizing dynamically buffered memory and refresh delays are known to present a challenge in real-time systems.

During such self-refresh cycles that last for a few microseconds, the CPU bus remains busy. Any attempt to read from the DRAM or other external devices would then stall the processor as long as the self-refresh cycles are in progress. These self-refresh cycles are asynchronous events as far as program execution is concerned and completely transparent. They would thus affect the timing calculations used in T-Rex. However, strategies exists for exactly measuring the duration of DRAM self-refresh cycles [3] and to treat DRAM refresh as a higher priority task [6, 7]. Since the refresh overhead challenge is orthogonal to our work and our aim was to assess how tight WCET bounds could become, we decided to eliminate these overheads in experiments by avoiding DRAM altogether and exploiting SRAM instead.

Compiler-Generated Runtime Overhead: In our current experiments we coded all tests and runtime / operating system code in C to reduce the amount of runtime overhead added by the compiler. Hence, instructions between the first and second instrumentation points around a function return of T-Rex are limited to stack unwinding operations and register restores. An object-oriented language, such as C++, would further add destructor overhead for objects locally declared within the method. Since destructors are user defined, providing tight WCET bounds for them presents a challenge.

Our implementation features a layered system architecture depicted in Figure 8. We ported a commonly used real-time operating system, MicroC OS II [21], which supports fixed-priority preemptive scheduling. We then implemented a scheduler based on rate-monotonic analysis (RMA) [25] on top of MicroC OS II. This scheduler supports threads of arbitrary periods imposing strict execution time control. Failure to complete by a deadline results in preemption and rescheduling during the next period. Most hard real-time systems use similar schedulers in order to guarantee deadline constraints on periodic tasks. We also provide synchronous application checkpoint calls for implementing T-ProT and monitoring of aggregate execution time per task with a one microsecond precision, but we exclude the time spent inside interrupt service routines and scheduler overheads (due to the complexity of measuring these).

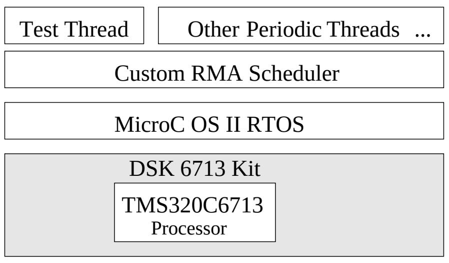

Fig. 8 System Architecture

6 Experiments

We first report the results of our simulation environment before discussing measurements obtained on the embedded hardware platform.

6.1 Common Attack Cycles

In the following, we first consider common shell codes used on Linux systems to determine typical attack scenarios. This is necessary since timing values of actual attacks for embedded systems are sparse in literature, at best. Metasploit, a repository for shell attacks, contains approximately 35 different Linux/Unix shell code examples of the same fundamental structure. A jump in the first line of the shell code transfers to another location within the shell code. This aids in determining the relative offset for addressing. An “exec” system call then invokes a command of the attacker’s choice. The most common examples found on Metasploit are useradd, shell, and tcp open directives.

Figure 9 provides measured timing values for common portions of attack code. We measure the average cost of execution from the hijacked return to the first instruction in the shell code (“Start”) and the average time of an execution system call (“Execpl”) with null arguments. If actual values are passed, measurements are significantly larger. *E.g.,* passing “Chmod”, a common attack to modify file permissions, dramatically increases the cycle overhead. The motivation here is to consider the effectiveness of our methods, and these examples of common shell code attacks provide realistic timings to this end.

Location	**Cycles**
Start	90
Execpl	2,800
Chmod	5,151,720

Fig. 9 Shell Code Timings

		No Caches		4KB I-Cache	
Program	**Function**	**WCET**	**Sensit.**	**WCET**	**Sensit.**
SRT	Initialize	39	5	21	13
SRT	BubbleSort	39	5	30	13
LMS	LMS	39	5	30	9
FFT	FFT	39	5	25	8
ADPCM	Encode	39	5	30	22
ADPCM	Decode	39	5	30	22

Fig. 10 T-Rex WCET and Sensitivity cycles

6.2 Simulation Experiments

In our implementation, T-Rex utilizes an *absolute* task timer to determine the total time since the simulation start. T-ProT and T-AxT are exercised in a modified preemptive real-time scheduler under the SimpleScalar environment developed elsewhere [31] to keep an *aggregate* timer for each of the executing jobs. This aggregate timer is compared against WCET bounds from static timing analysis. It is further saved in the scheduler-maintained thread control block at preemption and restored at reactivation. The value is reset at thread / task completion to prepare for the execution of the task’s next periodic job.

Timed Return Execution (T-Rex) Results

The attack outlined in Figure 1 was successfully detected by T-Rex as a buffer overflow since the injected code accounts for 14k cycles, which far exceeds its detection granularity of 5-22 cycles. Under legitimate sensor inputs, the sample program produces the correct output with an additional 40 cycles relative to the application itself. Figure 10 shows the sensitivity results of T-Rex for varying benchmarks and their respective functions. In this experiment, the attack code, after executing its injected code, returns to the exact spot in the code that the original return for a call would have jumped to. The table then reports the WCET in cycles for the return sequence as reported by timing analysis (WCET in column 3) and the number of slack cycles that would remain undetected (sensitivity in column 4), first without considering caches and in next two columns with a 4KB instruction cache.

The slack amounts to the difference between WCET and actual execution time, the latter of which is observed from SimpleScalar simulation. The WCET bound is extremely tight since T-Rex assesses time on a straight-line path of the control flow. Hence, the window of vulnerability is restricted to a sensitivity of 5 cycles without and 8-22 with caches. If an attack was to go undetected, it would have to be constrained to such a small amount of code as an injection. These results provide a lower bound. The upper bound for undetectable injections is given by the T-ProT or T-AxT methods, which address larger injections and omission of code sections in favor of injected code. However, it would be non-trivial to disguising the side effects of polluting stacks and registers.

The timing bounds and subsequent security checks for straight-line code are very precise as results in Figure 10 illustrate. Instruction cache effects loosen these bounds proportionally to the cache miss penalty of 10 cycles (as seen for ADPCM). Overall, this leaves little room for injected code to go undetected.

Timed Progress Tracking (T-ProT) Results

T-ProT relies on synchronous scheduler checkpoints to dynamically detect intrusions by WCET bounds violations. Its effectiveness is assessed by the results in Table 3, which reports checkpoints between adjacent instrumentation points in the control flow for each application. For example, checkpoint 0-1 denotes execution from entry of main() to a later basic block in CNT, 2-3 and 3-4 denote loop entry and exit, respectively, while 3-2 denotes a back-edge within the outer and inner loops, respectively (see Figure 11). For these code sections, Corresponding WCET bounds (column 3) and sensitivities (column 4) are reported in cycles.

Several checkpoints were instrumented in benchmarks as illustrated for CNT in Figure 11:

1. immediately after the original variable declarations but prior to the invocation of loop 1;
2. within the outer loop just prior to the inner loop invocation;
3. in the inner loop with logic surrounding it to only perform the check during half way through the total iterations of the inner loop; and
4. in the final block of the application just prior to exiting.

		No Caches		4KB I-Cache	
Program	**Checkpoint**	**WCET**	**Sensit.**	**WCET**	**Sensit.**
LMS	0 - 1	1,500	44	844	173
LMS	1 - 2	5975	65	3279	774
LMS	2 - 2	17199	259	8699	2120
LMS	2 - 3	11330	210	5549	1430
FFT	0 - 1	1,600	195	846	228
FFT	1 - 2	950	54	697	220
FFT	2 - 2	19,283	2,787	13,955	5,334
FFT	2 - 3	12,709	1,997	9,451	3,831
FFT	3 - 3	5,084	460	3,150	659
FFT	3 - 4	208	48	120	49
CNT	0 - 1	1814	120	786	147
CNT	1 - 2	69	9	46	14
CNT	2 - 3	14083	283	4341	1493
CNT	3 - 2	13599	239	4199	1481
CNT	3 - 4	13726	266	2760	1534

Table 3 T-ProT WCET and Sensitivity cycles

T-ProT has a coarser granularity for the reported bounds on undetectable injections as indicated by the results in Table 3. These bounds, while smaller in some case, range up to nearly 5k cycles on the upper end. Hence, scheduler callbacks result in less sensitivity than return path instrumentation. The more complex control flow (than just straight-line code as in T-Rex) causes this lower sensitivity.

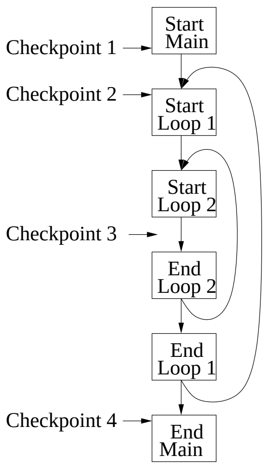

Fig. 11 CNT Control Flow

Checkpoints are scattered throughout the application as they may cross loop levels, as indicated by Table 4. This reduces the tightness of WCET bounds. WCET bounds of a loop iteration are generally less tight than straight-line code due to fluctuations in the number of iterations or conditionals inside the loop body. To obtain safe worst-case results, we have to conservatively calculate the worst case scenario (upper bound on loop iterations, longer path for conditional execution) in our static analysis.

The utilization of instruction caches, as depicted in the last two columns of Table 3, has an impact on the overestimation. This is due to the fact that relative checkpoints tend to not incur cache misses as most cold misses occur prior to the first checkpoint hit.

These scheduler checks result in strengthened support for security. Moreover, T-ProT is quite versatile in that it may be used to instrument code sections at arbitrary points in the application. This makes T-ProT suitable to detect compromised subroutines in a targeted manner.

There are additional security benefits to using T-ProT. Timing bounds on preemption require a look-up of the previous checkpoint and a comparison of the current timing values with the corresponding WCET bounds. When factored into the application execution, this cost is hardly noticeable and requires only insignificant additional slack in the real-time schedule of the task set at the benefit of *more secure cyber-physical systems* (see Section 8).

Table 4 T-ProT Checkpoint Hits

Program	**Total Checkpoints**	**Total Hits**
LMS	3	203
FFT	4	114
CNT	4	132

Timed Address Execution Tracking (T-AxT) Results

The coarsest granularity of our mechanisms is provided by T-AxT. It is also the most difficult to attack directly because it resides within the kernel and is not triggered by checkpoints from tasks. The periodic timer for these results was set at 20k cycles on a 100 MHz processor clock in simulation. This value was chosen to balance overhead, *e.g.*, SRT required 2051 checkpoints during job execution (see Table 5).

The coarser granularity of T-AxT is due to aggregation of conservative bounds during static timing analysis and approximate matching of PC values with WCET bounds. WCET values were associated with the next-smaller blocks of code relative to a PC value to conserve storage overhead for WCET bounds. The LMS benchmark generally retained the highest difference in cycle measurements *vs.* actual time. This is due to the complexity and size of multiple inner loops within LMS. The overestimation of WCET could be decreased using a finer granular configuration but at a larger storage cost. The benefit of T-AxT is its ability to bound the WCET of PC-constrained code sections within or across loops and to verify that the job's execution meets these bounds. For a given code section, bounds violations are a sufficient indication of intrusion.

Program	Period	WCET	Sensit.
CNT	20,000	21,225	1,225
CNT	20,000	28,200	8,200
CNT	20,000	27,750	7,750
CNT	20,000	27,225	7,225
CNT	20,000	26,775	6,775
LMS	20,000	30,991	10,991
LMS	20,000	28,434	8,434
LMS	20,000	33,473	13,473
LMS	20,000	28,918	8,918
LMS	20,000	32,597	12,597
SRT	20,000	23,400	3,400
SRT	20,000	24,128	4,128
SRT	20,000	22,701	2,701
SRT	20,000	22,372	2,372
SRT	20,000	22,701	2,701

Table 5 Timed Address Execution Tracking

6.3 Experiments on an Embedded Hardware Platform

The DSP hardware provides a platform for the next set of experiments, where both T-Rex and T-ProT were implemented. The first experiment features the benchmark ADPCM deployed as a single periodic task. The code of this task is enhanced by T-Rex to provide timed security. The single-task constraint allows us to control the experiment by eliminating additional preemptions between first and second calls that obtain clock values. We determined that the calls themselves add only negligible overhead. We used “assert” statements at checkpoints to check timing bounds. The tested assertion here is given by the comparison of the actual time elapsed since obtaining the first clock value and the expected WCET bound.

Figure 12(a) depicts the output of assertions that were added for trace visualization purposes. The first word in every output line indicates the ADPCM function instrumented, followed by the result of the assertion indicating if it passed or failed. The number before '$>$' indicates the WCET bound in microseconds for the corresponding function return and the number after '$>$' indicates the actually measured time for the same in microseconds.

Assertions compare these times with a predetermined WCET bound, which in this case is determined to be about 3.1 μsecs (rounded up conservatively to 4) for all functions using the C6713 device cycle-accurate simulator. The output shows that all timed return path values are within a range of 1-2 μsecs. Hence, all the assertions pass, *i.e.*, no timing violations were detected implying that no intrusion was seen.

In the second experiment, calls to a dummy function are issued after obtaining the first clock value but before a return from a function. In other words, we created a code injection scenario. The dummy function simply executes an empty loop (no-op) for 100 iterations before returning to the caller. This simulates code injection

```
scalel: ASSERT PASSED 4 > 1
dh: ASSERT PASSED 4 > 2
uppol2: ASSERTPASSED 4 > 2
uppol1: ASSERT PASSED 4 > 2
encode: ASSERT PASSED 4 > 2
filtez: ASSERT PASSED 4 > 2
filtep: ASSERT PASSED 4 > 1
```

```
scalel: ASSERT PASSED 4 > 1
dh: ASSERT PASSED 4 > 1
uppol2: ASSERT PASSED 4 > 1
uppol1: ASSERTPASSED 4 > 2
encode: ASSERT FAILED 4 > 16
filtez: ASSERT PASSED 4 > 1
filtep: ASSERT PASSED 4 > 1
```

Fig. 12 (a) All Asserts Pass (b) Some Asserts Fail

that returns to the original control flow without harming stack values, *i.e.*, the only noticeable effect is time dilation. Results of this experiment are depicted in Figure 12(b). As illustrated by the results, code injection through the dummy function resulted in a large deviation in elapsed time between obtaining clock values on the return path. Notice that even ten iterations accounting for 1.4 μsecs would suffice for detection as $2.0 + 1.4 > 3.1$, which gives an attacker little room for devising malicious code. The next experiment features a set of periodic tasks with mixed periodicities (containing smaller and larger periods than ADPCM) to co-exist with the ADPCM task. We further experimented with explicit sleep statements prior to obtaining the first and second clock values in order to force preemptions. As expected, assertions indicated intrusions in all these cases. These results are omitted here, since they resemble those reported in the previous figures.

Finally, T-ProT was implemented on the embedded hardware platform. As before, the WCET bounds between various checkpoints are obtained as the maximum cycle count for executing the program in a loop on the C6713 cycle-accurate simulator under worst-case conditions and inputs plus complete path coverage. This cycle bound is then converted into execution *time* by adjusting for the CPU clock speed before comparing with measured time on the hardware at a checkpoint. Our RMA scheduler provides a built-in mechanism to remember the previous checkpoint and assert the validity of the latest checkpoint. Table 6 shows the calculated WCET bounds and observed runtimes for FFT on the embedded TI DSP hardware platform. All checkpoints pass in this experiment indicating a safe execution in the absence of code injection (columns 2-4).

Table 6 Checkpoints of T-ProT for FFT on TI DSP

	No Injection			Code Injection		
Chkpt. #	**WCET**	**Actual**	**Chkpt**	**WCET**	**Actual**	**Chkpt**
Chkpt 0 - 1	3	2	pass	3	2	pass
Chkpt 1 - 1	5	3	pass	5	3	pass
Chkpt 1 - 2	7	5	pass	7	5	pass
Chkpt 2 - 2	4	3	pass	4	3	pass
Chkpt 2 - 3	3	2	pass	3	16	fail

We next injected code that executes between checkpoints 2 and 3 (depicted in columns 5-7 of Table 6). A small loop is introduced between these two checkpoints to simulate code injection. Results of Table 6 indicate that all tests between checkpoints 2 and 3 fail implying a detected intrusion. Overall, we have shown that our mechanisms facilitate intrusion detection in both preemptive and non-preemptive multi-tasking real-time environments. Thus, CPS applications can universally benefit from these approaches.

7 Trading off Security against Timeliness

The aim of our work is to increase the level of protection against attacks in systems at the cost of executing additional routines that monitor and check the system behavior. In cyber-physical systems with real-time constraints, these instrumentation and time validation checks affect system utilization and thus real-time schedulability.

Our sample attack in Section 2 shows that embedded systems with network connections, such as CPSs, are vulnerable to cyber attacks. Reports in practice reinforce this fact. Most notably, worms have entered monitoring equipment and disabled a safety system at a nuclear power plant [24]. In another incident, a virus reportedly spread past firewalls into the accounting system of the main Australian power company, which did not implement proper physical network separation between accounting and power control subsystems [33]. Further damage was only contained by reconfiguring servers between the two subsystems to prevent the virus from spreading uncontrolled into the power control subsystem.

These are just two examples illustrating the urgency of providing guards against cyber attacks in the CPS realm. Our timed security is one such technique readily deployable to complement existing intrusion detection techniques. The rationale of such deployment is to further strengthen security as a single protection mechanism can often be defeated by itself, yet a set of mechanisms is much harder to circumvent. In practice, the inherent costs of security are well justified. We also observe that many real-time systems provide sufficient slack in a task schedule so that security mechanisms could be accommodated under feasible schedulability. After all, real-time systems only have to ensure timeliness in the sense that deadlines are met. As long as deployed security methods, such as timed security, impose overhead within deadline bounds, correctness is guaranteed.

Conversely, systems with tight slack may limit the level of security that can be realized. Depending on vulnerability and criticality assessment, such networked systems may need to be redesigned for more powerful hardware targets, or a paradigm is needed to provide the ability to selectively augment code with security measures. Selectivity amounts to a tradeoff between availability of slack to meet deadlines and safety and vulnerability considerations of code sections. T-Rex, for instance, increases the execution time of an application due to its inherent instrumentation. This overhead is assessed in the results of Section 8.

Return-path instrumentation results in the invocation of only few checking instances at execution time in many embedded applications since the bulk of the work is performed in loops whose bodies do not contain function calls, thus resulting in negligible timing overhead. In codes containing hot spots in tight inner loops with function calls, in contrast, security checks impose a significant overhead that may easily exceed the available slack. In such cases, application code should be refactored based on transformation techniques such as inlining, single caller function specialization, which avoids allocating a new stack frame in place (commonly performed by the Intel compiler), or reduction of function call frequencies through restructuring. In future work, the balance between such transformations and security overhead of T-Rex to target given slack margins should be studied. Overhead is imposed by synchronous upcalls and timeout preemptions under T-ProT. This results in scheduler activations to subsequently check if the application operates within expected timing bounds. The overhead of the former (upcalls) is more significant than that of the latter (timeouts) as timeouts are only triggered upon an intrusion but otherwise canceled. This method should be used in conjunction with selective placement of checkpoints using strategic and statistical means (*e.g.*, random placement and random activation). Attacks would also become more difficult as random activations strengthen security.

The overhead of T-AxT can easily be controlled through its scheduler activations. Should frequent checks be required, timer interrupts would have to be triggered in shorter intervals adding to the overhead of interrupt service routines. The overall objective is to provide adequate coverage of checkpoints to maximize overall security within the given timing constraints. All methods are designed to allow selective instrumentation, but the details of such placements and their trade-offs are beyond the scope of this chapter.

Overall, we developed three security-enhancing methods based on timing information already inherent to CPS real-time control systems. Their overheads have acceptable costs when properly tuned for providing security without compromising timeliness. By adjusting the frequency of dynamic checks, particularly for less critical sections, one can trade off overheads for an increase in the vulnerability level. The trade-off between overhead and level of security is common in general-purpose computing, yet the implications on timeliness add another equation to this trade-off. Our techniques target real-time CPS where system criticality outweighs performance concerns making security a mandate rather than an option. A future direction of research might investigate the viability of additional security measurements. Some of them are quite feasible, such as exploiting average case execution times for checks on timing outliers. Such methods are probabilistic and may result in large numbers of false positives. More accurate results with lower false positives should be expected based on parametric models of execution time that take actual loop trip counts of dynamic execution into account, both for BCET/WCET bounds and average times [31, 29]. Early warning indicators could be dynamically triggered to activate stringent security checks that bare higher costs or to reduce system functionality in order to limit potential damage to the *physical* side of the CPS application.

8 Instrumentation Overhead

We also designed experiments to assess the cost of instrumentation relative to the performance costs of each of our methods. Table 7 depicts these overheads in percent relative to the application's base execution time without the security methods. We distinguish the "default overhead" and "scaled overhead". The former corresponds to the experiments of Section 6 while under the latter, variations on the frequency of intrusion checks are featured.

Overheads (default) range from 0.22% to 1.54% for three of the four benchmarks under T-Rex. Such overheads are negligible assuming just minimal slack in a real-time task schedule. The higher overhead of 18.71% for ADPCM is due to its modular structure compared to other benchmarks. It consists of several small functions that are called within a loop. Thus, T-Rex checks are invoked more frequently at a deeper nesting level than in other benchmarks. Code restructuring, such as inlining, reduces this overhead to that of the other benchmarks. For example, after inlining calls at the inner-most loop levels for ADPCM, the T-Rex scaled overhead was reduced to just 0.32%, as depicted in the last column of table 7. For the remaining benchmarks, default overheads did not justify any inlining so no scaled overheads are reported for T-Rex. Occasional code restructuring only imposes an insignificant performance cost.

Depending on the application instrumentation frequency overheads for T-ProT vary. The default overhead for the experiments in Section 6 ranges between about 7% and 16%. Such instrumentation with a high level of coverage incurs a sizable performance penalty in performing finer grain security checks. The scaled overheads in last column of Table 7 of about 3%-8% correspond to a reduction in the number of instrumentation checkpoints by half relative to the default method. This is accomplished by selective activation of instrumentation checkpoints but selective placement would be a valid alternative as well.

Tunable performance overhead is provided by T-AxT depending on the frequency of the periodic wake up that initiates the intrusion check. We used a periodic wake up of 20,000 cycles, which provides a reasonably frequent security check at a dynamic overhead comparable to that of T-ProT with a constant default overhead of approximately 16%. The scaled overhead amounts to about 8% for a 40,000 cycle instrumentation period (see last column of the table).

Overall, overhead is observed to scale linearly with instrumentation frequency for all of our techniques. Such scaling is easily controlled (a) for T-AxT through selection of periods, (b) for T-ProT through rate control and (c) for T-Rex through inlining, rate control or a combination of both.

9 Related Work

Generic security features have been considered in the context of scheduling of real-time application tasks in past work. Often, certain out-of-the-box security mecha-

Table 7 Dynamic Performance Overheads

Method #	Benchmark	Default Overheads	Scaled Overheads
T-Rex	SRT	0.22%	N/A
	LMS	1.54%	N/A
	ADPCM	18.71%	0.32%
	FFT	0.021%	N/A
T-ProT	LMS	7.55%	3.68%
	FFT	16.17%	7.92%
	CNT	10.05%	4.92%
T-AxT	LMS	15.89%	7.94%
	SRT	15.89%	7.94%
	CNT	15.89%	7.94%

nisms are applied at the cost of ensuring timeliness while arguing that security is improved [40, 46].

Past work on embedded systems security has focused on sensor networks including remote memory verification and network-related anomaly detection at the packet or application level [36, 48, 49, 47, 45]. Timing analysis is considered in literature as a means to reverse-engineer encryption techniques [35] instead of utilizing it for protection. The emphasis of this work is on utilizing timing analysis bounds to detect code injection attacks [28].

Shao et al. use a hardware/software combination to detect attacks [38], which is closely related to our work. The first technique adds a new stage to the processor pipeline to check on an address before data is written to it. If the value is greater than that of a special register delimiting vulnerable stack regions then write is denied. The second technique uses a new "sjmp" instruction to XOR the write address with the value stored in the special register to assess validity of the jump target. Other approaches rely on hardware buffers to store return addresses [20] when buffer space is available. These techniques do provide security with negligible performance overhead but at the cost of specialized modifications to hardware. Our work does not require special hardware support.

Significant work has been performed in the area of security of general-purpose and server environments in which attacks are more prevalent. These systems are generally much larger and more difficult to impose restrictions on due to their general-purpose nature. Efforts at reducing opportunities for code injection in these environments has resulted in concepts such as canary value placement. Buffer overflow may be detected in general-purpose systems by placing canaries adjacent to the return address on the stack, which may be overwritten in an attack [13]. If a tampered canary is detected prior to transferring control at a return, the program aborts itself. Canaries are typically pseudo-randomized at compile time to increase the difficulty of success during buffer overflow attacks. Thus, simply placing the canary value onto the stack next to the return address, which avoids detection for known canaries, becomes challenging [13]. Yet, even pseudo-randomized canaries can be exploited in systematic repeated attacks.

In general-purpose systems, another protection mechanism employed is to utilize address-space layout randomization (ASLR) [37]. The stack is placed in a hard-to-guess location in the memory. If an attacker attempts to jump to code placed on the stack, it becomes difficult to infer absolute stack addresses where attack code may have been injected. This method is best suited for systems that employ 64-bit addressing spaces, *i.e.*, where ample room for stack placement exists such that repeated brute-force attempts are statistically ineffective. However, such techniques may be circumvented by repeated attacks in a space-constrained embedded real-time system with 8/16/32-bit address spaces [37].

Dan et al. [15] discuss power grid challenges while Mitchell and Chen [27] provide a survey of CPS intrusion detection approaches of which we mention a few but otherwise refer the reader to the survey. Different detection techniques (knowledge vs. behavior) and deployment scenarios (host vs. network) are discussed. Many systems employ behavior-based techniques [23, 2], optionally using domain-specific knowledge [17] and are often targeted at wireless communication [39]. Several approaches follow a model-based approach utilizing varying methods ranging from regular expressions [12] over Petri nets [26] to neural nets [16].

This work extends our prior publication [50] by the following contributions: It contains more detailed explanations of our technical approach, tightness of bounds for detecting intrusions, motivational scenarios, more illustrative discussions of examples, a discussion of deploying hybrids of the proposed methods in a mutually complementing manner, consideration of scheduler interactions, discussions on resorting to fail-safe modes, measures to ensure tight WCET bounds, future early-warning enhancements, and a discussion of future work on cyber security specific to CPS.

10 Conclusion

Our work contributes three novel software methodologies that provide enhanced security in deeply embedded real-time systems, such as Power Grid control devices. We attain elevated security assurance through two levels of instrumentation that enable us to detect anomalies, such as timing dilations exceeding WCET bounds. (1) T-Rex: Tight timing bounds of selected code sections are obtained during static timing analysis at no extra cost during the required schedulability analysis and are subsequently utilized to monitor execution during run-time. Buffer overflow attacks are detected due to exceeded WCET bounds upon return path instrumentation for code injections as small as 5-22 cycles. (2) T-ProT: Application instrumentation issues synchronous scheduler calls to assess timing bounds validity for precisely delimited sections of code. T-ProT by itself uncovers coarser-grain injections between 9 and 5k cycles at controllable overhead and complements T-Rex. (3) T-AxT: Asynchronous scheduler-triggered validations of timing bounds are performed for approximated sections of code, which, compared to T-ProT, obviates application instrumentation, results in low overhead and complements T-Rex. Attacks uncovered by T-AxT alone are consequently the coarsest grained. These security checks can be strategically scheduled to utilize otherwise idle time in the schedule. Upon validation of timing bounds, no action is taken. Conversely, upon violation of bounds, an alert is raised that provides an opportunity to reduce system functionality, revert to a fail-safe state or shut down the system altogether pending further investigation/assessment. To the best of our knowledge, such detection of system compromises through micro-timing information is a novel contribution to real-time systems.

Within the realm of this work, overheads on performance and tightened security should become more balanced or tunable by a "dial". This may also include the exploitation of average-case execution time in statistical sanity checks or probabilistic timing analysis-based systems or parametric models [5, 29]. More gradual warning systems might provide several steps of reduced functionality while raising the bar for intrusions if threats are detected, similar to the Simplex approach for reliability [4, 14].

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques, and Tools. Addison-Wesley (1986)
2. Asfaw, B., Bekele, D., Eshete, B., Villafiorita, A., Weldemariam, K.: Host-based anomaly detection for pervasive medical systems. In: Risks and Security of Internet and Systems (CRiSIS), 2010 Fifth International Conference on (2010)
3. Atanassov, P., Puschner, P.: Impact of dram refresh on the execution time of real-time tasks. In: Proc. IEEE International Workshop on Application of Reliable Computing and Communication, pp. 29–34 (2001)
4. Bak, S., Chivukula, D., Adekunle, O., Sun, M., Caccamo, M., Sha, L.: The system-level simplex architecture for improved real-time embedded system safety. In: IEEE Real-Time Embedded Technology and Applications Symposium, pp. 99–107 (2009)
5. Bernat, G., Colin, A., Petters, S.: Wcet analysis of probabilistic hard real-time systems. In: IEEE Real-Time Systems Symposium (2002)
6. Bhat, B., Mueller, F.: Making dram refresh predictable. In: Euromicro Conference on Real-Time Systems, pp. 145–154 (2010)
7. Bhat, B., Mueller, F.: Making dram refresh predictable. Real-Time Systems **47**(5), 430–453 (2011)
8. Braberman, V., Felder, M., Marre, M.: Testing timing behavior of real-time software. International Software Quality Week (1997). URL citeseer.ist.psu.edu/braberman97testing.html
9. Burger, D., Austin, T., Bennett, S.: Evaluating future microprocessors: The simplescalar toolset. Tech. Rep. CS-TR-96-1308, University of Wisconsin - Madison, CS Dept. (1996)
10. Burger, D., Austin, T.M., Bennett, S.: Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, Madison (1996)
11. C-Lab: Wcet benchmarks. URL http://www.c-lab.de/home/en/download.html. Available from http://www.c-lab.de/home/en/download.html
12. Chana, S.K., Karale, S.J.: Analysis of Intrusion Detection Response System (IDRS) In Cyber Physical Systems (Cps) Using Regular Expression (Regexp). IOSR Journal of Computer Engineering (IOSR-JCE) (2014). URL http://dx.doi.org/10.6084/m9.figshare.1109874
13. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: Pointguardtm: protecting pointers from buffer overflow vulnerabilities. In: SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium, pp. 7–7 (2003)
14. Crenshaw, T., Gunter, E., Robinson, C., Sha, L., Kumar, P.: The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures. In: IEEE Real-Time Systems Symposium, pp. 400–412 (2007)

15. Dn, G., Sandberg, H., Ekstedt, M., Bjrkman, G.: Challenges in power system information security. Security Privacy, IEEE **10**(4), 62–70 (2012)
16. Gao, W., Morris, T., Reaves, B., Richey, D.: On scada control system command and response injection and intrusion detection. In: eCrime Researchers Summit (eCrime), 2010, pp. 1–9 (2010)
17. Hadeli, H., Schierholz, R., Braendle, M., Tuduce, C.: Leveraging determinism in industrial control systems for advanced anomaly detection and reliable security configuration. In: Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on, pp. 1–8 (2009)
18. Healy, C.A., Arnold, R.D., Mueller, F., Whalley, D., Harmon, M.G.: Bounding pipeline and instruction cache performance. IEEE Transactions on Computers **48**(1), 53–70 (1999)
19. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: CCS '03: Proceedings of the 10th ACM conference on Computer and communications security, pp. 272–280 (2003)
20. Kuperman, B., Brodley, C., Ozdoganoglu, H., Vijaykumar, T., Jalote, A.: Detection and prevention of stack buffer overflow attacks. Commun. ACM **48**(11), 50–56 (2005)
21. Labrosse, J.: Micro C/OS-II. R & D Books (1998)
22. Lauf, A., Peters, R., Robinson, W.: Intelligent intrusion detection: A behavior-based approach. In: 21st Advanced Information Networking and Applications: Symposium for Embedded Computing (2007)
23. Lauf, A.P., Peters, R.A., Robinson, W.H.: A distributed intrusion detection system for resource-constrained devices in ad-hoc networks. Ad Hoc Netw. **8**(3), 253–266 (2010)
24. Levy, E.: Crossover: Online pests plaguing the offline world. IEEE Security and Privacy **1**(6), 71–73 (2003)
25. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. of the Association for Computing Machinery **20**(1), 46–61 (1973)
26. Mitchell, R., Chen, I.: Effect of intrusion detection and response on reliability of cyber physical systems. Reliability, IEEE Transactions on **62**(1), 199–210 (2013)
27. Mitchell, R., Chen, I.R.: A survey of intrusion detection techniques for cyber-physical systems. ACM Comput. Surv. **46**(4), 55:1–55:29 (2014)
28. Mohan, S.: Worst-case execution time analysis of security policies for deeply embedded real-time systems. SIGBED Rev. **5**(1), 1–2 (2008)
29. Mohan, S., Hawkins, F.M.W., Root, M., Whalley, D., Healy, C.: Parametric timing analysis and its application to dynamic voltage scaling. ACM Transactions on Embedded Computing Systems p. (accepted) (2007)
30. Mohan, S., Mueller, F.: Preserving timing anomalies in pipelines of high-end processors. Tech. Rep. TR 2007-13, Dept. of Computer Science, North Carolina State University (2008)
31. Mohan, S., Mueller, F., Hawkins, W., Root, M., Healy, C., Whalley, D.: Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In: IEEE Real-Time Systems Symposium, pp. 233–242 (2005)
32. Mohan, S., Mueller, F., Whalley, D., Healy, C.: Timing analysis for sensor network nodes of the atmega processor family. In: IEEE Real-Time Embedded Technology and Applications Symposium, pp. 405–414 (2005)
33. Moses, A.: 'sinister' integral energy virus outbreak a threat to power grid (2009). URL http://www.smh.com.au/technology/security/sinister-integral-energy-virus-outbreak-a-threat-to-power-grid-20091001-gdrx.html
34. Mueller, F.: Timing analysis for instruction caches. Real-Time Systems **18**(2/3), 209–239 (2000)
35. Ravi, S., Raghunathan, A., Kocher, P., Hattangady, S.: Security in embedded systems: Design challenges. ACM Trans. Embed. Comput. Syst. **3**(3), 461–491 (2004)
36. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: Swatt: Software-based attestation for embedded devices. Security and Privacy, IEEE Symposium on **0**, 272 (2004)
37. Shacham, H., Page, M., Pfaff, B., Goh-Jin, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: CCS '04: Proceedings of the 11th ACM conference on Computer and communications security, pp. 298–307 (2004)

38. Shao, Z., Zhuge, Q., He, Y., Sha, E.H.M.: Defending embedded systems against buffer overflow via hardware/software. In: ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference, p. 352. IEEE Computer Society, Washington, DC, USA (2003)
39. Shin, S., Kwon, T., Jo, G.Y., Park, Y., Rhy, H.: An experimental study of hierarchical intrusion detection for wireless industrial sensor networks. Industrial Informatics, IEEE Transactions on **6**(4), 744–757 (2010)
40. Son, S.H., Mukkamala, R., David, R.: Integrating security and real-time requirements using covert channel capacity. IEEE Transactions on Knowledge and Data Engineering **12**, 865–879 (2000)
41. Venugopalan, R., Ganesan, P., Peddabachagari, P., Dean, A., Mueller, F., Sichitiu, M.: Encryption overhead for sensor networks and embedded systems: Modeling and analysis. In: Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 188–197 (2003)
42. Wegener, J., Mueller, F.: A comparison of static analysis and evolutionary testing for the verification of timing constraints. Real-Time Systems **21**(3), 241–268 (2001)
43. Whitham, J.: Real-time processor architectures for worst case execution time reduction. Ph.D. thesis, University of York (2008)
44. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The worst-case execution time problem — overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems **7**(3), 1–53 (2008)
45. Wu, B., Chen, J., Wu, J., Cardei, M.: A survey of attacks and countermeasures in mobile ad hoc networks. Wireless Network Security **30**(3), 103–135 (2007)
46. Xie, T., Qin, X., Lin, M.: Open issues and challenges in security-aware real-time scheduling for distributed systems. Journal of Information **6**(9) (2006)
47. Zhang, L., White, G.B.: Analysis of payload based application level network anomaly detection. In: HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences, p. 99 (2007)
48. Zhang, Y., Lee, W.: Intrusion detection in wireless ad-hoc networks. In: MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking, pp. 275–283 (2000)
49. Zhang, Y., Lee, W., Huang, Y.A.: Intrusion detection techniques for mobile wireless networks. Wireless Networking **9**(5), 545–556 (2003)
50. Zimmer, C., Bhat, B., Mueller, F., Mohan, S.: Time-based intrusion dectection in cyber-physical systems. In: International Conference on Cyber-Physical Systems, pp. 109–118 (2010)