Time-Based Intrusion Dectection in Cyber-Physical Systems *

Christopher Zimmer, Balasubramanya Bhat, Frank Mueller
North Carolina State University
{cjzimme2,bbhat}@ncsu.edu,mueller@cs.ncsu.edu

Sibin Mohan
University of Illinois at Urbana Champaign
sibin.m@gmail.com

Abstract

Embedded systems, particularly those with temporal constraints known as real-time systems, are increasingly deployed in every day life. Such systems that interact with the physical world are also referred to as cyber-physical systems (CPS). These systems are common in critical infrastructure from transportation to health care. They impact our life and the environment we live in. While security in CPS-based real-time embedded systems has been an afterthought, security aspects are becoming critical as these systems are increasingly networked and exhibit distributed interdependencies. The advancement in their functionality has resulted in more conspicuous interfaces, which can be exploited to attack such systems. Hence, security functionality is becoming a necessary component of embedded real-time design, particularly in the CPS realm.

In this paper, we present a method for time-based intrusion detection. More specifically, we detect the execution of unauthorized instructions in CPS environments with real-time constraints. The functionality in this work is provided through the utilization of values attained from performing worst-case timing analysis. Timing analysis values are readily available as they are determined prior to the schedulability analysis of real-time systems. Using the same tools that provide a macro view of timing within a CPS application, we demonstrate how to provide more focused timing values for specific execution scopes of an application. Utilizing such focused values, the application is enhanced to engage in internal self checks. Internal timing checks are verified against focused timing values to enable the detection of code injection attacks. To the best of our knowledge, such detection of system compromises through micro-timing information is a novel contribution to CPS environments with real-time constraints.

Keywords Real-Time Computing, Security, Timing Analysis

1. Introduction

Embedded systems have permeated into every aspect of day-to-day life. Examples range from non-critical systems, such as televisions or toasters, over moderately critical systems, such as stop lights or other enhancing infrastructure, to highly critical ones, such as anti-lock breaks, hydro-electric dam controls 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 in general. Most such cyber-physical control systems are embedded systems with real-time constraints. As these systems are increasingly used in our daily life, insuring that these devices are secure from intrusion and tampering by adversaries is a design challenge of utmost importance.

While the development of real-time systems for the CPS domain is very stringent, there might be vulnerabilities exposed by libraries or methods that may enable an attacker of the system to execute arbitrary instructions on the target machine, *e.g.*, by injecting malicious code. As more embedded applications, particularly CPS applications, utilize networks these attacks are prone to become prevalent against real-time systems as well.

The design constraints of embedded real-time systems lend themselves well to the development of security methodologies while such techniques would not be directly applicable to general-purpose applications. The primary constraint of interest is the detailed knowledge obtained from timing analysis on CPS applications within real-time systems. Here, analysis is performed to determine timing information about the application, such as worst case execution time (WCET) and best case execution time (BCET). These two timing metrics represent a subset of knowledge common to real-time applications, which lend themselves well to security analysis: As WCET and BCET safely bound the upper and lower execution time of specific code sections, execution times above or below the respective bounds are strong indications for a system compromise.

In this paper, we present a methodology that utilizes instrumentation and analysis from within real-time applications in the attempt to detect the execution of unauthorized code. Using actual timing metrics and comparing them with worst-case measurements allows the programs to detect security breaches due to intrusion within the system as well as situations where an application is going to exceed its timing requirements prior to the actual deadline miss, which provides ample time to transition to a fail-safe state.

2. Timing Analysis

Timing analysis is a strict requirement for hard real-time systems where a missed deadline may render the entire system incorrect. Timing analysis is used to insure that an application's best and worst case times can be bounded. The analysis allows designers to verify if system tasks can meet their deadline.

The purpose of timing analysis in real-time systems is generally to determine the schedulability of a task set, *i.e.*, to ensure that each task meets its deadline. In this context, the overall WCET bound of a task becomes the key metric. Our work heavily relies on WCET

* This work was supported in part by NSF grant EEC-0812121 and U.S. Army Research Office (ARO) grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WXYZ 'XX date, City.

Copyright © 20XX ACM [to be supplied]. . . $5.00

bounds, but for security reasons and not for the determination of schedulability.

To conduct our study, we use our WCET tool chain [3, 6, 5] that enables us to accurately gauge the WCET values of several applications from both the macro view of the application as well as micro ranges of instructions in the code. These analysis tools provide timing data at multiple levels and enable the evaluation of such data for more focused ranges of code. Figure 1 depicts a graphical representation of the tools utilized to perform timing analysis in our experiments. A compiler provides an assembly file of the application in annotated PISA assembly format. This intermediate code along with loop bounds 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. The framework utilizes the annotated assembly and loop bounds to derive safe WCET and BCET bounds.

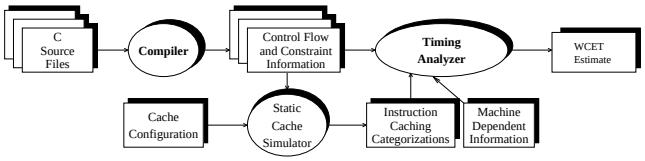

Figure 1. Timing Analysis Tools

Throughout our work, we enhanced the timing analysis toolset in Figure 1 to determine not only the WCET but also the best case execution time (BCET) bounds, and not just for entire tasks but also for micro ranges of code. The original toolset provided timing feedback at the functional and loop level. We enhanced this capability to supply timing feedback for a series of smaller ranges within the same simulation 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 attack code injection.

3. Design

This work puts forth the utilization of timing values readily available in real-time cyber physical systems to establish a intrusion detection technique. By utilizing our technique, critical and potentially vulnerable security-related information can be spread through-out the entire system. The primary goal of this work is to design and assess methodologies that provide real-time CPS applications with an intrusion detection security mechanism.

3.1 Timed Return Path Security (TRPS)

Timed Return Path Security (TRPS) is an application-level instrumentation that utilizes communication through the system clock in order to maintain a series of sanity checks structured throughout the code. To detect code injection attacks, we structure sanity statements mainly around application code that could potentially be overwritten, allowing the attacker to perform malicious actions. Such attempts are most commonly known as buffer overflow attacks. They involve overwriting the return address of a routine whose frames are stored on the stack. When the program executes the return statement of such a function. The control will be transferred to the location indicated by the overwritten return address. Attackers often choose a modified return value pointing into hand-written instructions. Such specialized attack codes may modify global program variables or even spawn new programs given sufficient knowledge of the affected application.

TRPS uses a mechanism to detect such attacks. Mischievous reasons for doing so may range 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 damage.

TRPS creates multiple sanity checks throughout an application at critical points where the program counter could potentially be transferred *via* a pointer to an undesignated area. These checks obtain clock information just before and after the return instruction as seen in steps 1 and 2 of Figure 2. Our method then utilizes the difference between the two time stamps and compares this delta against an already predetermined worst-case execution bound for the respective return path. This is depicted in step 3 of Figure 2. If the dynamically observed delta exceeds the WCET bound, excess instructions must be executed indicating a potential security compromise. In contrast to arbitrary code sections, static timing analysis on these focused regions yields tight WCET bounds since they mainly consist of a single straight-line execution path. Code sections subject to pointer-controlled flow transfers whose pointers are stored on stack generally comprise a series of loads and stores to restore prior processor state and unwind the stack. The communication structure of this method is displayed in Figure 2. It shows the application interfacing with the system twice to obtain values from the system clocks before checking the timestamp delta to validate WCET bounds. **It is important to note that even if these regions exceed the measured WCET, it does not mean the overall program will exceed its calculated WCET. This makes TRPS well suited for detecting attacks that would not result in a deadline miss otherwise.**

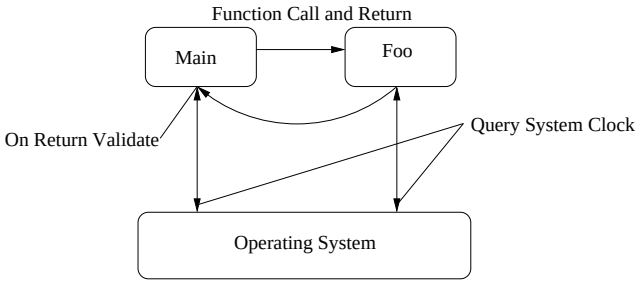

Figure 2. Timed Return Path Security

4. Implementation and Experimentation Framework

The overall framework for experimentation is depicted in Figure 3. We obtained our static WCET analysis tool that provided us with the necessary timing analysis data [3, 6, 5]. The timing analysis tool was configured for a system utilizing the PISA instruction set. The cache configuration for both the static cache simulator and the timing analyzer were configured without data caches but with instruction cache misses accounted for in the WCET analysis. The choice of the cache configurations 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 methods. For this metric, the smaller this threshold, the stronger is the protection.

To facilitate our experiments, we enhanced the timing analyzer with support for checkpointing instructions. These checkpointing instructions allow us to determine the exact cycle time at which a single instruction finishes execution.

We further obtained a customized version of the SimpleScalar processor simulator [1]. This modified version of SimpleScalar

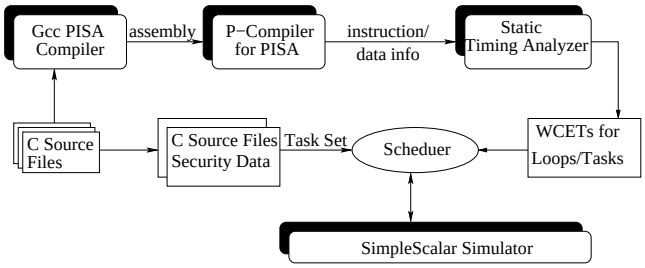

Figure 3. Framework

supports multitasking and has been enhanced to support a scheduler thread / task [4]. The target 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 configured with perfect branch prediction and prefect instruction caches but without data cache support. This matches the configurations of the static analysis tools.

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. Assessing such a trade-off is limited to a concrete implementation platform (see below) in this paper.

The scheduler utilized within 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 applied methods on the scheduler itself. Our implementation modified the scheduler to support relative time for each thread aggregated during preemptions and at security checks of a task to most accurately track the clock period of a particular task.

We further made the following enhancements to the SimpleScalar environment. We implemented two system calls to query timing information. Before a return from a function / method, the first system call is issued. At the destinations of a function / method return, the second system call is triggered. Both calls query the clock, and the difference in time between the two calls is then compared with static timing bounds for the respective code sections.

The motivation for creating two distinct system calls was to create a sequential ordering of these calls. If call one was issued without a corresponding call two (or vice versa), a control-flow violation is detected. Subsequently, a system-defined adverse 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.

We tested our implementation using a set of floating-point and integer benchmarks from the C-Lab benchmark suite [2]. The actual benchmarks used are shown in Table 1.

5. Results

In TRPS, the timed return path verification, utilizes an *absolute* task timer to determine the total time since the simulation start point.

C Benchmark	Function
adpcm	Adaptive Differential Pulse Code Modulation
lms	An LMS adaptive signal enhancement
srt	Bubble Sort
fft	Fast Fourier Transform

Table 1. C-Lab Benchmarks

Timed Return Path Security Results

Figure 4 depicts baseline / modified (TRPS) cycle overheads for WCET benchmarks SRT, LMS, ADPCM and FFT. The overheads, ranging between 0.22% and 18.71%, are often negligible or at most tolerable assuming sufficient slack in a real-time task schedule. Higher overhead in ADPCM is due to its modular structure compared to other benchmarks. It consists of several small functions that are called with a loop. Thus, our TRPS checks are invoked significantly more frequently (at nesting level one) than in other benchmarks (at nesting level zero — not inside of any loops).

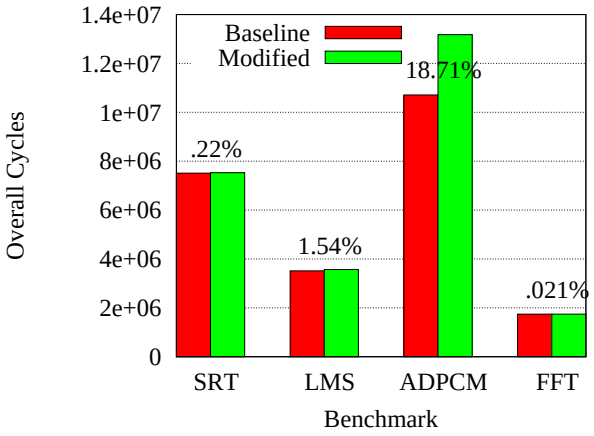

Figure 4. TRPS Overhead

Table 2 shows the sensitivity results of TRPS for various 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 (column 3) and the number of slack cycles that would remain undetected (column 4). This 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 TRPS assesses time on a straight-line path of the control flow. Hence, the window of vulnerability is restricted to a sensitivity of 9-39 cycles. This limits the amount of code that may be injected code without being detected.

These results provide a lower bound, but it can be argued that the upper bound for undetectable injections is larger. First, an attacker could skip over selected instructions on the return path that manipulate registers and stack and instead inject their own code. However, disguising the side effects of polluting stacks and registers may not be trivial depending on the actual code. Conversely, we argue that additional security measurements 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. Nonetheless, early warning indicators could be dynamically triggered to activate stringent security checks that bare higher costs. Alternatively, system functionality could be

reduced in order to limit potential damage to the *physical* side of the CPS application. Overall, the results in Table 2 illustrate that the timing estimations and subsequent security checks for straight-line code are very precise, thus leaving little room for injected code.

Table 2. TRPS WCET and Sensitivity 4KB I-Cache[cycles]

Benchmark	Function	WCET	Sensitivity
SRT	Initialize	35	25
SRT	BubbleSort	45	19
LMS	LMS	28	18
FFT	FFT	25	8
ADPCM	Encode	93	11
ADPCM	Decode	65	39

6. Conclusion

In this work, we developed a novel software methodology that provides enhanced security in deeply embedded real-time systems. We attain elevated security assurance through new levels of instrumentation that enable us to detect anomalies, such as timing dilations exceeding feasible bounds. We utilize tight timing bounds for selected code sections that are readily available at no extra cost whenever static timing analysis is required as part of schedulability analysis of a real-time system. The timing bounds are subsequently utilized to monitor execution during runtime. Upon validation of timing bounds, no action is taken. 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 CPS environments with real-time constraints.

References

[1] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, University of Wisconsin - Madison, CS Dept., July 1996.

[2] C-Lab. Wcet benchmarks. Available from http://www.c-lab.de/home/en/download.html.

[3] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[4] S. Mohan, F. Mueller, W. Hawkins, M. Root, C. Healy, and D. Whalley. Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*, pages 233–242, Dec. 2005.

[5] S. Mohan, F. Mueller, D. Whalley, and C. Healy. Timing analysis for sensor network nodes of the atmega processor family. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 405–414, Mar. 2005.

[6] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.