Time-Based Intrusion Detection in Cyber-Physical Systems

Christopher Zimmer, Balasubramany Bhat, Frank Mueller
Dept. of Computer Science
North Carolina State University, Raleigh, NC 27695-8206
{cjzimme2,bbhat,mueller}@ncsu.edu

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

***Abstract*—Embedded systems, particularly real-time systems with temporal constraints, 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 commonly find use in critical infrastructure from transportation to health care. While security in CPS-based real-time embedded systems has been an afterthought, it is becoming a critical issue as these systems are increasingly networked and inter-dependent. The advancement in their functionality has resulted in more conspicuous interfaces that may be exploited to attack them.**

In this paper, we present three mechanisms for time-based intrusion detection. More specifically, we detect the execution of unauthorized instructions in real-time CPS environments. Such intrusion detection utilizes information obtained by static timing analysis. For real-time CPS systems, timing bounds on code sections are readily available as they are already determined prior to the schedulability analysis. We demonstrate how to provide micro-timings for multiple granularity levels of application code. 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.

I. Introduction

Embedded systems have permeated every aspect of day-to-day life. Examples range from non-critical systems (televisions, toasters), moderately critical systems (stop lights) to highly critical ones (anti-lock brakes, 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. 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, and system restarts often cannot be instant due to unstable physical system state, *e.g.*, during chemical and thermo-dynamic reactions.

While the development of real-time software for CPSs is stringent, vulnerabilities are exposed by libraries and specific embedded domain device software that enables attackers to execute arbitrary code. Such code injection attacks have been common for several years in the general-purpose domain. As more embedded applications, particularly CPS applications, utilize networks they become more susceptible to such attacks.

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).

Fortunately, the design constraints of embedded real-time systems lend themselves well to security methodologies otherwise not applicable to general-purpose applications. The focus of this work is in exploiting detailed timing bounds obtained through static analysis of application code for security. Worst case execution time (WCET) 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.

Our technique is specifically designed for embedded real-time systems where general-purpose domain protection may prove ineffective: Randomization such as Address-space layout randomization [21] and StackGuard [6], designed for 64-bit space, can be defeated more easily in embedded 8/16/32-bit processors with brute-force attacks. Instruction Set Randomization [8] and other hardware enhancements [22], [9] require additional hardware (with limitations due to static buffer constraints) or high-overhead binary rewriting whose cost and overhead are shunned in lower-end embedded systems.

Contributions: We develop three mechanisms that utilize instrumentation and analysis from within real-time applications to detect the execution of unauthorized code and show their effectiveness 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 intrusion within the system as well as situations where an application is going to exceed its timing requirements prior to an actual deadline miss. (1) T-Rex utilizes timing bounds to detect intrusion at a fine-grained level through instrumentation of return paths. Code injections resulting in time dilations as small as 5-22 cycles, depending on system parameters, are discovered. (2) T-ProT validates intra-task checkpoints via synchronous scheduler invocations to uncover coarser-grain injections between 9 and 5k cycles. (3) T-AxT exploits asynchronous scheduler-triggered timing validations of application code sections without requiring instrumentation. These security checks can be strategically scheduled to utilize otherwise idle time in the schedule. The granularity of the schemes not only provides detection capabilities but also sufficient time to transition to a fail-safe state.

II. Attack Model and Scenario

Attacks on embedded systems with or without real-time constraints can materialize in a variety of ways. 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.

Past security work predominantly focused on wireless networks in the domain of embedded systems, such as [28]. 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.

Our adversary model is one where one or more network nodes have been compromised or an attacker has successfully authenticated a node under their 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 [31], [32], [30], 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). Specifically, 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. We have implemented an attack scenario on a MIPS-ISA where a network packet supplies the sensor data from a spoofed or compromised node. The initial input string overruns the bound of the localcpy array to overwrite 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 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).

General-purpose and network-level protection methods are insufficient for such attacks in embedded systems 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

```
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

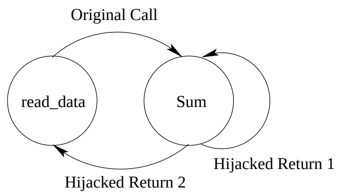

Fig. 2. Diverted Control Flow

input routines beyond POSIX library routines that may have exploits. (2) Statistical detection methods [11] 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 [24].

Our approach, detailed in Section IV, promotes a different approach. Since our focus is on real-time systems with statically analyzable timing bounds 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. The above intrusion was instantly detected with only minimal runtime overhead in the order of 1% of the application's execution time.

This example illustrates just one possible code injection attack that is detected by our approach. 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.

III. Timing Analysis

Accurate knowledge of execution time is a strict requirement for hard real-time systems where a missed deadline may render the entire application incorrect. Timing analysis determines an application's BCET and WCET bound that allows verification if a task's deadline can always be met. Timing analysis can be performed via dynamic [3], [25], static techniques [27], [15] or hybrids of them [2], [14], [26]. 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 compounds the costs of paths throughout the program taking architectural timing effects into account to a safe WCET bound at compile time. In contrast to the dynamic approach, static timing analysis has been shown to provide *safe* WCET bounds [25].

We utilize WCET bounds obtained from static timing analysis in this work. 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.

To conduct our study, we obtained a copy of an existing tool chain [7], [18], [16] depicted in Figure 3 that 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 PISA assembly. 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 that uses the annotated assembly and loop bounds to derive a safe WCET bound.

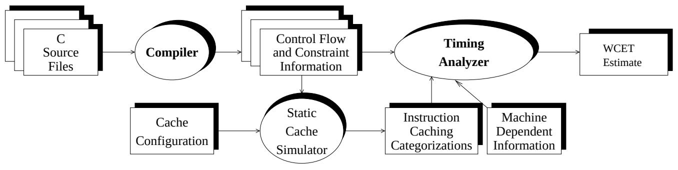
C Source Files
Compiler
Control Flow and Constraint Information
Timing Analyzer
WCET Estimate
Cache Configuration
Static Cache Simulator
Instruction Caching Categorizations
Machine Dependent Information

Fig. 3. Timing Analysis Tools

IV. Design

We have developed a methodology for verifying timing bounds at checkpoints during application task execution. 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 a *macro* and a *micro* check of timing bounds. The former, T-ProT, competes with scheduler-triggered T-AxT checking while the latter, T-Rex, complements the other two schemes. 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 its vulnerability due to tampering.

A. T-Rex: Timed Return Execution

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.

T-Rex uses a pair of checkpoints that compare WCET timing bounds with actually elapsed wall-clock time along a return (from subroutine) path (see Figure 4). 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. T-Rex is equally applicable to arbitrary control transfers, such as function pointers or large conditional switch/case statements resulting in indirect jumps. If the dynamically observed wall-clock delta between checkpoints exceeds the WCET bound then excess instructions were executed indicating 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.

When such a region exceeds the path-based WCET bound, the overall program may not necessarily exceed its overall WCET bound due to shorter paths taken during the remaining of execution. This makes T-Rex 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 WCET bound or its deadline.

T-Rex is built into the operating system as a state machine. It 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. Upon timeout, the corresponding interrupt then indicates a potential system intrusion. A side effect of the state machine is that the checkpoint addresses are checked to insure that they fall within the address range of instructions. 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, using the T-ProT technique described in the following section, such illegal control flow transitions would be detected.

B. T-ProT: Timed Progress Tracking

T-ProT utilizes synchronous calls at security checkpoints to the scheduler and validates WCET bounds of longer code

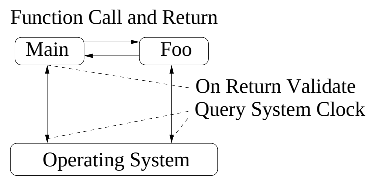

Fig. 4. Timed Return Execution (T-Rex)

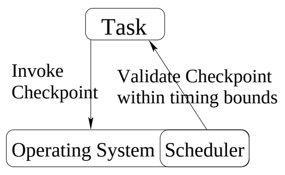

Fig. 5. Timed Progress Tracking (T-ProT)

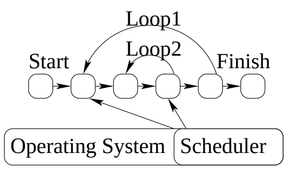

Fig. 6. Timed Address Execution Tracking (T-AxT)

sections than T-Rex (see Figure 5). 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, which can potentially be compromised.

T-ProT detects several intrusion scenarios that divert from the expected control flow, 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. If no checkpoint is encountered hit before this timer elapses, an intrusion is flagged.

Placing checkpoints in control-flow blocks guaranteed to be traversed during execution (*e.g.*, using post-dominator information [1]), we ensure that these checkpoints are always traversed when a job completes or its deadline expires — assuming that the application was not aborted prematurely due to an attack.

Determining the instrumentation points (checkpoints) controls the sensitivity of protection. In some algorithms, the best-case execution time may deviate significantly from the worst-case execution time. For instance, 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 minimizes 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, where the choice of k determines the strength in protection while assuring sufficient slack in the task schedule to accommodate the timing checks via scheduler invocations.

This also meshes well with code obfuscation techniques employing multi-version binaries where we can instrument at disjoint points for otherwise functionally equivalent binaries of the same application to increase the difficulty for attackers to systematically defeat our timed security approach.

C. T-AxT: Timed Address Execution Tracking

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 co-existing 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.

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. Any loops lacking induction variables are supplemented statically during code analysis with an induction variable.

In our experiments, WCET comparison bounds are determined in either absolute or relative time. We utilize WCET bounds *relative* to task activation when multiple execution paths exist. This allows us to eliminate path-aggregate overestimations 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, which tightens bounds and strengthens timed security as a means of intrusion detection.

Two of the timed security techniques rely on application instrumentation. After instrumentation, the overall real-time task set has to be reanalyzed to obtain WCET bounds that 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. The focus of this paper is on time-based intrusion detection while such evasive actions are beyond the scope of this work.

V. Implementation & Experimental Framework

The mechanisms discussed in Section IV were implemented in two different experimental frameworks, one that combines static timing analysis with architectural simulation and another that combines dynamic timing analysis with an embedded system hardware platform.

A. Simulation Framework

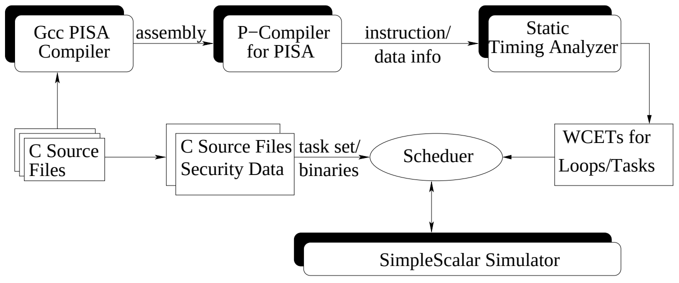

Fig. 7. Framework

The overall experimental framework is depicted in Figure 7. We enhanced a static analysis framework as discussed in Section III 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 [4] enhanced to support multitasking and scheduler threads / tasks, which we exploited to implement earliest deadline first (EDF) scheduling. 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.

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 destinations 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. Call sites are identified by their call stack / PC and frame pointer signature so that calls from injected attacker code are easily identified. We tested our implementation using a set of C-Lab benchmarks [5].

B. Embedded Hardware Framework

Our second set of experiments was conducted by combining dynamic timing analysis with implementations of T-Rex and T-ProT on an actual embedded hardware platform, namely the DSK6713 kit from Spectrum Digital. This board has a Texas Instruments C6 (TMS320C6713) DSP chip running at 150MHz featuring a 32-bit processor with Very Long Instruction Word (VLIW) architecture, 2 levels of caching and up to 256KB of on-chip SRAM programmed under Code Composer Studio v3.1. 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). Software security is deemed critical in power grids as malicious attacks could potentially damage equipment upstream affecting entire suburbs.

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 configured the platform for maximum predictability: (1) Caches are disabled. (2) We utilize SRAM instead of SDRAM to avoid spikes in memory access times during SDRAM self refreshes that last for several microseconds. Bounding refresh overhead is an orthogonal challenge.

Figure 8 depicts our layered system architecture used. We ported a commonly used real-time operating system, MicroC OS II [10], which supports fixed-priority preemptive scheduling. We then implemented a scheduler based on rate-monotonic analysis (RMA) [13] 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. We also provide synchronous application checkpoint calls for implementing T-ProT and monitoring of aggregate execution time per task since with a one microsecond precision.

VI. Results

A. Common Attack Cycles

Timing values of actual attacks for embedded systems are sparse in literature, at best. To determine typical costs, we consider common shell codes used on Linux systems. Metasploit, a repository for such 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

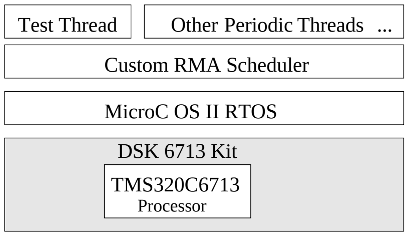

Fig. 8. System Architecture

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

permissions, dramatically increases the cycle overhead. These are examples of common shell code attacks to indicate realistic timings to consider the effectiveness of our methods.

B. Simulation Results

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 to keep an *aggregate* timer for each of the executing job. 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: T-Rex successfully detected the buffer overflow attack depicted in Figure 1 as 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), both without considering caches, while the next two columns show the corresponding results for 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. This limits the amount of code that may be injected without being detected.

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, disguising the side effects of polluting stacks and registers is non-trivial. Overall, the results in Figure 10 illustrate that the timing bounds and subsequent security checks for straight-line code are very precise, thus leaving little room for injected code. Instruction cache effects loosen these bounds proportionally to the cache miss penalty of 10 cycles (as seen for ADPCM).

Timed Progress Tracking (T-ProT) Results: Table 11 assesses the effectiveness of T-ProT, which relies on synchronous scheduler checkpoints to dynamically detect intrusions by WCET bounds violations. The table reports checkpoints between adjacent instrumentation points in the control flow for each application, *e.g.*, 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 12). Corresponding WCET bounds (column 3) and sensitivities (column 4) are reported in cycles for these code sections. We instrumented several checkpoints in benchmarks as illustrated for CNT in Figure 12:

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.

The results of Table 11 indicate that T-ProT has a coarser granularity in that the reported bounds on undetectable injections range up to nearly 5k cycles at the upper end. Hence, scheduler callbacks result in less sensitivity than return path instrumentation. This lower sensitivity is a result of more complex control flow than just straight-line code as in T-Rex. Checkpoints may cross loop levels and are scattered throughout the application. 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. Utilizing instruction caches as depicted in the second half of Table 11 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.

Overall, security is elevated by these scheduler checks. 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 preemption requires a look-up of the previous checkpoint and a comparison of the current timing values with the corresponding WCET bounds. When factored

		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

Fig. 11. T-ProT WCET and Sensitivity cycles

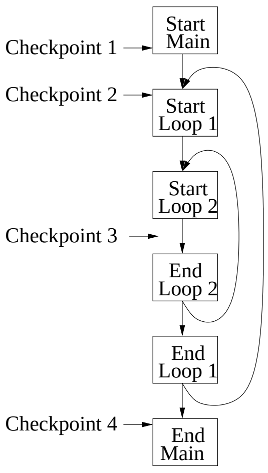

Fig. 12. CNT Control Flow

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

Fig. 13. Timed Address Execution Tracking

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 VIII).

Tab. I 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: T-AxT has the coarsest granularity of our mechanisms. 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 13). 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. Bounds violations are a sufficient indication of intrusion for a given code section.

C. Measurements on an Embedded Hardware Platform

We also implemented T-Rex and T-ProT on the DSP hardware platform discussed in the last section and conducted multiple experiments. 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 14(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 predetermined WCET bounds, 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 (due to intrusion) were detected.

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

Fig. 14. (a) All Asserts Pass (b) Some Asserts Fail

The second experiment consists of calls to a dummy function after obtaining the first clock value but before a return from a function, *i.e.*, we created a code injection scenario. This dummy function simply executes an empty loop (no-op) for 100 iterations before returning to the caller. This simulates code injection 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 14(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 $2.0 + 1.4 > 3.1$.

We next created 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. Since the results resemble those reported in the previous figures, they are omitted here.

We also implemented T-ProT 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 II shows the calculated WCET bounds and observed runtimes for FFT on the embedded TI DSP hardware platform. Without code injection (columns 2-4), all checkpoints pass in this experiment, thus indicating a safe execution.

Tab. II 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

Columns 5-7 of Table II show results for experiments where additional injected code executes between checkpoints 2 and 3. A small loop is introduced between these two checkpoints to simulate code injection. Results of Table II 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 multitasking real-time environments, which makes them universally suitable to CPS applications.

VII. TRADE-OFF: SECURITY VS. TIMELINESS

The objective of providing security in systems in general is to increase the level of protection against attacks 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 II 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 [12]. 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 [17]. Further damage was only contained by reconfiguring servers between the two subsystems to prevent the virus to spread uncontrolled into the power control subsystem. As 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. Hence, the inherent cost of security are well justified in practice.

Furthermore, 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 safety and vulnerability considerations of code sections on one end and availability of slack to meet deadlines on the other end.

More concretely, T-Rex increases the execution time of an application due to its inherent instrumentation. Our results in Section VIII assess this overhead. In many embedded applications, return-path instrumentation results in the invocation of only few checking instances at execution time 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. The balance between such transformations and security overhead of T-Rex to target given slack margins is subject to future work.

T-ProT inflicts overhead through synchronous upcalls and timeout preemptions that activate the scheduler to subsequently check if the application operates within expected timing bounds, where the former overhead is more significant than the latter as it is only triggered upon an intrusion. This method should be used in conjunction with selective placement of checkpoints using strategic and statistical means (*e.g.*, random placement and random activation). Random activations also strengthen security as attacks become more difficult.

T-AxT has easily controlled overhead since it is scheduler activated. 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. While the details of such placements and their trade-offs are beyond the scope of this paper, all methods are designed to allow selective instrumentation subject to future work.

Overall, our security-enhancing methods with 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 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.

VIII. Instrumentation Overhead

We assessed the overall benchmark overheads relative to the performance costs of each of our methods. Table III depicts these overheads in percent relative to the application's base execution time without the security methods. We distinguish the "default overhead" corresponding to the experiments of Section VI and "scaled overhead" with variations on the frequency of intrusion checks.

For T-Rex, the default overheads range from 0.22% to 1.54% for three of the four benchmarks. 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 innermost loop levels for ADPCM, the T-Rex scaled overhead was reduced to just 0.32%, as depicted in the last column of table III. For the remaining benchmarks, default overheads did not justify any inlining so no scaled overheads are reported for T-Rex. The performance impact of T-Rex after occasional code restructuring is low.

The overheads for T-ProT vary depending on the applications instrumentation frequency. The default overhead for the experiments in Section VI 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 III 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 could alternatively also be realized by selective placement.

T-AxT also supports a tunable performance overhead 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 last column of the table shows the scaled overhead of about 8% for a 40,000 cycle instrumentation period.

These results show that overhead scales 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.

Tab. III 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%

IX. Related Work

Much of past work has focused on the evaluation of generic security features in the context of scheduling real-time applications. Often, certain out-of-the-box security mechanisms are applied at the cost of ensuring timeliness while arguing that security is improved [23], [29]. Past work on embedded systems security has focused on sensor networks including remote memory verification, network-related anomaly detection at the packet or application level [20], [31], [32], [30], [28]. Timing analysis is considered in literature as a means to reverse-engineer encryption techniques [19] instead of utilizing it for protection. The emphasis of this work is on utilizing timing analysis bounds to detect code injection attacks.

The most closely related work uses a hardware/software combination to detect attacks [22]. In a first technique, a new stage is added 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. In the second technique, a new "sjmp" instruction XORs 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 [9] 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.

Buffer overflow may be detected in general-purpose systems by placing canaries adjacent to the return address on stack, which may be overwritten in an attack [6]. If a tampered canary is detected prior transferring control at a return, the program aborts itself. Yet, even pseudo-randomized canaries can be exploited in systematic repeated attacks.

Another protection mechanism employed in general-purpose systems is to utilize address-space layout randomization (ASLR) [21]. 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, in a space-constrained embedded real-time system with 8/16/32-bit address spaces, such techniques may be circumvented by repeated attacks [21].

X. Conclusion

We developed three novel software methodologies that provide enhanced security in deeply embedded real-time systems. 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. Such detection of system compromises through micro-timing information is a novel contribution to real-time systems to the best of our knowledge.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2002.

[3] V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. *International Software Quality Week*, May 1997.

[4] 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.

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

[6] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguardtm: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 7–7, 2003.

[7] 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.

[8] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, 2003.

[9] B. Kuperman, C. Brodley, H. Ozdoganoglu, T. Vijaykumar, and A. Jalote. Detection and prevention of stack buffer overflow attacks. *Commun. ACM*, 48(11):50–56, 2005.

[10] J. Labrosse. *Micro C/OS-II*. R & D Books, 1998.

[11] A. Lauf, R. Peters, and W. Robinson. Intelligent intrusion detection: A behavior-based approach. In *21st Advanced Information Networking and Applications: Symposium for Embedded Computing*, 2007.

[12] E. Levy. Crossover: Online pests plaguing the offline world. *IEEE Security and Privacy*, 1(6):71–73, 2003.

[13] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[14] S. Mohan and F. Mueller. Preserving timing anomalies in pipelines of high-end processors. Technical Report TR 2007-13, Dept. of Computer Science, North Carolina State University, 2008.

[15] 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.

[16] 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.

[17] A. Moses. 'sinister' integral energy virus outbreak a threat to power grid, Oct. 2009.

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

[19] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3):461–491, 2004.

[20] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. *Security and Privacy, IEEE Symposium on*, 0:272, 2004.

[21] H. Shacham, M. Page, B. Pfaff, E.-J. Goh-Jin, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[22] Z. Shao, Q. Zhuge, Y. He, and E. H. M. Sha. Defending embedded systems against buffer overflow via hardware/software. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 352, Washington, DC, USA, 2003. IEEE Computer Society.

[23] S. H. Son, R. Mukkamala, and R. David. Integrating security and real-time requirements using covert channel capacity. *IEEE Transactions on Knowledge and Data Engineering*, 12:865–879, 2000.

[24] R. Venugopalan, P. Ganesan, P. Peddabachagari, A. Dean, F. Mueller, and M. Sichitiu. Encryption overhead for sensor networks and embedded systems: Modeling and analysis. In *Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 188–197, Oct. 2003.

[25] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[26] J. Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, University of York, May 2008.

[27] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, Apr. 2008.

[28] B. Wu, J. Chen, J. Wu, and M. Cardei. A survey of attacks and countermeasures in mobile ad hoc networks. *Wireless Network Security*, 30(3):103–135, 2007.

[29] T. Xie, X. Qin, and M. Lin. Open issues and challenges in security-aware real-time scheduling for distributed systems. *Journal of Information*, 6(9), 2006.

[30] L. Zhang and G. B. White. Analysis of payload based application level network anomaly detection. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 99, 2007.

[31] Y. Zhang and W. Lee. Intrusion detection in wireless ad-hoc networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 275–283, 2000.

[32] Y. Zhang, W. Lee, and Y.-A. Huang. Intrusion detection techniques for mobile wireless networks. *Wireless Networking*, 9(5):545–556, 2003.