Worst-Case Execution Time Analysis of Security Policies for Deeply Embedded Real-Time Systems

Sibin Mohan
Dept. of Computer Science, North Carolina State University,
Raleigh, NC 27695-8206. smohan@cs.ncsu.edu

Abstract

Deeply embedded systems often have unique constraints because of their small size and vital roles in critical infrastructure. Problems include limitations on code size, limited access to the actual hardware, etc. *These problems become more critical in real-time systems where security policies must not only work within the above limitations but also ensure that task deadlines are not missed. A critical piece of information for security policies in real-time systems is the worst-case execution time (WCET) of the security code. This paper addresses some of the issues faced in the implementation of such security policies and also the process of determining WCETs for them.*

1. Introduction

Embedded systems are becoming ubiquitous. From cellphones to cars, dishwashers to aircraft, they increasingly permeate our everyday lives. Some embedded systems are not even visible to the everyday user but are all around her. Examples include embedded microprocessors (usually 4 and 8 bit processors) present in the power grid infrastructure, thermostats and water management systems. Such systems are often referred to as "deeply embedded systems" [2], and typically consist of one or more embedded systems connected to an enterprise system or even the Internet. While these are small in size, they could be of vital importance as they might provide access to vital information or a means to shut down critical infrastructure. Even the possibility of unauthorized users being able to monitor these systems, their installation or their usage, might not be desirable. For *E.g.* Shipley *et. al.* [8] obtained access to a modem line that controlled a high-voltage power line. Also, the survivability of such systems is just as important, if not more, than the data or systems that they control/monitor. This is usually more challenging than regular desktops/servers because it is difficult to apply security patches or even have extensive monitoring techniques or administrators present.

Many such embedded systems have timing constraints placed on then, in the form of "deadlines". Such systems are often referred to as *real-time* systems. This introduces additional complexities for the implementation of security policies. Not only must the security policies work correctly at keeping out intruders, but it must also ensure that task deadlines are not missed. Since the security policies must work within the framework of the real-time system, their worst-case execution time (WCET) must be known *a priori*. This could be particularly difficult because the input set (or rather the attack/violation) for the security policy being implemented may not be known in advance. Hence obtaining safe and tight WCET estimates is a non-trivial task.

2. Worst Case Execution Time Estimates

Knowledge of the worst case execution time (WCET) is required for most hard real-time systems. The WCET must be known or safely bounded *a priori*. This knowledge is required so that the feasibility of scheduling task sets in the system may be determined based on the scheduling policy, such as rate-monotone or earliest deadline first scheduling policies [3]. Past work focuses mainly on static timing analysis techniques [1, 5–7, 9]. Of course, one serious problem in performing static timing analysis is the complexity of task code which makes accurate static analysis a difficult task. Dynamic timing analysis techniques, on the other hand, use trace-driven or experimental methods to try to gauge the WCETs. These dynamic techniques have shown to be unsafe [10], mainly due to the extreme difficulty in determining worst-case inputs. This difficulty is more noticeable during the process of determining WCETs of security policies because the actual inputs will not be known until an attack occurs. We also recently introduced a hybrid mechanism [4] which uses hardware/software interactions to accurately gauge WCETs for high-end processors.

Static timing analysis techniques will be most suitable for obtaining WCETs for deeply embedded systems, particularly because such systems use simpler (typically 4 or 8 bit) processors. They have simple pipeline structures and almost never use caches. They typically run their applications/code in a tight control loop. Hence, such systems and the code running on them are more amenable to static WCET analysis. The problem is in the implementation of security policies for such systems. It is difficult to implement a policy which is comprehensive, mainly due to the shortage of resources (processing power/memory/etc.). Also, this policy must not interfere with the timing guarantees of other tasks in the system – *i.e.*, they must not cause deadlines to be missed. Hence, we must be able to determine the WCETs of these security policies so that they can be included in the schedulability analysis. As stated before, static analysis seems to be the way to obtain the WCETs for these policies. These estimates will be pessimistic as we may not be able to statically gauge the input size for the actual attack. There are *three* ways the security policy could be implemented; they are elaborated in the following sections. Each method changes the ways the WCETs are estimated, used and the level of safety/security that can be achieved.

Note: The issue of which security policy is actually used is orthogonal to the ideas discussed in this paper. Any secu-

rity policy that currently exists, or that will be developed in the future, can be used. This paper explains how that particular policy might be analyzed and implemented.

2.1. Included in Task Code

The security policies could be interspersed within the task code. Hence, the determination of the WCET for the security policy would be done when the WCETs of the tasks are determined. This has the advantage that the schedulability analysis for each task will include the overhead for the security and does not require separate analysis. It also has the advantage that since the security policy is integrated within the task code, checking for violations can be done at a very fine level and also very often. Tasks do not have to be interrupted to check for security breaches/problems. Of course, it raises the complexity of WCET analysis and also uses additional system resources by taking significant processor time to perform security checks. At times, such a fine granularity of checking may not be necessary. It is also not flexible as each new security policy will have to be implemented, and WCETs recalculated, for every task in the system.

2.2. Periodic Task for Security Policy

The security policy could be implemented as a separate *periodic* task. This would mean that other tasks in the system need not be aware of the actual security policy being used, which would considerably reduce their complexity. We must perform timing analysis separately for the security policy task, which may not always be a trivial enterprise. There also exists the problem of determining the period for this new task. If it repeats too often (short period) it will take up too much of the processor time, since any effective security policy will have a non-trivial WCET. Coupled with a short period, the effective utilization used up by this task (WCET/Period) will be quite significant. Hence the system designers will have a lower utilization to work with for the other tasks in the system and may have to compromise on functionality. Short periods also increase context switch overheads. On the other hand, if the period is too long, the security task may not always detect the breach/violation as the problem might occur between two instances of the task. There also exists the danger of detecting a breach "after the fact" which might be too late. Of course, if the periods are designed correctly, this method would probably work best and provide for the cleanest implementation.

2.3. Aperiodic Task for Security Policy

The security policy could be a task of its own, as we saw in the previous section, but instead of repeating periodically, it would be an *aperiodic* task. The task would be allowed to run when enough slack has been accrued in the system or when some "controller" (which could be the scheduler) decides that a breach is occurring/has occurred. This has the advantage that the security policy will use as little resources as possible as it will execute only when called upon to do so. Of course, with the case of the scheduler determining the existence of a security violation, the policy is actually split in two – part of it is in the scheduler while the rest is in the aperiodic task. Each time the scheduler is invoked, it can check to see if the security has been breached and invoke the aperiodic security task. With this design we could have different security policies, each one implemented as a separate aperiodic task. Which particular aperiodic task is invoked could depend on the type of violation/breach. Alternately, since a security breach is a serious situation the entire system could switch to a protective or recovery state, or perhaps even shut down. This introduces the challenge of performing timing analysis on the scheduler as well as the aperiodic task(s). The biggest drawback with this approach is that this task is invoked only *after* a security breach has taken place, which might be too late to stop the intruder.

3. Conclusion

We have seen that the field of deeply embedded systems, particularly those with timing constraints, require some form of security and related policies for safeguarding against intruders. It is important to obtain accurate timing estimates for security policies that will execute on such systems to ensure that other tasks do not miss their deadlines. While we are not concerned with what the *actual* security policies might be, we suggest some ways they can be analyzed and integrated with hard real-time systems. We intend to explore these techniques of including security policies into such systems in more detail in future work.

References

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

[2] P. Koopman. Embedded system security. In *IEEE Computer*, 2004.

[3] 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.

[4] S. Mohan and F. Mueller. Hybrid timing analysis of modern processor pipeline via hardware/software interactions. In *submitted to Real-Time Embedded Technology and Applications Symposium*, 2008.

[5] 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.

[6] 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.

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

[8] P. Shipley and S. Garfinkel. An analysis of dial-up modems and their vulnerabilities. Available from http://www.dis.org/filez/Wardial_ShipleyGarfinkel.pdf, 2001.

[9] S. Thesing, J. Souyris, R. Heckmann, F. R. andM. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, June 2003.

[10] 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.