Run-Time Security Evaluation: Can We Afford It?

Cristina Serban*
Bell Laboratories
Lucent Technologies
Murray Hill, NJ 07974
cristina@lucent.com

and

Bruce McMillin†
Dept. of Computer Science
University of Missouri-Rolla
Rolla, MO 65401
ff@cs.umr.edu

Abstract

The use of the run-time security evaluation (RTSE) method for a distributed application takes a toll in overall application performance. The associated overhead and its major sources are discussed, along with possible solutions for improvements, and questions that remain still open.

1 Introduction

Earlier this year at Oakland we introduced the concept of run-time security evaluation (RTSE) for distributed applications[6], as a necessary addition - and not a replacement - for verification, as it is traditionally performed at development time.

The goal of the present paper is to discuss the feasibility and costs of using RTSE for real-life applications, in order to see what can and should be done to bring it to the easy-to-use realm.

While the RTSE method itself is well-based and mature from the theoretical point of view and it has been successfully used already for our prototype, its application involves costs that cannot be ignored and must be therefore taken into account and minimized, if feasible.

---

* The work presented in this paper was performed while this author was with the University of Missouri - Rolla.

† This work was supported in part by the National Science Foundation under Grant Number MSS-9216479 and, in part, from the Air Force Office of Scientific Research under contract number F49620-92-J-0546 and F49620-93-1-0409.

1996 ACM New Security Paradigm Workshop Lake Arrowhead CA

2 Overview of RTSE

The new paradigm in RTSE does not concern the security model to be used or the type of policy to be enforced, but gives a new response to the question "***When*** to evaluate whether the security requirements given for an application are fulfilled?". The answer proposed is "***At run-time***" and along with it a method is introduced for evaluating if a given distributed application complies to the security requirements given for it, while the application executes.

The need for run-time evaluation of security comes from the likelihood of occurrence of abnormal run-time conditions that may appear in the underlying system (violating the assumptions about its behavior, as they were made for verification purposes at development time). Checking at run-time ensures that run-time conditions can be handled: RTSE covers hardware and software faults as well, and defines a security violation in the most general sense as being any behavior that is not in accord with the given security specifications, regardless of the source of this behavior.

The problem to be solved is to evaluate for a given application and the corresponding security specifications if the current execution of the application complies to the security specifications, or if any specification is not fulfilled, no matter the underlying cause, signal a security violation.

We introduced the RTSE method to solve this problem, and we achieved both a theoretical and a practical set of results (presented with all due details in [6]). The method is shown to have a sound foundation from the theoretical point of view, and the prototype built using RTSE proves its feasibility for practical applications.

The application is assumed to be composed of distributed, independent processes, in a share-nothing configuration in which information is exchanged only by message passing. This type of environment is mod-

eled using CSP [2], and the tool used for actual programming is CCSP [4], an in-house developed tool that offers CSP-like syntax on top of a C environment and also a run-time support set of mechanisms.

In the rest of this section an overview of RTSE is given in order to provide a basis for later discussions.

The RTSE method takes a distributed application, the security specifications given for it, plus the current execution of the application, and checks whether the execution in progress complies to the given specifications or if at a certain moment this is not the case, a security violation is detected at that moment.

The steps involved in the RTSE method are:

1. Specify the security properties of the targeted computing application using formal logic.

2. Collect run-time event/trace histories, the causal structure of the execution, and

3. Evaluate security properties of the application on the event/trace histories.

At step 1, for security requirements evaluation, we use an algebraic approach, in which the security specifications are expressed as a set of executable assertions that can be evaluated on application traces using set operations. In general, assertions express boolean invariants; executable assertions are assertions that can be embedded into code and evaluated at run-time. For security evaluation, the executable security assertions are obtained at this step from the security specifications, then the assertions are embedded into the application's code and their evaluation takes place at run-time, using step 3 of this method.

At step 2, we have developed algorithms to obtain a run-time history and to build the causal structure of the execution. The run-time history of a distributed application is obtained by collecting and partially ordering events occurring in the application.

Each process collects the events it observes into its own-history, updating it with new events as soon as they become known. The process also records the events it observes into out-histories (collections of events observed since previous communication with the respective process the out-history is maintained for), one for each process with which it communicates. At communication time, the process exchanges its respective out-history with the process it communicates with, permitting in this manner the events to diffuse within the system.

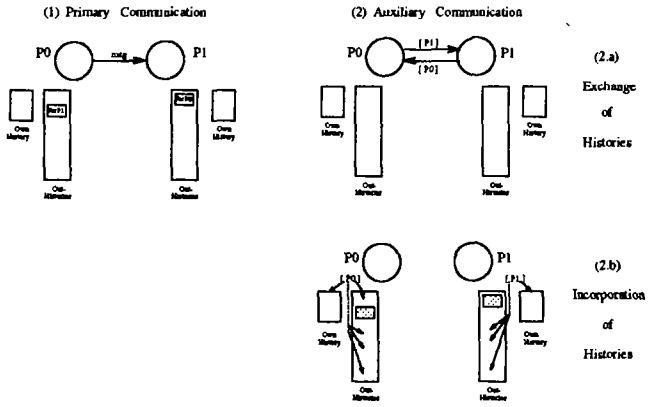

Figure 1. The Augmented Communication

Each message-passing communication between two processes is expanded to a 2-phase operation called **augmented communication** (presented in Figure 1). The phases of an augmented communication are:

1. Primary Communication: The sender process P0 sends message *msg* to the receiver process P1, according to functional requirements.

2. Auxiliary Communication:

   (a) Exchange of Histories: P0 sends to P1 the out-history P0 collected for P1, containing all the events observed by P0 since the last communication with P1. Similarly, P1 sends to P0 its out-history for P0.

   (b) Incorporation of Histories: Upon receiving the latest updates from P1, P0 incorporates these events into its own-history, and also into the out-histories. The out-histories for the partners in communication are cleared. As of this moment P0 and P1 have the same image of the application - and this is also a synchronization moment for their vector clocks.

While a process incorporates each event from a received history into its own-history and into the out-histories, it performs a **consistency check**: if an event from the received history has the same timestamp as an event from the own-history (or an out-history) and both events occurred at the same process, then the event information must be the same for both events (up to differences given by sanitization procedures that might have modified parts of the event information for the two events on their respective arriving paths).

If the consistency check fails, an inconsistency is detected, signaling a problem (i.e., a faulty process or a security violation) in the application.

Also during the incorporation of received histories into out-histories at a given process, beside the consistency checks one more type of checking is necessary. If the process incorporates all the events it received into all its out-histories (to be sent to all processes it communicates with), an unrestricted flow of information results, compromising any confidentiality requirements that exist for the application.

Therefore, before incorporating an event into an out-history, the process must check whether any constraint exists, according to the security specifications, on disseminating that particular event to the process for which the out-history is built. Such constraints are expressed in RTSE by a set of **dissemination restrictions** we derive from the security specifications.

A dissemination restriction indicates if an event can be incorporated into an out-history without any constraint, or it may not be incorporated at all, or it may be incorporated only after it is sanitized. The specific **sanitization procedures** are also derived from the security specifications.

At step 3, we apply the set of executable assertions to verify whether the collected event history satisfies the security specifications of the application as formalized at step 1. Since an event history is a sequence of events occurring within the application, it represents a process' observation of all the processes during execution. This history can be utilized to do evaluation of assertions at run-time. The evaluation is a simple matter, then, to break down the assertions into predicate calculus expressions quantified over this history sequence. If the run-time behavior violates its specifications, then appropriate actions should be taken.

During the operational evaluation step in RTSE the assertions are evaluated using the data provided by the histories of events in a **distributed checking**, in which each process checks each assertion on each event in its own-history to ensure the checking is complete and meaningful.

3 The Area of Concern: Overhead at Run-Time

To check the validity of our method for a real application, we built a prototype - the Boots System - based on a model problem introduced by Colin O'Halloran[5]. This prototype is a distributed, transactional application that controls the movement of footwear by sending *orders* from *HeadQuarter* to specialized processes, under given security constraints. The Boots System was implemented as a set of independent processes which communicate by message passing to exchange information.

No. of Orders	Time					
	Prim Comm	Aux Comm			Asser tions	Total
		Xch	Ops	Total		
20	20	10	0	10	0	30
40	53	25	1	26	1	82
60	81	40	2	42	2	125
80	110	55	4	59	3	172
100	148	71	6	77	4	229
120	183	87	9	96	6	385
140	222	103	12	115	8	345
160	274	119	15	134	11	419
180	336	135	20	155	14	505
200	384	152	24	176	17	583

Table 1. Timing Results for Boots System

The events in the Boots System are modeled as messages, and the histories contain the messages exchanged by processes, along with the names of sending and receiving processes and the timestamp.

Initially, in the first version of the Boots System, the performance of the application when using assertions and histories of events for RTSE was severely degraded by comparison with the Boots System with bare functionality. This was due to the unbounded growth in size of event histories when all the events were collected and kept since the beginning of the execution.

Clipping the old events from histories was therefore necessary, and we adopted the gossip technique[1], in which events about which all processes know can be discarded from histories, as being too old[1]. This solved the problem of overflowing histories and increasing times for maintaining them, but the overhead due to event histories and assertions is still important.

The timing results are presented in Table 1, and they are composed of times for primary communication (which represent the bare functionality of the Boots System), plus times for auxiliary communication (exchange of histories and operations on histories) and evaluation of assertions (which constitute the part required by RTSE for security evaluation).

The overhead measured in execution times for the Boots System reaches values as high as 40% - a value that might be still acceptable for "almost batch" appli-

[1] In the gossip algorithm, an event is considered too old when its timestamp is less than all local values for vector clocks.

cations (alas, these are really few nowadays), but not too much so for applications with tight time constraints for response and overall performance.

The factors we identified as responsible for overhead are the following:

1. Extra communication for exchange of histories:

   When two processes communicate, the primary communication is required by functionality and the sender's out-history for the receiving process and its vector clock are piggybacked to the actual message, so the only extra time is due to a longer message to be sent. From the other direction though, for the receiver to send its out-history and vector clock to the sending process, an extra communication is needed, which was not required by functionality - and this represents a significant increase in communication time.

2. Less-than-optimum communication:

   Our tool CCSP was very useful in developing the prototype, providing mechanisms for collecting and updating histories of events, and evaluating security assertions on these histories. However the tool itself was not built having the application's performance as a goal, and so communication for instance is not optimized - which explains part of the overhead we obtain for execution.

3. Less-than-perfect methods for performance evaluation in distributed systems:

   Measuring the time necessary for several independent processes to complete a given global task is a difficult operation, due to the lack of a global clock and view, and also to the different speeds actions can be performed in different processes leading to potentially different interleavings of actions at each run of the system. There is no perfect way to monitor a distributed application's performance, unless specialized hardware[3], software, or a combination of both[7] is available beside the application itself.

4. Complexity and quantity of operations to perform for event histories and evaluation of security assertions:

   The creation and maintenance of event histories involve a significant quantity of operations (collection of directly observed events into own-history and out-histories, exchange of out-histories between communicating processes, incorporation of newly received events into own-history and out-histories, consistency checks).

   The executable assertions in turn may have complex forms, and the application may require many security assertions to be checked on the event histories, as a direct consequence of the security requirements given for the application.

4 Possible Solutions

Considering the major sources of overhead for our prototype, as presented above, we noticed that we have solutions for some, but we do not for others.

We do not have a better solution for the first problem, the extra communication for exchange of histories: this communication is the only way to send out required information (out-history and vector clock) in response to an unsolicited message.

We can (and actually plan to, as part of future research) optimize CCSP and have it use a minimum amount of time for communication between processes, such that the total communication time might be reduced.

For the third problem, the timing results obtained for the Boots System depend on the method used to take time measurements. The Boots System is a transaction-type system, in which an *order* originating from *HeadQuarter* generates a multitude of actions throughout the system. In such a setting, timing measurements taken by adding up the times of all processes are not an adequate basis for comparison, as actions occurring at processes in the "middle" of the system can be interleaved arbitrarily and do not have an useful meaning for comparison. A better approach is to measure the time at the "boundaries" of the system, either at the beginning process (*HeadQuarter* in this case, which is the source of *orders* for the whole Boots System), or at the ending process (*Archive* in this case, as it is the final point information from *orders* reaches).

Each choice has its problems though. When taking times at *HeadQuarter*, the number of *orders* emitted is known, but for different runs the *orders* still being processed may not have reached the same point in their processing. For the other end, when timing at *Archive* the number of *orders* completely processed (i.e., archived) is known, but there is no way to tell exactly how many other *orders* are being processed within the system, but have not reached *Archive* yet.

We opted for *HeadQuarter* as the process at which to take timing measurements, counting the number of *orders* emitted, under assurance that a fair, comparable amount of processing for these *orders* is performed downstream, with no points of accumulation or bottlenecks that would bias a comparison in favor of a system

with a fast-emitting source, but with no or little processing further on.

The last source of overhead mentioned above - complexity and quantity of operations involved in RTSE - is actually the point where discussions in the workshop would be most useful, as we do not have up to this moment a good way to reduce this complexity or the amount of brute force work needed for run-time evaluation.

5 Conclusion

The conclusion is that RTSE as a method for security evaluation comes at a cost that may prove to be unacceptably high for certain applications or reasonably high for others, and this brings the question: Can we afford it for real-life applications?

Nevertheless, there are two points that counterbalance this concern.

First, the benefits of using RTSE are real - for instance for an application for which formal verification was not performed, run-time evaluation appears as a must to insure the security requirements are met - and these benefits may offset the actual costs of using RTSE. Even when verification at development time was done and was successful, run-time security evaluation brings added assurance that the application behaves within the boundaries imposed by its security specifications, or if not - a violation is detected.

Second, improvements can and should be made for RTSE to be less costly and easier to use for real implementations. Such improvements constitute the object of further study. Hopefully, discussions on RTSE and the related overhead will suggest new solutions for the current problems.

6 Post-Workshop Thoughts

The comments and discussions during (and even after) the workshop presentation raised a few points of interest:

- The method is useful and it should be used - in the proposed form, or in a simpler version - wherever applicable, as the potential benefits are significant.
- Optimizations are feasible, using software engineering techniques, to reduce the amount of overhead.
- How acceptable or unacceptable is the degradation in performance induced by RTSE on the application is in final analysis a matter of point of view:
  - In normal business and industry settings, any security-related overhead of more than 5% is hard to be accepted, and anything over 10% is out of question.
  - In highly sensitive environments, if a significant increase in assurance is feasible, the related overhead is acceptable to be in the range of a whole order of magnitude.

References

[1] A. Bernstein and P. Lewis. *Concurrency in Programming and Database Systems.* Jones and Bartlett Publishers, 1993.

[2] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, UK, 1985.

[3] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance monitoring: Methods, tools, and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.

[4] B. McMillin and E. Arrowsmith. CCSP - a formal system for distributed program debugging. In *Proceedings of the Software for Multiprocessors and Supercomputers, Theory, Practice, Experience*, pages 260–269, Moscow, Russia, Sept. 1994.

[5] C. O'Halloran. On requirements and security in a CCIS. In *Proceedings of the Computer Security Foundations Workshop V*, pages 121–134, Franconia, June 1992.

[6] C. Serban and B. McMillin. Run-time security evaluation (RTSE) for distributed applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 222–232, Oakland, CA, May 1996.

[7] J. Tsai, K. Fang, and H. Chen. A noninvasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, Mar. 1990.