Chapter 19

ON-LINE INTRUSION PROTECTION BY DETECTING ATTACKS WITH DIVERSITY

J. Reynolds, J. Just, E. Lawson, L. Clough and R. Maglich

Abstract We have built a system for protecting web servers to securely connected, known users that includes an innovative use of diversity for on-line attack identification. We are able to use attack identification to immediately protect the system without debilitating waits for anti-virus updates or software patches by positively verifying attacks with a sandbox. Unique to our approach is the use of diverse process pairs not only for isolation benefits but also for detection. The architecture uses the comparison of outputs from diverse applications to provide a significant and novel intrusion detection capability. With this technique, we gain the benefits of n-version programming without its controversial disadvantages. Diversity of applications also contributes to the isolation of intrusions by software, which is further improved by random rejuvenation.

Keywords: Computer security, fault tolerance, intrusion detection, diversity

1. Introduction

A potential solution to the problem of building more secure but still affordable and timely systems is to combine Commercial-Off-The-Shelf (COTS) hardware and software with proven techniques from the fault tolerant community. COTS software and hardware can provide cheap (though unreliable) components to build information systems. Fault tolerant techniques can build reliable systems from unreliable components despite intermittent or transient faults. In fact, highly available systems have been built with this approach [4]. There have been many other explorations of fault-tolerant approaches to providing reliable systems based on COTS hardware and software [1,9,12,15].

Most fault tolerant techniques work against faults that can be treated as rare events occurring at random. The faults that pertain specifically to computer and network security have different characteristics. These "faults" depend on what are usually called vulnerabilities. Vulnerabilities are most often design

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35697-6_26

E. Gudes et al. (eds.), *Research Directions in Data and Applications Security*

or programming mistakes, which cause software components to exhibit unintended behavior when presented with data or circumstances not foreseen by the developer. They can be exploited by an attacker to obtain privileges or to inject additional errors into the system or to deny service to the system's legitimate users. An attacker can explore a series of potential vulnerabilities until successful, and when successful, he can repeat the successful exploit against the same system or other systems with the same vulnerability. Therefore, computer and network attacks, viewed as faults, cannot be treated as either rare or random.

In systems with redundant software for fault-tolerance but no diversity to avoid common vulnerabilities, these security-related faults are more likely than non-security related faults to propagate from one machine to another as well as repeat in time. This implies a significant additional value for fault diagnosis, including machine learning techniques, and system adaptation for intrusion prevention. Early in the development of fault tolerant computing, considerable attention was paid to fault diagnosis [7,13]. With the recognition that most software faults were Heisenbugs [8] and the importance of failfast semantics to minimize mean time to repair, fault diagnosis became a luxury.

To combat the long-lived, persistent, and malicious characteristics of computer and network attacks, additional innovative techniques are required. One area of research, which has been applied to this domain by others, is the study of Byzantine faults [6]. Byzantine faults are arbitrary faults, that is, faults not defined by the fault model of the system. As such, they represent a large, indeed, infinite class of faults. A fundamental result from the study of Byzantine faults is that fault tolerance for n Byzantine faults requires 3n+1 redundant components [11]. Consequently, implementations of the algorithms for Byzantine fault tolerance may quickly become impractical, not only because of cost but also because of system complexity and size, thus reducing their utility in the development of secure, affordable systems.

Our approach is to augment standard fault tolerant techniques with adaptation and design diversity. Repeatable errors are prevented by an out-of-band control system, which blocks newly identified attacks. Where feasible, the system employs COTS-supplied design diversity (different operating systems and server applications like web servers or Data Base Management Systems). These measures are combined with fault tolerant techniques (including failure detection, failfast semantics, and redundancy) to tolerate intrusions.

This paper describes the system called HACQIT ("hack-it")1. In the remainder of this section, we describe our assumptions and the system hardware and software architecture. Section 2 describes attack identification. Section 3 describes how diversity can be used for detection. Section 4 describes isolation mechanisms. Section 5 contains our plans and the conclusions supported by our work so far.

1.1 Assumptions

HACQIT is not designed to protect general-purpose servers connected to the Internet. Anonymous users are not allowed. All connections to the system are through a Virtual Private Network (VPN), and therefore during any period of operation the number of users is bounded. We assume that configuration or setup of the system has been done correctly, which includes the patching of all known vulnerabilities. We assume the Local Area Network (LAN) is reliable, cannot be flooded, and is the only means of communications between users and the system.

An attacker can be any agent other than end users or HACQIT system administrators, all of whom are trusted. Attackers are assumed to be well-financed and resourceful cyber-terrorists, perhaps employed by nation-state intelligence agencies. We assume they may have prior access to HACQIT design documentation. While such groups could employ or subvert insiders, insiders are assumed to be handled by other means than HACQIT. Attackers do not have physical access to HACQIT cluster hardware. Attacks entail communication via LAN either with software already resident on a host or new software downloaded by the attacker to that host.

Our project focus is tolerating software failure. Software failures are either repeatable or not. Non-repeatable failures may be caused by intermittent or transient (soft) faults (perhaps indicating Heisenbugs). The causes of repeatable failures would include attacks (maliciously devised inputs) that exploit the same vulnerability (bug) in one of our software components. However, we do not presume to be able to divine intent, e.g., malice, so all inputs that cause repeated failures are treated the same: they are blocked. On the other hand, we recognize that the system may fail intermittently from certain inputs, in which case we allow retry of those inputs.

1.2 Hardware and Software Architecture

The focus of this paper is the software on the HACQIT "Cluster," shown in the lower half of Figure 1 (below the "Corporate LAN"). A HACQIT cluster consists of at least five computers: a gateway computer running a commercial firewall and additional infrastructure for failover and attacker blocking; two or more servers of critical applications, only one of which at any time is the primary server and, together with a designated backup, forms a "process pair;" an Out-Of-Band (OOB) machine for overall control and monitoring; and a "sandbox" for attack identification (not shown). The machines in the cluster are connected by two separate LANs.

HACQIT uses primary and backup systems, but they are unlike ordinary primary and backup systems for fault tolerance. The two systems are isolated by the OOB computer, so no propagation of faults, for example, by a worm or

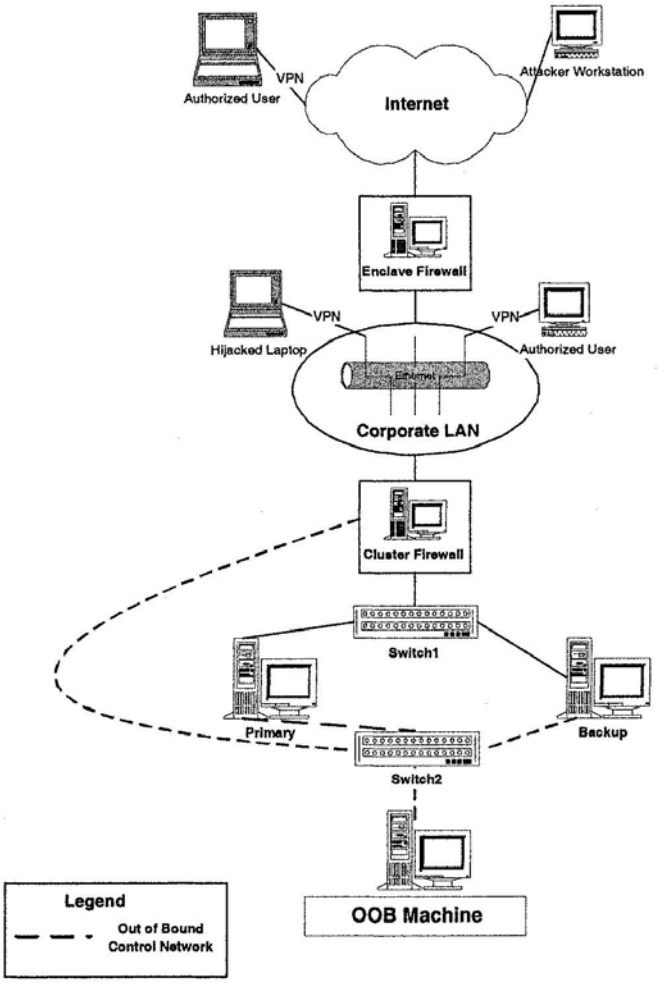

Figure 1. Intrusion tolerant hardware architecture.

an email virus, directly from the primary to the backup, is possible. (The hosts are connected by switches not hubs, and the switches are configured so that no direct communication is possible between the primary and backup.) The potential for propagation from the primary to the OOB machine is limited by sharply constraining and monitoring the services and protocols by which the OOB communicates with the primary. Failover is entirely controlled by the Mediator/Adapter/Controller (MAC) on the OOB computer. When failure is detected on the primary (possibly caused by intrusion), continued service to the end user is provided by promoting the backup to be the new primary.

The software architecture is shown in Figure 2. The large, lower box contains the software components running on the OOB machine. The large, upper

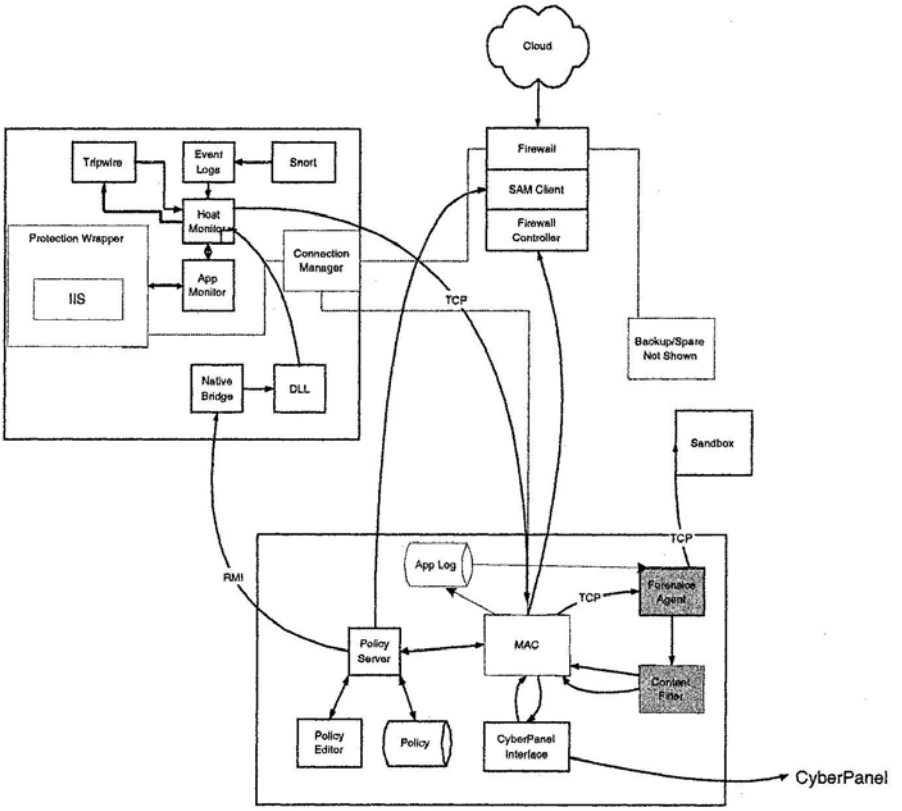

Figure 2. Intrusion tolerant software architecture.

left box contains software components running on each of the servers of critical applications, including the primary and backup. The system administrator controls the cluster through the Policy Editor, which runs on the OOB machine. Policy parameters determine normal and contingent behavior and responses to error conditions for all the components. Once policy has been set, it is disseminated to all components by the Policy Server on the OOB machine. Overall monitoring and control responsibility resides with the MAC.

We will discuss in detail the Forensics Agent, Sandbox, and Content Filter in Section 2. Other software components include the following:

- Web Server Protective Wrapper: This wrapper [3] monitors calls to Dynamic Link Libraries (DLLs) for file access, process execution, memory protection changes, and other potentially malicious functions. When it detects a violation of specified behavior, it will alert, disallow or modify the call depending on policy.

- Application Monitor: This software implements specification based behavior monitoring of the critical application in conjunction with the wrapper described above. It starts and stops its application according to policy.
- Host Monitor (HM): The HM communicates with the MAC and sends either heartbeats or alerts. Alerts are divided into two categories, either severe enough to cause failover or less severe, which requires further analysis by the MAC. It receives policies from the Policy Server through the Native Bridge and DLL module. The HM contains a rudimentary recovery capability that enables it to restore a failed primary to a healthy backup.
- Firewall Controller: The Firewall Controller runs on the HACQIT cluster gateway machine, which hosts the firewall to the cluster. It sends requests/replies through the switch to the current primary and uses the Suspicious Activity Monitor (SAM) Client to block and add users.

2. On-Line Attack Identification

HACQIT's on-line attack identification software provides a qualitative improvement over standard attack prevention. Currently, for most systems, two inadequate activities are performed for attack prevention. First, new attack signatures are downloaded to prevent newly discovered attacks (worms, viruses, etc.). This takes at least a few hours for humans to analyze the attacks and craft rules for the attack prevention software (anti-virus programs, firewalls, etc.) Second, patches for newly discovered vulnerabilities are downloaded and applied to critical software. This takes at least a few days for the vendor to identify the vulnerability, correct the software, and run extensive regression testing; it may also require a reboot to install the patch. Our approach provides immediate protection by automatically preventing repeatable attacks and their variants without software updates.

We use on-line testing to positively identify attacks, eliminating false positives, and rule-based similarity reasoning to use a single attack to recognize a subset of attacks that exploit the same vulnerability. When a failover occurs, the MAC asks the Forensics Agent (FA) to start attack diagnosis. The FA analyzes the "App Log" containing recent requests to determine which request(s) may have caused the failure. It then tests each suspicious request by sending it against a "Sandbox." The Sandbox is an exact duplicate of the machine and application that failed. If the suspicious request causes the same conditions on the Sandbox that resulted in failover of the primary or backup, then it is identified as a "Bad Request" to the MAC.

The MAC puts "Bad Requests" on a list by calling the function UpdateBadReq in the Content Filter. The Content Filter uses the list to tell the MAC

if a future request should be blocked. The FA learns attacks from failures, while the Content Filter generalizes bad requests identified by the FA, so that simple variants are also blocked. In this way, previously unknown attacks (and yet unseen variants) are automatically and immediately prevented from repeatedly causing failover.

Every time a request is received on the primary, it is forwarded to the MAC. The MAC calls AllowRequest in the Content Filter with the new request as the parameter. If there is an exact match with a "Bad Request," it returns false to the MAC, meaning block the request. If an exact match is not found, the Content Filter uses the components of the requests (both new and bad) to determine if the new request is "similar" to a bad request. If it is similar, AllowRequest also returns false; otherwise, it returns true. In this way, learning is generalized from specific requests that have been identified as bad.

One additional aspect of the design of the Content Filter software is worth discussing. The UpdateBadReq function uses AllowRequest in a very clever way. It first calls AllowRequest with the bad request received from the MAC. If AllowRequest returns true, that means the bad request is not on the bad request list, so it is added. If AllowRequest returns false, it means the request is on the bad request list, so it is not added to the list. This prevents duplication. Not only will duplicates be prevented, but also trivial variants will not extend the bad request list to a performance-crippling length.

3. Detection by Comparison of Outputs from Diverse Software

Although we detect intrusions through various sensors and begin attack diagnosis for many different cases, we use diversity to provide a unique and powerful discriminator. Duplicate and compare is an old and effective strategy for hardware error detection [14]. The same input is sent to identical components and the outputs from the components are compared. If the outputs are different, an error has been detected. If the input is retried and the outputs are the same, the failure was transient; otherwise, the components must be repaired or replaced.

Typically, the duplicated components are within one single device, such as a CPU, which, in the case of permanent (hard) failure, is the unit repaired and replaced. If this is not the case, the module or system consisting of the paired components, must stop. (Following Gray and Reuter [8], we call this behavior failfast, because the delay between detecting the error and stopping must happen as quickly as possible to prevent fault propagation.) If a system built on paired components simply stops, then the redundancy has supported error detection but not failure masking, which is necessary to provide continued service despite failures.

Table 1. HTTP response codes.

Code	Category	Meaning
Status codes 100 - 199	*Informational*	Request received, continuing process
Status codes 200 - 299	*Success*	Action was received, understood and accepted
Status codes 300 - 399	*Redirection*	Further action must be taken in order to complete the request
Status codes 400 - 499	*Client Error*	The request contains bad syntax or cannot be fulfilled
Status codes 500 - 599	*Server Error*	The server failed to fulfill an apparently valid request

HACQIT combines redundancy and diversity to both detect errors and isolate failures for software components. This is a unique innovation for software fault (intrusion) tolerance. HACQIT uses two software components with identical functional specifications but different implementations. It can be seen as combining the benefits of n-version programming [2] without its controversial disadvantages, namely higher cost overall and lower quality components (versions) [5].

We can do this with web server and other well known, TCP/IP based applications like mail servers, because these applications must implement a single specification, and there are already many implementations of them. The HACQIT version discussed in this paper uses two web servers, Microsoft's Internet Information Service (IIS) and the open source Apache web server, to implement this design approach.

Every HTTP request to the primary is intercepted and sent to the MAC. After checking that the request is allowed (see Section 2), the request is sent to both web servers, one on the primary and one on the backup, one IIS and one Apache. The HTTP specification defines the status codes with which a web server must respond. The MAC compares the responses (analogous to the outputs from duplicate hardware). If they are the same, the result (by itself) indicates that no failure (intrusion) has occurred. If the status codes are different, the result indicates that a failure may have occurred.

In the latter case, failfast semantics demands failover as quickly as possible, especially because the danger from intrusion propagation is so much greater than the danger from transient fault propagation. The question, then, is how to determine which web server was successfully attacked. An analysis of HTTP status codes makes this possible [10].

Status codes belong to five general categories, as shown in Table 1.

There are only ten different combinations of different status code combinations. Most are not observed or simply not indicative of an error. One mean-

ingful combination that can occur is 200/400. This means one web server responded with success while the other responded with client error. In this case, the cause was a malformed request or a request for a resource that does not exist on one machine. Either cause would indicate that the machine responding with success should be taken off-line (stopped). For example, this is the case when the well-known attack, Code Red, is sent in a request to IIS and Apache: IIS is taken over by Code Red but responds with a success code (200) while Apache is unharmed and responds with an client error code (400).

The second meaningful combination is 200/300 (success/redirection). This indicates one web server (the one responding with success) sent back different content than the other, which responded with redirection, because the client already had the content requested. In this case, the cause is assumed to be that content has changed on the first web server indicating a defacement may have occurred. That server would be taken off-line.

The third significant combination is any code returned by one web server combined with no response (a timeout) from the other. Obviously, the non-responding web server would be treated as having failed. Other combinations of different status codes are ignored.

Different status codes in general are rare, including the three combinations analyzed in the preceding paragraphs. If those combinations occur but are not caused by an intrusion, service will continue. For example, it is possible a 400 code may be combined with a success code because one web server correctly implements the HTTP specification, but the other does not. In that case, a failover would occur once, service to the user would continue, and future repetitions of the incorrectly formed request would be prevented. We believe this can occur only infrequently, because of the maturity of the specification and the implementations.

Besides comparing status codes, failures can be detected by other sensors in the system. For example, the HM monitors system health rules, which include detections of Quality of Service (QoS) violations to the HM, e.g., each process has a specified maximum CPU utilization percentage. The rule allows authorized processes to exceed the percentage parameter for short periods but will kill the offending process after that, including the web server, which might be appropriate if its process is taken over by an intruder.

4. Software Isolation

4.1 Diversity

As described in the Section 3, COTS-provided diversity is part of HACQIT's design. Not only does it help with the detection through comparison mechanism, but it also is part of our isolation strategy. Most attacks use exploits of particular vulnerabilities in a software product, either a server application like

a web server or an operating system or one of its major components like the networking protocol stack. An exploit that works against one product of a type of software (web server, OS, etc.) will seldom work against another product of the same type. Consequently, as long as we have two different web servers operating on our primary and backup, the probability that an exploit that succeeds on one will propagate to the other should be small, even though we are passing the request that contains the exploit to both.

4.2 Random Rejuvenation

It is possible for an intrusion to become part of a legitimate process (create a new thread that lives within the process) but not do anything ("sleep") for an indefinite length of time. In this case, our detection mechanisms may not detect a failure until the malignant thread(s) "wake up" and attempt to do some damage. Random rejuvenation is a counter-measure for this type of intrusion. The MAC randomly initiates a failover with the average interval between random failovers set through the Policy Editor. This should minimize the effectiveness of "stealth" attacks that sleep before they cause errors we can identify. Typically, this value would be set at a few hours or more.

5. Plans and Conclusions

We have developed an intrusion tolerant system for a dynamic but simple web server application. We have developed a novel technique that uses application diversity to support both detection (through comparison of status codes) and isolation (by significantly reducing the likelihood of the same attack succeeding on both the primary and backup). We plan to increase the complexity of our web server based application to see if it scales to real-world use. We would like to host a second TCP/IP application, for example, email, which is inherently stateful, and see if the techniques we have developed would extend successfully. We would like to checkpoint the persistent stores for our applications using a commercial Database Management System so that we can provide fine-grained rollback once an error in data, perhaps caused by an undetected intrusion, has been discovered.

We believe at least three conclusions are supported by our work:

1 Even for the general case (any server on the Internet), it is possible to prevent repeated attacks from succeeding by the use of a sandbox to positively verify attacks to eliminate false positives. Note that the sandbox could run on a virtual machine and not require any additional hardware.

2 Diversity (or n-version programming) can be effectively used for intrusion tolerance, serving both detection and isolation functions.

3 For a bounded problem space (not an unbounded number of anonymous uses), it is possible to use techniques from the fault tolerance field, suitably modified, to increase the availability of our systems in the face of concerted cyber attacks.

Of course, these benefits have a cost in additional hardware, software, and administration.

References

[1] H. Abdel-Shafi, E. Speight and J. Bennet, Efficient user-level thread migration and checkpointing on Windows NT clusters, *Proceedings of the Third USENIX NT Symposium*, Seattle, Washington, 1999.

[2] A. Avizienis, The n-version approach to fault-tolerant software, *IEEE Transactions on Software Engineering*, vol. SE-22(12), pp. 1491-1501, 1985.

[3] R. Balzer and N. Goldman, Mediating connectors, *Proceedings of the Nineteenth IEEE International Conference on Distributed Computing Systems*, Austin, Texas, pp. 73-77, 1999.

[4] T. Barclay, J. Gray and D. Slutz, Microsoft terraserver: A spatial data warehouse, MS-TR-99-29, Microsoft Research, Advanced Technology Division, Redmond, Washington, 1999.

[5] S. Brilliant, J. Knight and N. Leveson, Analysis of faults in an n-version software experiment, *IEEE Transactions on Software Engineering*, vol. SE-16(2), 1990.

[6] M. Castro and B. Liskov, Practical Byzantine fault tolerance, *Proceedings of the Third Symposium on Operating System Design and Implementation*, New Orleans, Louisiana, 1999.

[7] T. Dahbura, System-level diagnosis: A perspective for the third decade, in *Concurrent Computations: Algorithms, Architecture and Technology*, Tewksbury, Dickson and Schwartz (eds.), Plenum, pp. 411-434, 1988.

[8] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, 1993.

[9] Y. Huang, P.E. Chung, C. Kintala, C.Y. Wang and D.R. Liang, NT-SwiFT: Software implemented fault tolerance on Windows NT, *Proceedings of the Second USENIX NT Symposium*, Seattle, Washington, pp. 47-54, 1998.

[10] Hypertext Transfer Protocol-HTTP/1.1 (www.w3.org/Protocols/rfc2616/rfc2616.html).

[11] L. Lamport, R. Shostak and M. Pease, The Byzantine generals problem, *ACM Transactions on Programming Languages and Systems*, vol. 4(3), pp. 382-401, 1982.

[12] J. Plank, M. Beck, G. Kingsley and K. Li, Libckpt: Transparent checkpointing under Unix, *Proceedings of the USENIX Winter Technical Conference*, New Orleans, Louisiana, pp. 213-223, 1995.

[13] F. Preparata, G. Metze and R.T. Chien, On the connection assignment problem of diagnosable systems, *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 848-854, 1967.

[14] L. Spainhower and T. Gregg, IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective, IBM reprint 0018-8646/99, 1999.

[15] J. Strouji, P. Schuster, M. Bach and Y. Kuzmin, A transparent checkpoint facility on NT, *Proceedings of the Second USENIX NT Symposium*, Seattle, Washington, pp. 77-85, 1998.

[16] L. Welch, B. Ravindran, B. Shirazi and C. Bruggeman, Specification and analysis of dynamic, distributed real-time systems, *Proceedings of the Nineteenth IEEE Real-Time Systems Symposium*, pp. 72-81, 1998.