Retrofitting Networked Applications to Add Autonomic Reconfiguration*

Michael G. Merideth and Priya Narasimhan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{mgm, priya}@cs.cmu.edu

ABSTRACT

To reduce user maintenance is an important goal for applications that must dynamically adapt based on their environments. There are many existing popular applications that lack support for this autonomic reconfiguration, but that are beginning to be used in these dynamic environments, in which they must update themselves frequently; not all of these applications will be completely redesigned and redeveloped in order to support autonomic features. In this paper, we explore how to retrofit pre-existing networked applications to add support for autonomic reconfiguration. To illustrate our methods, we retrofit a popular open-source intrusion detection system, *Snort*, to enable it to reconfigure itself using online program updates and information about its environment.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*software configuration management*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software maintenance*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; C.4 [**Computer-Communication Networks**]: Performance of Systems—*reliability, availability, and serviceability*; C.2.0 [**Computer-Communication Networks**]: General—*security and protection*

General Terms

Design, Management, Reliability, Security

Keywords

Autonomic, reconfiguration, intrusion detection, software upgrades

*This work is supported by the Army Research Office through grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to the Center for Computer and Communications Security at Carnegie Mellon University.

1. INTRODUCTION

As administrative costs begin to dominate the costs of software acquisitions, there is a great deal of interest in software with autonomic [14] capabilities. How software can best act autonomically is an area of active research [1, 13]. This leads to another important problem—there are a variety of popular open-source networked applications in wide use that are unlikely to be completely redesigned and redeveloped in order to support autonomic features. As the capability for these applications to configure and reconfigure themselves sees increasing demand, these applications will instead need to be retrofitted with autonomic capabilities.

Retrofitting autonomic capabilities onto applications need not be done in an ad-hoc fashion. We believe that there are principles and techniques that can be used to address commonalities within classes of applications to allow autonomic features to be cleanly added. There is a substantial body of work on building software that has the capability to update itself without incurring downtime [25, 2]; this capability shows promise in being used as a building block toward autonomic computing [4]. For self-updating to be useful, the software must be aware of alternate configurations, and may need to be able to determine the best configuration itself. The new configuration should be checked for suitability before the reconfiguration [16], and potentially rolled-back in the event of a problem after the reconfiguration.

In our exploration of autonomic reconfiguration, Network Intrusion Detection Systems (NIDSs) [19], which are an important class of application software in many networks, prove to be an interesting experimental case-study for a number of reasons. First, NIDSs currently lack support for autonomic reconfiguration. Second, from a configuration standpoint, one of the most interesting properties of a NIDS is that it must be reconfigured frequently to incorporate new ways to recognize new security problems; if a NIDS is not reconfigured, its utility actually decreases with time. Third, as we describe in Section 2.1.2, a reconfiguration process that requires downtime can result in additional vulnerabilities. Fourth, the configuration of a NIDS is very environment-dependent—no single configuration is best for all environments; this presents a problem because the environment can change continually. Finally, frequent reconfiguration of a NIDS may be also desirable in a number of new intrusion-detection usage models, which we discuss in Section 2.2.

As a case study, we introduce Flexiphant, a novel unified system architecture for retrofitting a NIDS to add support for dynamic reconfiguration that may enable it to meet the usage requirements of tomorrow. Flexiphant consists of mechanisms to gather configuration parameters about the network, to generate new system configurations based on these parameters, and to perform the reconfiguration of the NIDS dynamically, in a dependable, online fashion automatically, in response to a variety of administrator-tunable triggers. A NIDS enhanced with Flexiphant can reconfigure itself with targeted, more resource-conserving configurations that can allow it to perform better than before by (1) reducing its resource consumption (resulting in a lessened chance of error or failure), and (2) prioritizing the alerts it produces to increase its utility to the end-user.

The remainder of the paper is organized as follows. In Section 2, we discuss the characteristics and limitations of NIDSs that make them a good candidate class of application for incorporating autonomic capabilities. In Section 3, we present our case-study, Flexiphant, an architecture for retrofitting a NIDS with support for autonomic reconfiguration.[1] Section 4 discusses open research questions, while Section 5 presents related work. Finally, Section 6 concludes.

2. TARGET APPLICATION (NIDSs)

Network Intrusion Detection Systems (NIDSs) [19] are an important component of modern network security. NIDSs provide the critical security function of detecting attacks and compromises on a network. When configured properly, they provide early warnings of potential system compromises. Yet, they suffer from two related problems: high resource requirements, and the generation of unnecessarily large numbers of alerts. We propose that these problems are due in part to static, over-generalized system configurations that do not take into account the actual network services that need to be protected from attack; a NIDS capable of reconfiguring itself would not suffer from these limitations.

2.1 Limitations in Configuration

2.1.1 Choosing a Configuration

Most deployed NIDSs are signature-based, so they require a rule database that relates known attack signatures to rules for classifying and taking action on any malicious network traffic. Because new attacks emerge rapidly, this database must be updated frequently with signatures and rules for these new attacks.

The configuration of a NIDS specifies which rules in its database are active; if a rule is not active, then the NIDS does not have to watch for the attacks that would trigger only that rule. In common patterns of usage, when new rules and signatures are introduced, the NIDS administrator must decide which rules apply to the local network, and update the rules configuration, possibly using tools, *e.g.,* [22, 27], to help merge the new rules into the existing configuration.

[1]While the details of the internal workings of NIDSs in Sections 2 and 3 might seem application-specific and peripheral to the theme of the paper, these details are crucial to understanding how we added reconfigurability to an existing application. Additionally, these details provide insight into how we propose to build on this knowledge to handle other networked applications.

Consider the task of keeping the rules configuration of a NIDS ideal and up-to-date. Initially, the administrator must choose an appropriate starting configuration for the NIDS; the simplest thing to do is to enable all of the default rules with little regard as to the actual services on the network. This conservative approach may require unnecessarily or unmanageably high resource requirements, and may in fact overlook certain unusual services that are on the network but that are unlikely to be covered by a generic, default configuration.

To err on the side of safety, network administrators may be tempted to enable alerts for many more services than necessary. For example, a NIDS may be configured to monitor for FTP, HTTP, and Telnet attacks; yet, if no FTP, HTTP, or Telnet services are present on the network, the overhead of checking network traffic for the signatures of these attacks may be unnecessary and may result in alerts for attacks that (although potentially real) are necessarily ineffectual as there are no services for them to compromise.

In this case, erring on the side of safety can cause two types of problems. First, it has been shown that high workloads can cause a NIDS to miss attacks and even to crash [20]. Because a NIDS is typically required to process network traffic in real-time, and because network buffers are finite, extensive monitoring in heavy traffic can cause a NIDS to ignore important network traffic. Second, a high volume of alerts can make the job of the network administrator much harder, as it becomes more difficult to separate the important alerts from those that are less critical. Alerts for attacks on non-existent services may likely be considered to be of lower priority than alerts for attacks on real (and vulnerable) services. However, if there is no correlation between the services actually deployed on the network, and the attack traffic on the network, establishing such priorities is non-trivial.

2.1.2 Effecting a Configuration

It is perhaps surprising to note that certain widely-deployed modern NIDSs require a period of downtime for routine maintenance/reconfiguration. The current state of the practice for updating the popular open-source NIDS known as Snort [26] to defend against new attacks is to download the new rules and signatures, merge them into the current database, update the configuration manually, and *restart* the Snort process in order for the updates to take effect.

Unfortunately, this type of naive approach to performing reconfiguration can result in unexpected vulnerabilities [17]. Many modern NIDSs track various types of connection state in addition to performing stateless packet analysis. This connection-state tracking helps defend against a class of attacks that specifically targets the inherent lack of "memory" of stateless NIDSs.

Additionally, NIDSs *fail-open:* attacks can occur even if the NIDS is down. This is because NIDSs typically passively monitor network traffic, meaning that they analyze the traffic, but generally do not attempt to modify or filter it. This passive monitoring has the advantage that it is minimally intrusive because it does not affect the flow of network traffic. However, it also means that if a NIDS crashes or is down

for updates, the network will continue to function without the monitoring and protection provided by the NIDS.

Thus, one can see that the criticality of NIDSs, combined with the facts that they fail-open and may be subject to additional vulnerabilities caused by downtime, means that any successful reconfiguration strategy must minimize the NIDS downtime required. The Flexiphant technique described in this paper meets this requirement by providing autonomic, live (or online) NIDS reconfiguration based on changing requirements and changing environmental characteristics.

2.2 Arguing for Autonomic Reconfiguration

The set of services run on most networks is not static; this is particularly true of wireless and ad hoc networks. Therefore, a static configuration may not be appropriate over time, as new systems are added to the network, and existing systems are removed from the network. The new systems may introduce new services that are not accounted for by the current NIDS configuration, and the removed systems may remove the only instances of certain services on the network. In both cases, the NIDS should be reconfigured.

The NIDS itself may move to a new environment. For example, a network administrator could install a NIDS onto a mobile device, such as a laptop, and use it to patrol different areas of a wireless network. In this scenario, resource constraints such as the reduced processing power and limited battery life of the mobile device would further suggest a need for reconfiguration into precise, minimal configurations.

Administrator preferences may change. A network administrator who has just installed new HTTP servers may be concerned mostly with any attacks targeting these servers, but, if the servers prove resilient over time, the administrator may become more concerned with other attacks, *e.g.,* FTP attacks.

3. CASE STUDY: FLEXIPHANT

In this section, we present Flexiphant, a novel unified system architecture for retrofitting a Network Intrusion Detection System to add support for dynamic reconfiguration. Using Flexiphant, it is possible to create NIDSs that are capable of dynamically modifying their own rule configurations, both at deployment time and at run time, in order to defend against changing security risks and changing resource constraints.

The features of Flexiphant that are described in this section handle three important challenges in retrofitting a networked application to add support for autonomic reconfiguration: (1) they provide a way for the application to gather information about the state of its surroundings; (2) they provide a way to determine a new configuration; and (3) they provide a mechanism for this new configuration to be swapped into the system, without requiring that the system be restarted.

As discussed in Section 2.2, it may be desirable to trigger reconfiguration based on a number of factors, including changes in active services, changes in available resources, and changes in administrator preferences. All of these scenarios require dynamic reconfiguration support for determining the services on the network and for using this information to reconfigure the NIDS dynamically.

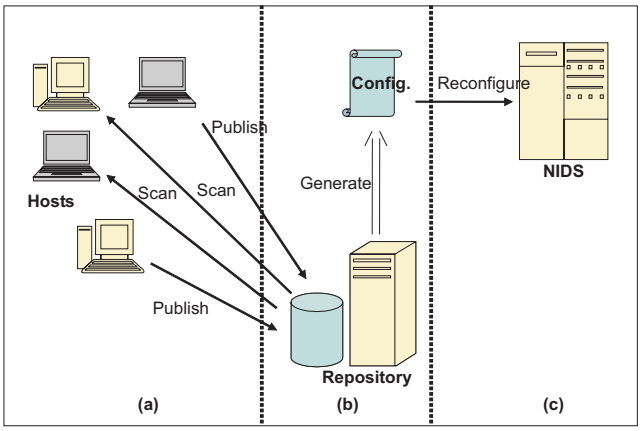

Figure 1: The Flexiphant system architecture.

3.1 Flexiphant System Architecture

In this section, we present the Flexiphant system architecture as a concrete example of a way to retrofit a specific networked application, *i.e.,* a NIDS, with support for autonomic reconfiguration. A Flexiphant-enabled NIDS continually analyzes its environment at runtime and dynamically reconfigures itself to work best in that environment.

Figure 1 presents the system architecture of Flexiphant: (a) Flexiphant analyzes its environment to collect data about the services that are available (and, therefore, potentially vulnerable) on the network, and holds and indexes this data in a service repository; (b) Flexiphant uses the information from its service repository to create new configurations based on administrator preferences; and (c) it dynamically reconfigures the running NIDS without incurring downtime. This three-stage process can be repeated throughout the life cycle of the Flexiphant system. These steps are described in this section in greater detail.

3.2 Gathering Configuration Parameters

Flexiphant basis its reconfiguration decisions partially on the set of services that are actually running on its network. Therefore, for a Flexiphant-enabled system to tune its configuration dynamically and effectively, it must collect data about these services. Flexiphant uses two complementary techniques to perform this data collection: in service scanning, the service repository scans for updates periodically; in service publishing, the servers on the network publish to the service repository information about the services that they host. Flexiphant stores the information that it collects in its service repository.

Service Scanning: Flexiphant can identify machines and determine the services that are running on these machines through network scanning tools like *nmap* [21]. The nmap tool can perform both TCP and UDP scans to determine the active services and the operating systems that run on a network of machines. This provides one way to determine

classes of active services (for example, FTP or WWW services) on the network, to determine the program name and version of many of the services, and to determine the number of machines running these services. This information is useful in its own right, but can be further refined through banner grabbing [24] to determine the actual services running in cases where nmap cannot reliably do so.

Service scanning can be viewed as a form of polling that requires active, periodic work by Flexiphant. Like any polling system, there is a tradeoff between how frequently to poll (scan), and how often the data (the list of running services) changes. If used as the sole means by which to obtain service data, this polling might present a challenge to scalability.

Service Publishing: Service publishing is a complementary alternative to service scanning. It relies on the network hosts to send information to Flexiphant about the services they run. Each host must have installed on it a Flexiphant publisher program, which can use host-specific techniques for determining available services. For example, on a Windows system, the Flexiphant publisher can make use of the Windows Registry and the Services list to determine not only the services that are currently running, but also those that are configured or registered to run and that therefore might be expected to run in the near future.

Service publishing faces some inherent reliability and security challenges. The Flexiphant publisher program on any given end host might be disabled either by a crash or by an end-user who either attempts to remove the program or to block it (*e.g.,* through the use of a personal firewall product) from accessing the network. It also might be modified or replaced in an attempt to provide Flexiphant with misleading information.

The Flexiphant Service Repository: In Flexiphant, the service information obtained through scanning and publishing is accumulated, indexed, and stored in a central location called the Flexiphant service repository. This repository is queried during the process of creating new system configurations. Storing the data in a central database has the advantage that the data can be readily shared with components other than Flexiphant that might need access to it. Much of the service data used by Flexiphant may already be gathered on some networks for other purposes, such as security auditing. In such situations, Flexiphant can be configured to use the existing data or to augment it where desirable.

3.3 Determining the Configuration

Flexiphant uses the configuration-parameter information to create new configurations. It uses its general information about the types of services running on the network in order to categorize coarsely the categories of rules that should be configured to run. This can allow Flexiphant to disable or lower the priority of large groups of rules in batches. The detailed information that Flexiphant gathers through service banner-grabbing and service publishing, about the services that are running or which may potentially run, can be used to tune further the configuration.

Flexiphant takes additional factors into consideration when assigning priorities. These factors include the importance of the particular service to the network, the importance of the machine on which the service is running (*e.g.,* an important server machine *vs.* a user's desktop), and factors commonly taken into account by NIDSs like the potential severity of the attack method.

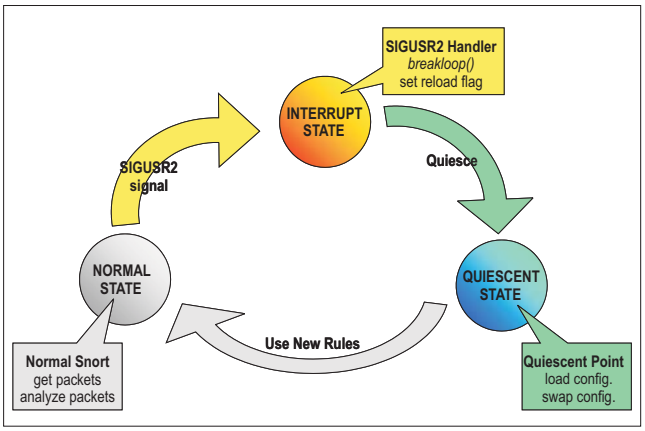

Figure 2: Elephant Reload implementation.

By design, Flexiphant can create intermediate configuration scripts that are not specific to any one NIDS in as much as possible. It then translates these scripts into configuration information for any of a variety of NIDSs. This feature can allow for multiple NIDS versions to be reconfigured to work on the same network.

3.4 Performing Reconfiguration

To perform dynamic reconfiguration, Flexiphant must be able to load new configurations as they become available, while maintaining data-integrity and state-consistency. Flexiphant uses the Elephant Reload algorithm [17] to perform the actual system reconfiguration in a live, online fashion. In the original Elephant system, the network administrator assembles a new rule configuration and manually instructs Elephant to reconfigure the NIDS. Flexiphant instead autonomically determines the best new rule configuration, and uses Elephant Reload to instruct the running NIDS to reload its rule configuration in response to environmental triggers that can be tuned by the administrator, substituting the newly generated configuration for the old without incurring downtime or losing state.

As seen in Figure 2, the Elephant Reload algorithm works as follows: first, it finds a quiescent point in the NIDS, *i.e.,* a point where the rules loaded by the current NIDS-configuration are not in use; next, it loads the rules required by the new NIDS-configuration into memory; and, finally, it replaces the current rules with the new rules. From an application consistency standpoint, it is important that the reconfiguration be applied at a quiescent point in order to avoid the possibility of network traffic being checked partially using the old configuration and partially using the new configuration.

4. LOOKING FORWARD

In presenting Flexiphant, we focus on three steps of an autonomic reconfiguration process: gathering information about

the environment, determining the new configuration, and applying the new configuration. We outline here other activities, including validating the new configuration, that form our future work.

Validating the Reconfiguration

There are questions about the benefit of each new configuration: is the system actually more reliable or more secure after each successive reconfiguration? How do we ensure that the reconfiguration itself does not degrade the resilience of the entire system? How do we know that the reconfiguration did not violate any quality-of-service requirements expected of the system?

Self-Stabilization

There is a set of questions related to how best to tune the system. While the process of reconfiguration should provide benefits in terms of resource usage and alert generation, reconfiguration itself requires time and resources. If reconfiguration is triggered too often, how is the system to stabilize? As the set of services on the network changes, there may be latency between the changes and the reconfiguration that is triggered by those changes. As the environment changes, the goal of reconfiguration may change accordingly; therefore, the autonomic reconfiguration functionality may need to be able to reconfigure itself.

Dependable Reconfiguration

The reconfiguration process itself needs to be reliable. Currently, the reconfiguration process in Flexiphant is atomic—if the NIDS crashes during reconfiguration, its configuration will still be consistent upon restart. Whether other networked applications can be made to share this trait is a subject of further investigation. We also have not implemented support for rolling-back configuration changes that prove problematic. Replication is frequently declared to be a solution for classes of reliability problems in networked applications; in the case of NIDSs, replication may not be the answer for reliable reconfiguration [17].

Transparent Reconfiguration

The availability of the source-code for our case-study application allows us to manipulate the internal behavior of the application directly. Specifically, as described in [17], finding a safe quiescent point is facilitated by the event-loop driven, single-threaded architecture of Snort, our case-study target application. Future work may consider retrofitting applications for which the source code is not available; this would necessitate treating the application as a black-box.

5. RELATED WORK

There has been a great deal of recent interest in autonomic computing [14]. This can be seen in the proceedings of recent workshops and conferences in the area, *e.g.,* [1, 13]. Research has focused on many areas of autonomic computing, including how to verify that a new configuration will not be problematic [16], how to choose the best adaptation from a set of possibilities [10, 23], and how to use software architecture for self-adaptation [8]. There has also been attention focused on how to take research technologies and make them adoptable for industrial use [6]; our work here seeks to do that for autonomic reconfiguration.

One of the most widely researched areas in software reconfiguration is software upgrading, which is surveyed by Ajmani [2] and Segal and Frieder [25]. In our case-study, we use Elephant Reload [17] to perform autonomic reconfiguration. Elephant Reload is similar to techniques for live software upgrades, but less complex because it is a solution specific to NIDSs that takes advantage of the fact that the rule configuration can be separated from any code that maintains state.

Our method of determining when it is safe to apply the new configuration relies on determining when the application is quiescent. Kramer and Magee [15] provides a fundamental definition of quiescence as the property that "[a] node is not within a transaction and will neither receive nor initiate any new transactions". The system presented in [15] relies on application nodes to become quiescent when instructed to, and provides an algorithm for moving a group of nodes into a quiescent state within a bounded period of time. Bidan *et al.* [7] follows [15], but attempts to minimize impact on concurrency by reducing the number of nodes that must be made quiescent at any one time; the technique is restricted to systems built from multi-threaded nodes, and works by passivating only a part of each node. In the Eternal System [18, 29], object quiescence is defined as any state in which the object is not executing any of its methods. Eternal guarantees distributed consistency by providing an atomic switch-over operation, which mandates the quiescence of all objects involved in the switch-over.

We make use of Elephant Reload to detect and enforce quiescence. There have been a number of other techniques introduced for finding quiescence. Both PODUS [25] and ABACUS [3], which operate at the procedure granularity-level, recognize quiescence by determining when a procedure is inactive. Gagliardi *et al.* [9] assumes that a module is quiescent when it has no IPCs outstanding, and requires that each module notify a third party component of any IPCs it is issuing. Hauptmann and Wasel [11] provides tools for locating potential quiescent points, but requires the programmer to associate explicity the quiscent points in the old and new versions. POLYLITH [12] provides mechanisms to export the state of a non-quiescent process; in this way the programmer need not identify quiescent points, but still must explicitly name reconfiguration points.

There has been work done on tools for offline management of the NIDS rule and signature updates that become available when new attacks are discovered. Oinkmaster [22] and SnortCenter [27] provide ways to download rule updates for Snort and to merge the new rules into the current rule configuration in an offline fashion. They do not provide any mechanism for applying the updates to a running Snort process, and they do not provide support for autonomically creating new rule configurations.

Axent NetProwler [5] is a commercial NIDS that can scan the network for services and automatically apply rule updates without needing to be restarted. However, the internal details of NetProwler have not been released publicly, and the configuration/reconfiguration step requires that the administrator manually choose the services to protect; NetProwler does not autonomically reconfigure itself. Snort-

Rules [28] is a small program for Snort that uses *nmap* [21] to determine the ports on the network that are active, and to allow the rules configuration to be filtered based on the list of active ports. SnortRules might serve as an initial implementation of the service scanning and configuration-filtering portions of Flexiphant.

6. SUMMARY AND FUTURE WORK

In this paper, we have argued that there will be a growing demand for autonomic software, and that existing software will have to be retrofitted to meet the demand. To learn lessons in retrofitting an existing application to add support for dynamic autonomic reconfiguration, we have presented a case-study.

Our case-study introduces Flexiphant, a novel unified system architecture for retrofitting a NIDS with support for dynamic reconfiguration that may enable it to meet the usage requirements of tomorrow. Flexiphant consists of mechanisms to catalog the services on a network, to generate new system configurations based on this catalog, and to perform the reconfiguration of the NIDS dynamically, in a dependable, online fashion automatically, in response to a variety of administrator-tunable triggers.

Our current research prototype of the Flexiphant system has enabled us to understand some of the challenges involved in retrofitting a pre-existing networked application to add autonomic reconfiguration. Our next steps will involve addressing some of the open research questions that we have outlined, particularly for other networked, and perhaps distributed, applications.

7. REFERENCES

[1] ACM SIGSOFT Workshop on Self-Managing Systems (WOSS04). `http://www.cs.cmu.edu/~garlan/woss04/`, October/November 2004.

[2] S. Ajmani. A review of software upgrade techniques for distributed systems. `http://pmg.csail.mit.edu/~ajmani/papers/review.pdf`, August 2002.

[3] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *USENIX 2000 Annual Technical Conference*, pages 307–322, San Diego, CA, June 2000.

[4] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. D. Silva, O. Krieger, D. J. Edelsohn, M. A. Auslander, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot-swapping. *IBM Systems Journal*, 42(1), 2003.

[5] Axent Netprowler. `http://www.c2000.com/products/sec_netp.htm`, February 2005.

[6] R. Balzer, M. Litoiu, H. A. Müller, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Fourth International Workshop on Adoption-Centric Software Engineering. In *Workshop at the 26th IEEE/ACM International Conference on Software Engineering (ICSE 2004)*, Edinburgh, Scotland, May 2004.

[7] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *Fourth International Conference on Configurable Distributed Systems*, pages 35–42, Annapolis, MD, June 1998. IEEE Computer Society Press.

[8] S.-W. Cheng, A.-C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[9] M. Gagliardi, R. Rajkumar, and L. Sha. Designing for evolvability: Building blocks for evolvable real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 100–109, Boston, MA, June 1996.

[10] D. Garlan, V. Poladian, B. Schmerl, and J. P. Sousa. Task-based self-adaptation. In *ACM SIGSOFT 2004 Workshop on Self-Managing Systems (WOSS'04)*, Newport Beach, CA, 2004.

[11] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *3rd International Conference on Configurable Distributed Systems*, pages 70–80, Annapolis, MD, May 1996.

[12] C. Hofmeister and J. Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *13th International Conference on Distributed Computing Systems*, pages 101–110, Pittsburgh, PA, May 1993.

[13] IEEE First International Conference on Autonomic Computing (ICAC-04). `http://www.caip.rutgers.edu/~eparashar/ac2004/`, May 2004.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, pages 41–50, January 2003.

[15] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[16] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 287–296, Helsinki, Finland, 2003.

[17] M. G. Merideth and P. Narasimhan. Elephant: Network intrusion detection systems that don't forget. In *Hawaii International Conference on System Sciences (HICSS-38)*, Big Island, HI, January 2005.

[18] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. A. Tewksbury, and V. Kalogeraki. Eternal: Fault tolerance and live upgrades for distributed object systems. In *DISCEX Information Survivability Conference*, pages 184–196, Hilton Head, SC, January 2000.

[19] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[20] D. Newman, J. Snyder, and R. Thayer. Crying wolf: False alarms hide attacks. *Network World*, June 2002.

[21] Nmap free security scanner for network exploration and hacking. `http://www.insecure.org/nmap/`, February 2005.

[22] Oinkmaster. `http://oinkmaster.sourceforge.net/`, October 2004.

[23] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic configuration of resource-aware services. In *26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, May 2004.

[24] G. K. S. McClure, J. Scambray. *Hacking Exposed*. McGraw-Hill, fourth edition, 2003.

[25] M. Segal and O. Frieder. On-the-fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, 10(2):53– 65, 1993.

[26] Snort. `http://www.snort.org`, February 2005.

[27] SnortCenter - Snort Management Console. `http://users.pandora.be/larc/`, February 2005.

[28] SnortRules. `http://www.andrew.cmu.edu/user/rdanyliw/snort/snortrules.html`, February 2005.

[29] L. Tewksbury, L. Moser, and P. Melliar-Smith. Live upgrades of CORBA applications using object replication. In *IEEE International Conference on Software Maintenance*, Florence, Italy, November 2001.