# Self-Protection in a Clustered Distributed System

Noel De Palma, Daniel Hagimont, Fabienne Boyer, and Laurent Broto

**Abstract**—Self-protection refers to the ability for a system to detect illegal behaviors and to fight-back intrusions with counter-measures. This article presents the design, the implementation, and the evaluation of a self-protected system which targets clustered distributed applications. Our approach is based on the structural knowledge of the cluster and of the distributed applications. This knowledge allows to detect known and unknown attacks if an illegal communication channel is used. The current prototype is a self-protected JEE infrastructure (Java 2 Enterprise Edition) with firewall-based intrusion detection. Our prototype induces low-performance penalty for applications.

**Index Terms**—Middleware, clustered systems, self-protection.

## 1 Introduction

The complexity of today's distributed computing environments is such that the presence of bugs and security holes is statistically unavoidable. A very promising approach to this issue is to implement a self-protected system which refers to the capability of a system to protect itself against intrusions, i.e., detect them and fight them back.

This paper presents a self-protected system in the context of cluster-based applications. We consider that the hardware environment is composed of a cluster of machines interconnected through a local area network with an Internet access via a router. The software environment is composed of a set of application components deployed on the cluster. These assumptions correspond to the point of view of a machine provider which rents his cluster infrastructure to different customers. We consider that each customer has a set of machines exclusively allocated to his applications. However, the local network and the Internet access are shared by all the applications. Therefore, the threat may come from outside of the cluster through the Internet, but also from inside because of a hostile accredited customer.

The approach is based on *the principle of least privilege* applied to communication channels. Any attempt to use an undeclared communication channel is trapped and a recovery procedure is executed. Legal communication channels are automatically calculated from the hardware and software architectures of the system and are used to generate protection rules that forbid the use of unspecified channels. Moreover, if the architecture of the system evolves, the protection rules are updated accordingly without any human intervention. The main characteristics of our system are: 1) to minimize the perturbation on the managed system while providing a high reactivity, 2) to automate the configuration (and reconfiguration) of security components when the system evolves, and 3) to keep the protection manager (which implements the protection policy) independent from the protected legacy system. The main limitation relates to the scope of the detected attacks and to the supported communication protocol; the current system can only detect attacks which use illegal communication channels based on the TCP/IP protocol. In order to validate our approach, we applied it to the self-protection of a cluster of machines which hosts a web application structured as a JEE architecture.

The remainder of the article is organized as follows: Section 2 presents the related work. Section 3 presents our design. The evaluation is reported in Section 4. We conclude in Section 5. The implementation details are reported in Annex I, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.161.

## 2 Related Work

This section reviews the main tools and techniques currently used by security experts to fight against intrusions and the existing systems which implement a self-protected behavior.

### 2.1 Intrusion Detection

Two main approaches have been explored [20] to ensure intrusion detection: *misuse intrusion detection* and *anomaly intrusion detection*. These approaches have been used in the case of Firewalls and Intrusion Detection Systems (IDS). While Firewalls are often used as filtering gateways to detect and to block illegal communication in real time, IDS mainly work offline and perform deep analysis to trigger alarms afterward. *Misuse intrusion detection* aims at detecting traces of well-identified attacks. The principle is to rely on a database which gathers well-known attack scenario

- N. De Palma and F. Boyer are with the INRIA - SARDES Research Group, University of Grenoble, 655 avenue de l'Europe, Montbonnot 38334 St-Ismier Cedex, France.
E-mail: noel.depalma@inrialpes.fr, Fabienne.Boyer@imag.fr.
- D. Hagimont and L. Broto are with the IRIT/ENSEEIHT, University of Toulouse, 2 rue Charles Camichel - BP 7122, 31071 Toulouse cedex 7, France. E-mail: {Daniel.Hagimont, Laurent.Broto}@enseeiht.fr.

*Manuscript received 6 Apr. 2010; revised 30 Nov. 2010; accepted 24 Mar. 2011; published online 25 May 2011.*
*Recommended for acceptance by M. Parashar.*
*For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-04-0202.*
*Digital Object Identifier no. 10.1109/TPDS.2011.161.*

specifications (also called *attack signatures*). The intrusion detector searches the database for a signature which matches the observed behavior (registered in an *audit*) and it raises an alert if one is found. Snort [19] is an example of such systems. This approach induces a small number of false-positives but cannot detect unknown attacks. *Anomaly intrusion detection* tries to spot irregular behaviors of the system by defining the normal behavior of the system (instead of attacks). The system can be observed and any misbehavior is signaled. An early work [6] modeled and verified behavior correctness at the level of system calls. Recent examples of anomaly-based detection can be found in [17], [8], [9], and [5]. This approach can detect unknown attacks but at the price of a lot of false positives.

## 2.2 Backtracking Tools

Backtracking tools [14] record detailed data about the system activity so that once an intrusion attempt has been detected, it is possible to determine the sequence of events that led to the intrusion and the potential extent of the damage (e.g., data theft/loss). The *Taser* system [10] provides the ability to restore the system in a trusted state. It enhances the file system with a selective self-recovery capability. Taser logs all file system access for each process. If a process is compromised, Taser computes illegal access for each file and is able to rollback illegal modifications. Such backtracking tools can help to automate parts of this process but human expertise is still required for an accurate understanding of the attack.

## 2.3 Self-Protected Systems

Self-protected systems are systems which are able to autonomously fight back intrusions in real time.

*Rootsense* [15] is an example of self-protected system. It differs from classical IDS in the sense that it detects and blocks intrusions in realtime. It audits events within different level of the host operating system and correlates them to comprehensively capture the global system state. It restricts the detection domain to root compromises only; doing so reduces runtime overhead and increases detection accuracy. It also adopts a dual approach to intrusion detection: a root penetration detector detects attempts to hijack the system and a root misbehavior detector tracks misbehavior by root processes (if the system was hijacked).

*MLIDS* [1] (multilevel intrusion detection system) is another example of self-protected system. It automates the detection of network attacks and proactively protect against them. MLIDS analyzes network traffic using three levels of granularities (traffic flow, packet header, and payload), and employs an efficient fusion decision algorithm to improve the overall detection efficiency and minimize the occurrence of false alarms.

The *Self-cleansing system (SCS)* [11] is another solution to build self-protected software. It targets stateless replicated servers (e.g., web servers) involving a load-balancing strategy. This pessimistic approach makes the assumption that all intrusions cannot be detected and blocked. In fact, the system is considered to be compromised after a certain time. Hence, this approach periodically reinstalls a part of the system from a secure repository. However, this solution only applies to stateless components.

## 2.4 Summary

With most common security tools, human administrators are heavily solicited by the produced alarms. In particular, after checking the relevance of alarms, they are usually in charge of initiating lots of actions, both for coordinated defense at the cluster scale (e.g., through reconfiguration of the security software) and investigation (e.g., with backtracking tools). SCS targets specifically a load balancing pattern and the backend replicas must be stateless. Moreover SCS does not provide any detection mechanism—it simply periodically reinstall part of the system. Taser is not fully automated and requires a human intervention if a dependency is found between an illegal and a legal access. Moreover Taser is restricted to a single host. Rootsense adopts both a misuse intrusion and an anomaly detection approach. It covers a broad range of attack but still may fire a high number of false-positive. It also targets a singlehost rather than a clustered-wide infrastructure. MLIDS analyzes network traffic using three levels of granularities. It reduces the number of false-positive but is very sensitive to the payload accuracy.

From this related work, we analyze that a self-protected system should 1) be fully automated both in its configuration and its reaction to intrusions, 2) fire near-zero false-positive since the response is automated, and 3) induce a low-performance overhead on an application performance to enable real-time protection.

## 3 Design of the Self-Protected System

Our approach relies on the capacity to maintain a consistent view of the global architecture of the cluster in terms of machines, software and their interconnections (the *sense of self* in Forrest's terminology [7]). For that purpose, human administrators use a deployment manager provided by the infrastructure to remotely install and interconnect software in the cluster. This deployment manager is the only way to add or remove software in the cluster. Therefore, this manager initializes the view of the global architecture and traps all modifications to maintain the consistency of the view.

The self-protection manager relies on this view of the global architecture. It is able 1) to compute from this architecture the legal communication channels, 2) to detect and block any deviation from these communication channels, and 3) to isolate the machine belonging to the cluster that breaks these communication channels. In other words, our self-protection manager is *deployment-aware*: the protection rules that guard the system in terms of legal communication channels are configured automatically from the architecture and are updated accordingly when this architecture evolves. At any time, only the minimal set of communication channels is opened.

## 3.1 Sense of Self

The *sense of self* is implemented by managing a data structure which describes the architecture of the system. This data structure that we call the *System Representation* is built using a component model. Components involve two kinds of relations: 1) bindings, which model service dependencies between elements and, 2) composition links, which represent containment dependencies (used to model

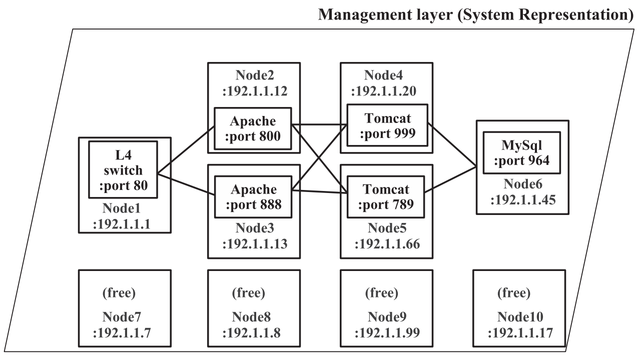

Fig. 1. System representation for a clustered JEE system.

the structure of the system). For instance, the relation between a machine and a software running on that machine is reified as a composition link in the *System Representation*. Similarly a cluster is represented as a composite component which includes a set of machine subcomponents. Software interactions are reified as component bindings and software configuration variables are reified as component attributes. The system is aware of 1) the machines that belong to it, 2) the software that run inside, and 3) the legal communication channels which involve each software across the machines and their configuration. This awareness is used to distinguish between normal and abnormal behaviors. Any software element which does not belong to the architecture can be viewed as a foreign element. This detection method allows to detect known and unknown attacks but only if they use illegal communication channels.

Any element (machine or running software) is therefore represented as a component in the *System Representation*. This component associated with an element of the system is also used to control the element, i.e., manage its configuration and life cycle at runtime. Therefore, each component wraps an element to implement its management interface. All wrappers provide the same management interface for the encapsulated elements. This interface allows managing the configuration elements, relations, and lifecycle without having to deal with complex, proprietary configuration interfaces, which are hidden in the wrappers. Wrappers implementations are specific to each software. However, it is our experience that such wrappers are small and easy to write [18].

Therefore, the *System Representation* implements a management layer on top of the legacy layer (composed of the actual managed software and hardware). Fig. 1 provides a full description of the *System Representation* in the case of a JEE architecture [12] which includes hardware nodes. It depicts an L4-switch which balances the requests between two Apache server replicas. These servers are connected to two Tomcat server replicas. The Tomcat servers are both connected to the same MySQL server. For the sake of clarity, we omit to represent other basic services that may be deployed into the cluster such as DNS or NFS Servers. It is important to point out that any provided service must be explicit in the System Representation as well as the bindings that reify the communication channels between the clients of a service and the service itself. Through wrappers, the *System Representation* provides all the facilities required for:

*Introspection*. The introspection interface allows observing components. For instance, an administration program can inspect an Apache web server component (encapsulating the Apache server) to discover that this server runs on node2:port 800 and is bound to a Tomcat Servlet server running on node4:port 999. It can also inspect the overall cluster infrastructure, considered as a single composite component, to discover that it is composed of one L4-switch, two Apache servers interconnected with two Tomcat servers connected to the same MySQL database server. It can also discover that no software is running on nodes 7, 8, 9, and 10.

*Reconfiguration*. The reconfiguration interface allows control over the component architecture. In particular, this control interface allows changing component attributes or bindings between components. These changes are reflected onto the legacy layer by wrappers.

Thanks to the Introspection interface provided by wrappers, the self-protection manager is able to discover the legal communication channels of the system: a legal channel between two nodes is represented by a binding between two components; the port on which the application is running and the IP addresses of the nodes. This allows the detection of any attack breaking the structural rules of an application. For instance a communication between node 1 and node 6 is identified as an attack as well as a communication between node 2 and node 4 on a different port than the port specified by the existing binding between these two machines.

Depending on the accuracy of the *System Representation*, the protection system could generate false-positives or false-negatives. A 100 percent accurate *System Representation* will not induce any false-positive. A intruder which does not break explicated communication channels remains undetected, which corresponds to false-negatives. We will see in the following section that reconfiguration is a key to manage the evolution of the *System Representation* and therefore the evolution of the system and its legal behavior.

## 3.2 Self-Protection Manager

The self-protection manager is responsible of the management of the *System Representation* and its use to detect illegal communications and to take counter-measures.

### 3.2.1 Management of the System Representation

In order to manage such a *System Representation*, we rely on the services associated with the component framework we used (Fractal [3]). Traditionally, a component framework provides services for the deployment of a component architecture and the modification (reconfiguration) of this architecture. Therefore, any administration action (machine or software installation or startup) is achieved as an action on the component architecture and reflected on the real environment, which implicitly maintains consistency between the two levels. In order to install a software, a component is deployed in the *System Representation*. Similarly, to uninstall a software, its associated component is removed from the *System Representation*. Therefore, all the changes in the system are first performed on the *System Representation* which then reflects them on the legacy system.

Once the environment is deployed, the system must be able to track communications and decide if they are legal or not. Our implementation of this detection mechanism is detailed in Annex I, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.161, it is based on protection rules which specify legal communication paths. These rules are automatically generated from the binding between components (discovered by introspecting the *System Representation*). Moreover, any change in the environment must be taken into account by this detection system, which requires to: 1) identify the communication paths impacted by the changes: components may have been added/removed in the *System Representation*, a binding between two components may have changed or the configuration of a component may have been modified. 2) Update the protection rules according to the impacted communication paths, in order to detect behaviors which don't comply with the updated *System Representation*. Protection rules are distributed and implemented on each nodes as sensors and actuators. Actuators allow update of protection rules and sensors are responsible for detection of protection rule violations.

#### 3.2.2 Reaction to a Detected Intrusion

If an illegal communication is detected, different policies can be applied. In our self-protected system, we enforce an isolation policy which consists in isolating the compromised machines from the network. In reaction to the violation of a communication channel, the manager identifies (from the *System Representation*) the machine to isolate and all the software running on that machine. It deletes all the bindings which involve the identified components in the *System Representation*. The modification of the *System Representation* will automatically update the protection rules associated with the involved set of machines, which will forbid any communication from/to the suspected machine. This simple policy isolates the incriminated machine to limit the damages caused by the intrusion, but it may lead to service unavailability. Another policy is to force the failure of the incriminated machine (fail-stop) and to provide a repair algorithm (detailed in [18]).

#### 3.2.3 Securing the Protection System

Enforcing protection of the self-protection system consists in implementing a reference monitor which is responsible for assigning and verifying protection rules, and assuring the security of the system. This enforcement relies on three key properties: 1) the enforcement that the self-protection manager cannot be compromised. 2) the enforcement that sensors and actuators cannot be compromised. 3) since the previous entities are distributed, communications between these entities must not be subject to attacks.

The self-protection manager receives reports from sensors and triggers operations on actuators to adapt protection rules. The self-protection manager, sensors, and actuators can be run in privileged mode (and therefore trusted) and if communications are trusted, the distributed control loop is trusted. Actuators must authenticate the commands sent by the self-protection manager. Similarly the manager authenticates sensors reports. This can be achieved by using an asymmetric authentification mechanism (such as TLS/SSL) that enforces authentification and avoids message forgery.

If we only consider attacks against applicative components, the privileged mode which enforces protection of these sensors, actuators, and the self-protection manager can be the kernel mode. If we consider that the kernel is subject to threats (e.g., kernel level rootkits), then the privilege mode can be at a lower level such as a hypervisor which isolates control loop components from the potentially compromised kernels. In this case, the description of the legal communications paths should involve kernel communications.

### 3.3 Positioning the Design

This section characterizes the design of our self-protected system.

Concerning Forrest et al.'s terminology [7], our design is distributed because protection rules, which are automatically configured from the *System Representation*, are enforced independently by each node of the cluster. Moreover, the self-protection manager is duplicated on multiple nodes to avoid a single point of failure. Our system is also

Concerning Kephart's definition of Autonomic Computing [13], our design is based on feedback loops (i.e., closed control The self-protection manager relies on a component-based *System Representation* which provides the knowledge of the system.

Concerning Software engineering, our design can be related to Model@runtime systems. Model@runtime [2], [4], [16] leverages model-driven engineering techniques (MDE) at design time as well as at runtime. Models@runtime promotes a causally connected representation of the underlying system. However, such representation is based on the artifacts produced from the MDE process and the software engineering methodologies employed. Our reflective component-based approach manipulates lower level abstractions that are related to the computation models and represents runtime entities and communication channels. Moreover Model@runtime has never been experimented nor evaluated in the context of self-protection. Our solution adapts automatically the protection rules on each nodes in the case of the deployment or modification of applications in the cluster.

## 4 Experimental Validation

### 4.1 Experimental Environment

We have chosen JEE clusters to evaluate our self-protection system. This platform allows the construction of web application services that are organized in 3 or 4 tiers: a web server tier, a presentation tier, a business tier (optional), and a database tier. Each tier can be duplicated for dependability and availability reasons as depicted Fig. 2.

We used Netfilter[1] (a packet filtering framework provided by the Linux kernel) on each node to enforce protection rules. Netfilter is used to build actuators for configuring protection rules and isolating machines (when intrusions are detected), and to build sensors which detect illegal JEEs communications (see Annex I, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.161, for more details).

1. www.netfilter.org.

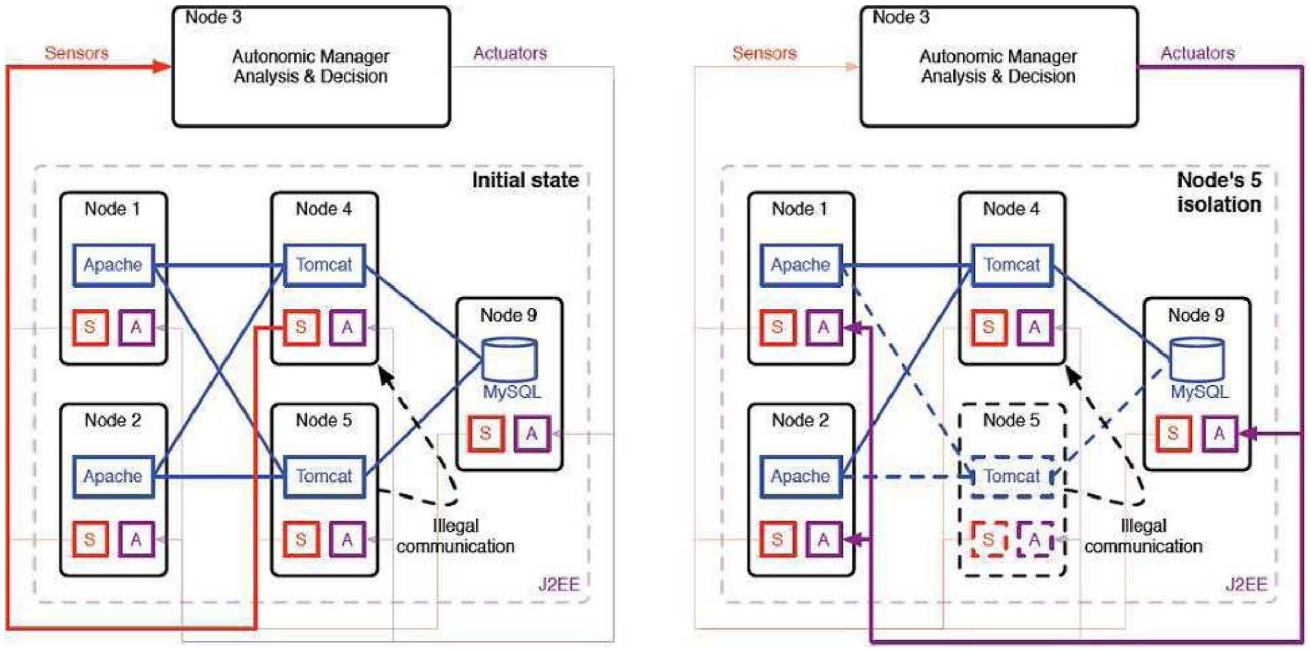

Fig. 2. Isolation example.

Our evaluation is based on the RUBiS benchmark, which emulates an online auction application. The infrastructure corresponds to that of Fig. 2. Load injection in RUBiS is configured by two transition matrix: a read-only matrix (or browsing mix) and a read-write matrix (or shopping mix). These matrix correspond to typical web application workloads. Software versions are: Java v1.6, Tomcat 5.5, MySQL v5.0, and Fedora Core 6 Linux. The physical machines have the following specifications: Intel Core Duo 1.66 GHz, 2 GB memory, and Ethernet Gigabit network.

### 4.2 Level of Protection

Our sensors detect all the communication not explicitly authorized in the *System Representation*. Hence, it is possible to react to all kind of attacks (known and unknown) using an illegal communication channel. For instance, it is possible to detect a port scanner and block the attack before the real intrusion. However, it is impossible, for the moment, to detect the attacks which follow a legal communication channels such as SQL injections in the case of a JEE infrastructure.

### 4.3 Control Loop Reactivity

This experience evaluates the time between the detection of an illegal communication and the isolation of compromised nodes. We measured the delay between the detection of an illegal communication coming for node 4 and the reconfiguration operations on nodes 1, 2, and 9 in order to isolate node 5. The average time measured (over 1,000 runs) is 2.133 ms with a 0.146 ms standard deviation. Our prototype is very reactive and can quickly block an intruder.

### 4.4 Control Loop Intrusivity

This experience evaluates the impact of the protection control loop on the performance of RUBiS. The load injector emulates a variable number of clients (from 0 to 3,000 in our experiments) sending a series of requests. The results of our experiments are depicted on Figs. 3 and 4. Fig. 3 illustrates the browse_only_matrix scenario whereas Fig. 4 is about the default_matrix scenario. In both scenarios, we increment progressively the number of clients until we reach the saturation point. We compare the throughput with and without the self-protection system when ramping up (i.e., when the load is increasing progressively between 0 and

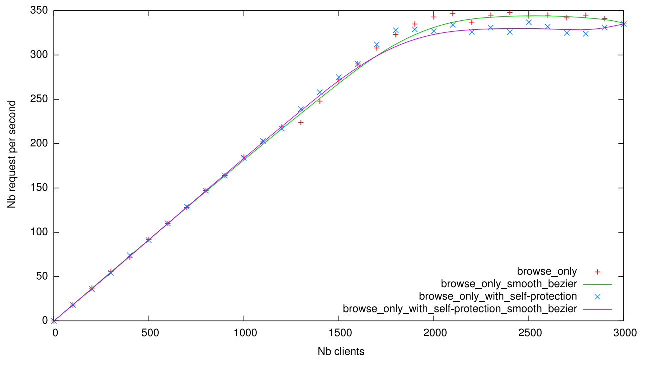

Fig. 3. Bandwidth for the browse_only_transitions matrix.

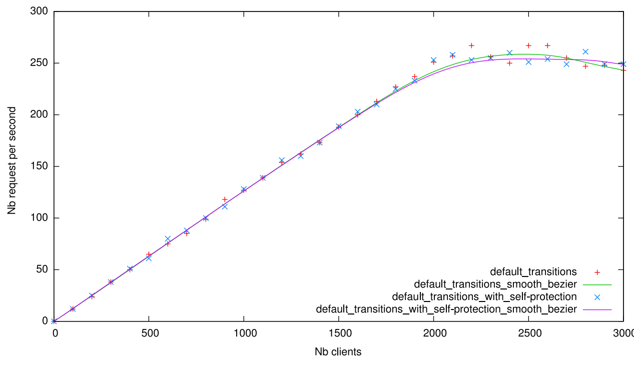

Fig. 4. Bandwidth for the default_transitions matrix.

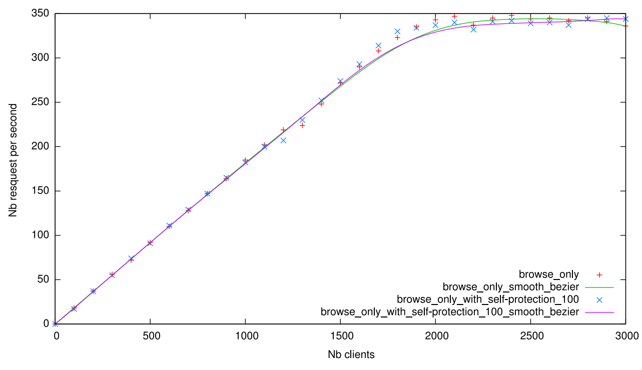

Fig. 5. Bandwidth for the browse_only_transitions matrix.

2,000 clients in parallel) and until saturation (between 2,000 and 3,000 clients in parallel). The results show that for each matrix, in both regimes (ramp-up and saturation), the throughput with and without the self-protection system are very close (between 0 and 4 percent overhead).

However, the filtering rules corresponding to Fig. 2 require a dozen of rules per machine. This number of rules may impact the system's performance. To check the scalability of our system, we have inserted 100 additional rules in each machines before the 10 real rules which were generated for our JEE testbed. This amount of rules represents a medium size cluster composed of approximately 50 nodes. Results of this latter experiment are given on Fig. 5. We can see that the overhead induced by the additional rules is also very low.

Our last experiment measures the response time of the system to client's requests. We measure the delay between the emission of a request by a client and the reception of the response from the corresponding server. We only evaluate the delay for read requests (browse_only_matrix) because they are the fastest kind of request and therefore the most penalized by the overhead induced by self-protection. Results represent the average response time with and without our system for 1,000 runs occurring after the ramp-up phase (i.e., the server's caches are full). We notice a reasonable penalty on response time with a maximal overhead of 3.5 percent.

## 5 Conclusion

Today, distributed computing environments are increasingly complex and difficult to secure. Following the autonomic computing vision, a very promising approach to deal with this issue is to implement a self-protected system which is able to distinguish legal (*self*) from illegal (*nonself*) operations. The detection of an illegal behavior triggers a counter-measure to isolate the compromised resources and prevent further damages. In this vein, we have designed and implemented a self-protected system whose main characteristics are: 1) to minimize the perturbation on the managed system while providing a high reactivity, 2) to automate the configuration (and reconfiguration) of security components when the system evolves, 3) to keep the protection manager (which implements the protection policy) independent from the protected legacy system.

We showed how to take advantage of the knowledge of a component-based application to provide a means of distinction between legal and illegal operations. We implemented a prototype system for a realistic use case, a clustered JEE application. When an illegal communication is detected, the *self-protection manager* quickly isolates the compromised nodes. The overhead induced by our approach is very low and acceptable for high-performance data servers.

For the moment, the scope of the detected attacks is limited to illegal communications over TCP/IP. We are thus unable to spot intruders respecting the expected control flow and/or targeting different protocols. Our work mostly targets controlled environments such as data centers (where most nodes are trusted) and "silent" attacks (aimed at quietly stealing or destroying data) rather than open grids and denial-of-service attacks. Our approach is well suited to the context of multitier applications deployed in a data center because an attacker knows a priori little about the structure of the system and will likely have to expose itself while exploring the network and trying to hijack other nodes. However, our current proposition may not be very helpful for peer-to-peer systems, where anyone acts as a router and can easily determine the architecture of the application. Currently, the only counter-measure implemented is the isolation of compromised nodes. Our prototype nonetheless provides an easy way to develop various counter-measures (e.g., reinstalling compromised nodes, starting an intrusion backtracking procedure, etc.).

We also intend to investigate new mechanisms to spot and block attacks targeted at legacy softwares (SQL injection, etc.) that follow legal communication channels. Since, in our model, legacy softwares are wrapped by manageable components, it is possible to encapsulate information about their normal behaviors. For instance, one could specify the children processes expectedly created by a particular application in order to block an illegal *fork/exec*. We may also add the definition of well formed requests to prevent exploits like *SQL injections* on the database. We also intend to investigate the support for other communication protocols.

## Acknowledgments

The work reported in this chapter benefited from the support of the French National Research Agency through projects Selfware (ANR-08-SEGI-017).

## References

[1] Y. Al-Nashif, A.A. Kumar, S. Hariri, Q. Guangzhi, L. Yi, and F. Szidarovsky, "Multi-Level Intrusion Detection System (ML-IDS)," *Proc. Int'l Conf. Autonomic Computing*, pp. 131-140, 2008.

[2] G.S. Blair, N. Bencomo, and R.B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22-27, Oct. 2009.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, "The Fractal Component Model and Its Support in Java," *Software—Practice and Experience*, vol. 36, nos. 11/12, pp. 1257-1284, 2006.

[4] B.H.C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty," *Proc. ACM/IEEE Int'l Conf. Model Driven Eng. Languages and Systems*, 2009.

[5] L. Ertoz, E. Eilertson, A. Lazarevic, P. Tan, J. Srivastava, V. Kumar, and P. Dokas, *The MINDS-Minnesota Intrusion Detection System Next Generation Data Mining.* MIT Press, 2004.

[6] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A Sense of Self for Unix Processes," *Proc. IEEE Symp. Research in Security and Privacy,* 1996.

[7] S. Forrest, S.A. Hofmeyr, and A. Somayaji, "Computer Immunology," *Comm. the ACM,* vol. 40, no. 10, pp. 88-96, 1997.

[8] D. Gao, M.K. Reiter, and D. Song, "Behavioral Distance for Intrusion Detection," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID '05),* Sept. 2006.

[9] J.T. Giffin, D. Dagon, S. Jha, W. Lee, and B.P. Miller, "Environment-Sensitive Intrusion Detection," *Proc. Int'l Symp. Recent Advances in Intrusion Detection,* Sept. 2005.

[10] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara, "The Taser Intrusion Recovery System," *Proc. 20th ACM Symp. Operating Systems Principles,* 2005.

[11] Y. Huang and A. Sood, "Self-Cleansing Systems for Intrusion Containment," *Proc. Workshop Self-Healing, Adaptive and Self-MANaged Systems,* 2002.

[12] Sun Microsystems, *Java 2 Platform Enterprise Ed. (J2EE),* http://java.sun.com/j2ee/, 2011.

[13] J. Kephart, *An Architectural Blueprint for Autonomic Computing.* IBM White Paper, 2003.

[14] S.T. King and P.M. Chen, "Backtracking Intrusions," *ACM Trans. Computer Systems,* vol. 23, no. 1, pp. 51-76, 2005.

[15] R. Koller, R. Rangaswami, J. Marrero, I. Hernandez, G. Smith, M. Barsilai, S. Necula, and S. Masoud, "Anatomy of a Real-Time Intrusion Prevention System," *Proc. Int'l Conf. Autonomic Computing,* pp. 151-160, 2008.

[16] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel, "Taming Dynamically Adaptive Systems Using Models and Aspects," *Proc. IEEE Int'l Conf. Software Eng.,* 2009.

[17] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna, "Anomalous System Call Detection," *ACM Trans. Information and System Security,* vol. 9, no. 1, pp. 61-93, Feb. 2006.

[18] S. Sicard, F. Boyer, and N. De Palma, "Using Components for Architecture-Based Management: The Self-Repair Case," *Proc. Int'l Conf. Software Eng.,* 2008.

[19] M. Roesch, "Snort—Lightweight Intrusion Detection for Networks," *Proc. Large Systems Administration Conf.,* Nov. 1999.

[20] A. Sundaram, "An Introduction to Intrusion Detection," *ACM Crossroads Student Magazine,* vol. 2, no. 4, pp. 3-7, 1996.

**Noel De Palma** received the PhD degree in computer science in 2001. Since 2002 he was an associate professor in computer science at the University of Grenoble (ENSIMAG/INP). Since 2010, he is a professor at Joseph Fourier University. He is a member of the SARDES research group at LIG laboratory (UJF/CNRS/Grenoble INP/INRIA), where he leads researches on Autonomic Computing, Cloud Computing, and Green Computing.

**Daniel Hagimont** received the PhD degree from Polytechnic National Institute of Grenoble, France in 1993. He is a professor at Polytechnic National Institute of Toulouse, France and a member of the IRIT laboratory, where he leads a group working on operating systems, distributed systems and middleware. After a postdoctorate at the University of British Columbia, Vancouver, Canada in 1994, he joined INRIA Grenoble in 1995. He took his position of the professor in Toulouse in 2005.

**Fabienne Boyer** received the PhD degree in computer science in 1995, while she was a joint researcher of Bull Laboratories and the IMAG research Center from the University Joseph Fourier at Grenoble (France). Since 1995, she is an associate professor in computer science at the University Joseph Fourier. She is a member of the SARDES research group at LIG laboratory (UJF/CNRS/Grenoble INP/INRIA), where she works on software reconfigurability with a special emphasis on reflective models.

**Laurent Broto** received the PhD degree from the Toulouse University, France in 2008. He is an associate professor at Polytechnic National Institute of Toulouse, France and a member of the IRIT laboratory, where he is member of a group working on operating systems, distributed systems and middleware. After a postdoctorate at the Oak Ridge National Lab, Oak Ridge TN, 2009, he took his position of an associate professor at Toulouse in 2009.