A Genetic Algorithms-Based Approach for Optimized Self-protection in a Pervasive Service Middleware

Weishan Zhang[1], Julian Schütte[3], Mads Ingstrup[1], and Klaus M. Hansen[1,2]

[1] Aarhus University
{zhangws,ingstrup}@cs.au.dk
[2] University of Iceland
kmh@hi.is
[3] Fraunhofer Institute for Secure Information Technology
julian.schuette@sit.fraunhofer.de

Abstract. With increasingly complex and heterogeneous systems in pervasive service computing, it becomes more and more important to provide self-protected services to end users. In order to achieve self-protection, the corresponding security should be provided in an optimized manner considering the constraints of heterogeneous devices and networks. In this paper, we present a Genetic Algorithms-based approach for obtaining optimized security configurations at run time, supported by a set of security OWL ontologies and an event-driven framework. This approach has been realized as a prototype for self-protection in the Hydra middleware, and is integrated with a framework for enforcing the computed solution at run time using security obligations. The experiments with the prototype on configuring security strategies for a pervasive service middleware show that this approach has acceptable performance, and could be used to automatically adapt security strategies in the middleware.

1 Introduction

Security is an important quality of service (QoS) requirement in pervasive computing systems. On the one hand, the higher security, the better. On the other hand, resource restrictions on pervasive computing devices may compromise the security requirements, as usually the higher security, the more resources are needed to implement and enforce them. Therefore, an interesting concern in relation to system quality is not how secure or efficient a system can be made, but rather how secure we can afford to make a system given the constraints set by available resources and other requirements, such as memory consumption and latency. Tradeoffs between security, performance, and resources are always involved, especially in pervasive computing systems.

Hence, an investigation on how to obtain an optimized solution following security, resource, and performance requirements is an interesting issue. Although several research contributions have been made towards making security mechanisms adaptable [1], we have found that most of this work focus on security in isolation rather than on managing an appropriate tradeoff between several quality attributes at runtime.

In this paper we present a way for systems to dynamically optimize the tradeoffs between security, resources and performance as users' preferences are changed to reflect, at run time, the relative importance of these three quality attributes. We have accomplished this by relying on a general architecture for self management developed in the EU-funded Hydra project[1], in which Genetic Algorithms (GAs) [2] are used to obtain optimized solutions at run time, from a number of conflicting objectives.

Approaching adaptive security from the perspective of making systems self-managing has particular merit because security is thereby managed alongside other quality attributes. Moreover, since even security mechanisms that are arguably simple to use are frequently misunderstood and applied incorrectly by end users [3], automating their configuration may make systems more secure by precluding their incorrect configuration by human operators, who express their goals declaratively as policies.

The remainder of the paper is organized as follows: First we explain the self-management architecture of Hydra and how its components interact to optimize self-protection (section 2). Our approach uses semantic models of resource consumption and security characteristics, which are described in Section 3. Section 4 describes a scenario of self-protection and the security strategies used within it. Next, section 5 describes how genetic algorithms are used to optimize protection in face of specific resource requirements. Section 6 presents our prototype implementation, and evaluations that show our approach can perform acceptably. Finally, we review related work (section 7) and conclude the paper (section 8).

2 Semantic Web-Based Self-management and Work Flow of Self-protection in Hydra

2.1 Self-management Architecture

The Hydra self-management features cover the full spectrum of self-management functionalities, including self-configuration, self-adaptation, self-protection, and self-optimization. The self-management of Hydra follows a three layer model proposed by Kramer and Magee [4] as detailed in Figure 1, where the interaction between different layers are through events via the Hydra Event Manager, following a publish-subscribe [5] communication style.

Besides the Event Manager, the Self-management component also needs to collaborate with other Hydra components, including the EventProcessingEngine component for Complex Event Processing (CEP), which is used to monitor dynamic resources and other context changes, and a QoS manager, which is used to retrieve the QoS properties for services/devices and monitor QoS changes.

[1] `http://www.hydramiddleware.eu`

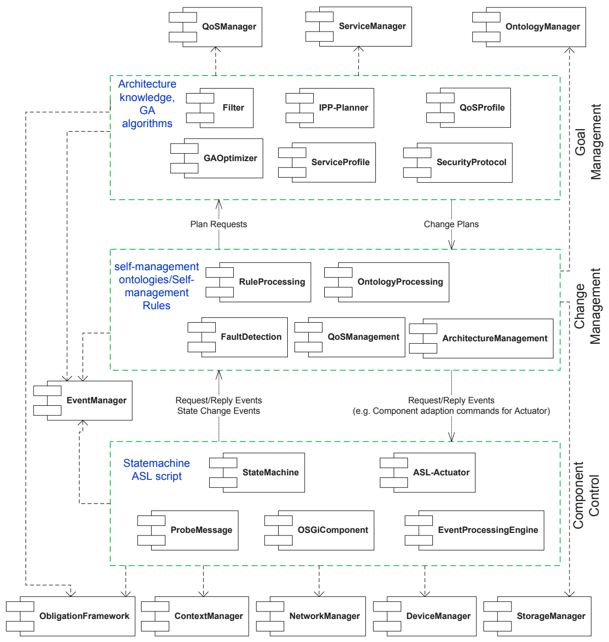

Fig. 1. Architecture of the Self-Management component in Hydra

Also, as we are adopting a Semantic Web-based self-management approach [6], the management of OWL[2](Web Ontology Language)/SWRL[3] (Semantic Web Rule Language) ontologies is handled by the Ontology Manager. The diagnosis results are stored via the Storage Manager for future analysis.

Component Control Layer. The component control layer has two responsibilities: to enable higher layers to monitor low-level events, and to actuate changes to the underlying system.

For detecting situations which require changes of the system's configuration, three components are available: device run time state monitoring via state machines, service invocations monitoring using message probes and detecting specific patterns of events through an event processing engine. Event patterns can be ordered in an increasingly abstract hierarchy, ranging from low-level events (e.g., raw sensor data) to high-level events (e.g., more complex situations like "fire in the hall"). In the Component Control layer, the EventProcessingEngine based on Complex Event Processing (CEP) is used to detect situations requiring changes of the system's security configuration, such as additional services or devices joining a network.

For the second purpose of the Component Control layer, the ability to actuate changes to a system's configuration by obligation policies triggering the execution

[2] http://www.w3.org/TR/swbp-n-aryRelations/

[3] http://www.w3.org/Submission/SWRL/

of ASL (architectural scripting language) [7] scripts is provided by its interpreter. This is shown as an *ASL-Actuator* component in Figure 1.

Change Management Layer. The Change Management layer is responsible for executing predefined schemes of self-management, i.e., this layer will respond to detected deficiencies in a system and execute strategies defined in advance or dynamically generated for a specific change event. A primary approach in Hydra is the usage of SWRL [6] to define these self-management capabilities. Further, QoS is considered if necessary for all self-management activities.

Goal Management Layer. Two complementary approaches are adopted in the Goal Management layer to achieve planning. First GAs are used for obtaining optimal solutions given some QoS goals and restrictions. Second, once a desired target solution has been chosen, it becomes input to the IPP planner [8] which generates an actuation plan.

A GA based approach [9] is used for optimization. Here, optimization (for example choosing the most suitable services for self-configuration) is one important task in self-management for pervasive service computing. These optimization tasks can be considered as problems of multi-objective services selection with constraints, where GAs are effective.

Non-trivial plans are generated with the IPP planner. Given a domain description, a start configuration and a condition describing the goal state, IPP planner can generate a sequence of actions available in the domain (architectural configurations in our case) that lead to a goal state.

2.2 Self-protection Work Flow

Figure 2 illustrates how the work flow of automatically re-configuring security settings in the middleware based on the components introduced above.

In the first step, situations are detected which might require a reconfiguration of security parameters and mechanism. For this purpose, events broadcasted on the event bus are being monitored and fed into the *EventProcessingEngine*, which then detects specific patterns of events. Once an event pattern has been detected (e.g. a new device with some additional managers joining the network), the EventProcessingEngine initiates the GAs to find the optimal configuration for the new situation.

In general, a number of steps is required to come from the current to the optimal solution identified by the GAs. Therefore, the optimal solution is at first sent to an *IPP planning engine* which calculates an enforceable plan leading to this solution. This execution plan is passed to an *obligation distribution point* (ODP) which is responsible for applying the individual steps of the plan to the system by sending appropriate obligations [10] to *enforcement points* (OEP). Obligations are signed by the Obligation Distribution Point (ODP) to prevent manipulation and to ensure authenticity of the obligation. When receiving an obligation, OEPs validate the attached signature and invoke appropriate enforcement plugins which are able to execute the actions stated within the obligation.

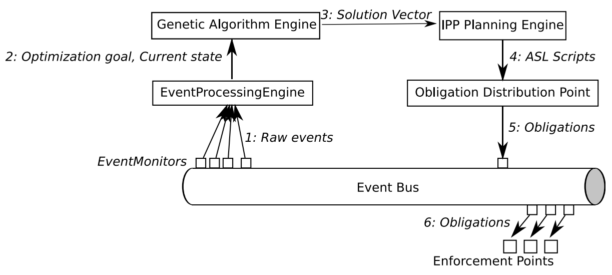

Fig. 2. Workflow of Self-Protection (components as boxes, communication as arrows)

After the enforcement process, OEPs can send back status report events indicating success or failure which can again be monitored by the component control layer.

From Figure 2, we can see that the proposed approach relies on two aspects: the underlying security contexts (implemented as ontologies) and an eventing mechanism for context provision. Therefore our approach is generic and is applicable to situations other than the Hydra middleware where the self-protection approach originated.

3 Security Ontologies

The Goal Management layer in Figure 1 requires information about security mechanisms that can be applied to the system to make proper decisions. This information is modeled in a set of security ontologies, which need to describe not only security mechanisms and their targeted protection goals, but also quality of those mechanisms which differentiate our security ontologies to the existing ones, such as the one from FIPA TC [11] and NRL ontology [12]. The ontology used in our approach is application-agnostic and provided as part of the middleware. Developers can add application-specific information by inserting additional instances into the predefined ontologies.

3.1 Modeling Protection Goals

For self-protection, the security ontology mainly serves two purposes: at first, it assigns each configuration to the protection goals it supports. Secondly, the ontology should provide information about the quality of a mechanism, i.e., describe how well it is suited to achieve the protection goals and how high the costs in terms of memory and CPU consumption will be. We will now describe the most important concepts of the ontology as depicted in Figure 3 and explain how they address those two purposes.

We model protecion goals as instances of the *SecurityObjective* class in order to describe which configuration of the system is suited to achieve a certain protection goal. This concept is modeled similarily to what is done in the NRL

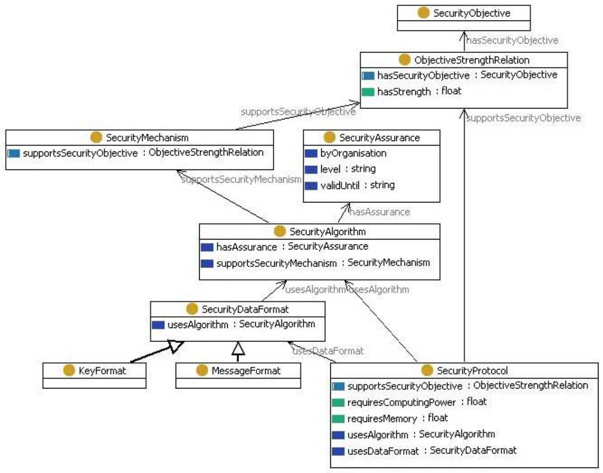
SecurityObjective
hasSecurityObjective
ObjectiveStrengthRelation
hasSecurityObjective : SecurityObjective
hasStrength : float
supportsSecurityObjective
supportsSecurityObjective
SecurityMechanism
supportsSecurityObjective : ObjectiveStrengthRelation
SecurityAssurance
byOrganisation
level : string
validUntil : string
supportsSecurityMechanism
hasAssurance
SecurityAlgorithm
hasAssurance : SecurityAssurance
supportsSecurityMechanism : SecurityMechanism
usesAlgorithm
usesAlgorithm
SecurityDataFormat
usesAlgorithm : SecurityAlgorithm
usesDataFormat
KeyFormat
MessageFormat
SecurityProtocol
supportsSecurityObjective : ObjectiveStrengthRelation
requiresComputingPower : float
requiresMemory : float
usesAlgorithm : SecurityAlgorithm
usesDataFormat : SecurityDataFormat

Fig. 3. Main concepts and properties of the Security Ontology

ontology, i.e. it comprises instances such as *Confidentiality*, *MessageIntegrity*, *ReplayPrevention*, etc.. Further, the concept *SecurityProtocol* represents the actual configuration that could be applied to the system. This concept is the only one whose instances refer to specific software modules or settings (e.g., OSGi[4] bundles or sets of preferences). As not all instances of *SecurityProtocol* are equally suited to fulfil a certain protection goal, we modeled an n-ary relation between *SecurityProtocol* and *SecurityObjective* using the *ObjectiveStrengthRelation* concept to overcome the lack of n-ary relations in OWL. In this way, we are able to express qualified relations using security levels like "RSA-512 serves *low* confidentiality". By querying the ontology given protection goals it is thus possible to retrieve a set of applicable implementations and configurations, ranked by the degree to which they address protection goals.

3.2 Modeling Resource Consumption

In most cases, security is not for free and so the second purpose of the security ontology is to provide information about the trade-off between the security level and the required performance costs for each instance of the *SecurityProtocols*. The resource consumption of each instance is represented by the properties *requiresComputingPower* and *requiresMemory*. Obviously, both properties vary depending on the platform and various other factors, so the values in the ontology may only be taken as a rough estimation. However, for our optimization approach the absolute values are not of interest but rather the relation of modules according to their resource consumption. Hence, we argue that in this case it is feasible to represent such platform-specific information in a system-wide

[4] http://www.osgi.org/

security ontology. The *requiresComputingPower* property describes the additional processing time that is required by adding a certain security module or configuration. That is, the values refer not only to cryptographic operations but to the overall processing time required by the module. The *requiresMemory* property describes the additional memory overhead that is added by applying a security module. It refers to the sum of memory allocated by all objects and methods of the module.

3.3 Usage of Security Ontologies

In Hydra, we are using SWRL rules to retrieve information from the security ontology. For example, the following rule is used to retrieve the security protocols and their corresponding memory consumption, computing time consumption, authenticity level and its value. This information is then used in the fitness evaluation functions described in Section 5.1.

Rule: SecurityResource
$SecurityProtocol(?protocol) \wedge$
$requiresComputingPower(?protocol, ?power) \wedge$
$requiresMemory(?protocol, ?memory) \wedge$
$authenticityObj(?protocol, ?auth) \wedge$
$hasStrength(?auth, ?value)$
$\rightarrow sqwrl : select(?protocol, ?memory, ?power, ?auth, ?value)$

Further, the security ontology is needed to automatically replace security mechanisms once they are considered to be insecure. From time to time, new attacks on cryptographic algorithms become feasible and their level of security decreases. Reflecting such changes in the security ontology by modifying the *ObjectiveStrengthRelation* (c.f. the following section) will trigger a re-configuration of the middleware, replacing outdated mechanisms by more secure equivalents. This work is still under investigation and will be reported in the near future.

4 Security Strategies and a Scenario for Self-protection in Hydra

In this section, we will describe how different security strategies described by the security ontology have been combined with the self-management architecture in order to realize self-protection in the Hydra middleware.

4.1 Security Strategies

A *Hydra device* is basically a set of *managers* (i.e. web services) which can either be hosted locally on a single platform or be distributed across devices. To protect communication between those managers (which we refer to as *Core Hydra*) a number of security modules with different properties are available. Besides the Core Hydra configuration, further security settings can be made in the middleware: the communication between Hydra devices can be protected in different ways, different trust models (e.g. OpenPGP, PKI-Common, etc.) can be used, and message formats such as XMLSecurity or S/Mime can be chosen. In this paper, however, we will focus on the Core Hydra configuration only, i.e. the selection of different security strategies for the communication between managers (the procedure for other configurations is analogous).

The protection of Core Hydra communication is realized by SOAP[5] security handlers implementing the following security strategies: *Null*, *XMLEnc*, *XMLEncSig* and *XMLEncSigSproadic* each representing a different protection level. These security handlers are hooked into the *web service handler chain*, a series of Java classes that is called immediately before a SOAP call is routed into the actual web service and immediately after the response leaves it. Thus, these Core Hydra handlers are supposed to be completely invisible for users of the middleware.

Null. This strategy switches off all message protection mechanisms and the Core Hydra security handler simply passes all messages on to the receiving manager. This strategy is obviously the most insecure but also the fastest way of sending messages in Core Hydra.

XMLEnc. This strategy applies XMLEncryption[6] to messages in Core Hydra. The message payload is encrypted using a 192 bit TripleDES key. This symmetric key is then attached to the message, encrypted by RSA 1.5 using the 1024 bit public key of the receiving manager. This strategy ensures confidentiality but does not fully prevent message modification or replay attacks.

XMLEncSigSporadic. For this strategy, nonces ("*n*umber used *once*") are added to messages in order to prevent replay attacks and XMLSignature[7] using RSA is applied in addition to XMLEncryption. Receivers will however only randomly verify a certain percentage of the arriving messages to save resources. While this strategy may allow attackers to send some individual forged messages, it is not possible to inject a whole sequence of faked messages. It depends on the messages content and the application whether this strategy adds any additional security – in the worst case it is equivalent to *XMLEnc*, in the best case it is equivalent to *XMLEncSig*.

XMLEncSig. For this strategy, messages are created in the same way as in the previous strategy. In addition, all signatures are verified by the receiver. So, the *XMLEncSig* strategy ensures confidentiality and authenticity as well as it prevents attackers from re-playing previously recorded messages.

Table 1 lists the security strategies with the degree of support for confidentiality and authenticity as well as their resource consumption, which are encoded in the security ontologies and will be used at run time as security contexts. For XMLEncSigSporadic, 50% of the arriving messages are verified in our case. The CPU processing time and memory consumption values have been obtained

[5] http://www.w3.org/TR/soap/
[6] http://www.w3.org/TR/xmlenc-core/
[7] http://www.w3.org/TR/xmldsig-core/

Table 1. Protection levels (0 to 10) and resource consumptions of security strategies

	Level of protection		Resource consumption	
Strategy	Confidentiality	Authenticity	CPU (ms)	Memory (KB)
Null	0	0	16.3	0.32
XMLEnc	4	4	21.4	28.96
XMLEncSigSporadic	4	7	102.4	54.97
XMLEncSig	4	9	114.3	57.52

by measuring the Hydra middleware with different security configurations on a VMWare Windows XP with 512 MB memory and an Intel Core2 Duo processor.

4.2 A Self-protection Scenario in Hydra

The Hydra middleware has been developed to interconnect heterogeneous embedded devices. In such scenarios developers have to deal with resource-constrained platforms and the performance versus security trade-off. Usually this requires design decisions to be made at development time and knowledgeable developers who know the benefits and deficits of different security mechanisms. The aim of self-protection is to relieve developers from this task as much as possible by automatically adapting security mechanisms to the current situation. As an example, we look at how the middleware automatically selects the security strategies that best fit the resource and security requirements of the application.

Suppose Hydra is the supporting middleware for an airport management system, a public area that needs high security. All of 10 different Hydra components are deployed on different devices: PDAs, PCs, and security checking machines, connected via the Internet. All data sent between the managers should be confidential, and – if possible – protected against modification and replay attacks. At the same time, resource constraints must be considered, i.e., the latency and memory consumption should not exceed limits. As there are 10 managers, there are $\binom{10}{2} = 45$ bi-directional connections/channels to consider. For each connection, three different security strategies are available (omitting the *Null* strategy as it does not provide any confidentiality). The problem space for finding the optimal solution is 3^{45}, a scale that works well for GAs. Therefore, the following goals for the overall system's security configuration (referring to all 45 channels) are passed as input to the Hydra Goal Management layer:

– *Authenticity* should be maximized (highest value is 10 for a channel)
– *Latency* must not exceed 2000 ms
– *Memory* should be minimized, not more than 2 Mbytes should be used

In the following section we will describe how the self-protection architecture finds an optimal solution to this problem, plans its execution and finally enforces all necessary steps.

5 Obtaining Optimized Protection Using GAs

First, we will formulate the abstract requirements as an optimization problem that can be solved using a GA engine.

5.1 Optimization Objectives and Constraints Formulation

The memory consumption of a Hydra device's security mechanisms (the M objective) is calculated by the sum of each channel's memory consumption as: $M = \sum_{i=1}^{n}\sum_{j=1}^{m} M_i \cdot E(i,j)$, where $E(i,j) = 1$ if for a channel i (with a scope of $[1,n]$) a security strategy that has memory consumption M_i is selected, otherwise $E(i,j) = 0$. j represents the sequence number of a concrete security strategy with a scope of $[1,m]$. In the scenario under consideration, $n = 45$ and $m = 3$. As we choose exactly one security strategy for each channel, there is exactly one $E(i,j) = 1$ and all other $E(i,j) = 0$ for all $j \in [1,m]$.

Similarly, we can formulate the CPU consumption (the P objective) to calculate the total processing time required by security mechanisms as: $P = \sum_{i=1}^{n}\sum_{j=1}^{m} P_i \cdot E(i,j)$, where $E(i,j) = 1$ if a component i (with a scope of $[1,n]$) that has power consumption P_i is selected, otherwise $E(i,j) = 0$. j represents the sequence number of a concrete component implementation with a scope of $[1,m]$.

Authenticity, as said, should be maximized. We instead minimize the un-authenticity to formulate all objectives in a similar way. The un-authenticity (the Ua objective) is calculated as: $Ua = n \cdot 10 - \sum_{i=1}^{n}\sum_{j=1}^{m} A_i \cdot E(i,j)$, where $E(i,j) = 1$ if a channel i (with a scope of $[1,n]$) that has authenticity A_i is selected, otherwise $E(i,j) = 0$. j represents the sequence number of a concrete security strategy with a scope of $[1,m]$.

5.2 Chromosome Encoding and Fitness Evaluations

A chromosome corresponds to a unique solution in the solution space. GAs can typically make use of booleans, real numbers and integers to encode a chromosome. The representation of chromosome in our case is using integers (starting from 0). That is to say, we are using an integer vector $V = [V_1, V_2, ...V_i, ..., V_n]$ (where n is the number of decision variables – in our case 45) to represent a solution. V_i is a natural number, acts as a pointer to the index of the security strategy of the ith strategy. For example, a chromosome [0,1,2,1,2,0,1,1,2,1...] represents that a solution chooses the first security strategy for channel 1, the second security strategy for channel 2, the third security strategy for channel 3, and so on. In our case, this relates to *XMLEnc*, *XMLEncSigSporadic*, *XMLEncSig* (cf. Table 1). Based on the chosen security strategies, the GAs then decide fitness using the objective equations as introduced in Section 5.1, and will at the same time evaluate whether the constraints mentioned in Section 5.1 are met.

6 Prototype Implementation

In order to test the self-protection approach, we developed a prototype that has been integrated into the Hydra middleware. In this section we will discuss the architecture of the prototype implementation and the achieved performance.

6.1 Implementing GA-Based Optimization for Self-protection

As in our former evaluation of GAs for self-management [9], we used the JMetal GA framework[8] for the implementation of the self-protection optimization problem. As shown in Figure 4, we model a SelfProtectionProblem as a SelfManagementProblem. Evaluations of solution fitness using the formulas introduced in Section 5.1 are implemented in the SelfProtectionProblem class as usual when a developer is to implement self-management optimization problems. The GAEngine is the core class for the GA-based self-management planning, and defines the common methods for getting the solutions.

The package *evaluations* defines utility classes for obtaining the Pareto front/set[9], and the evaluation of the solution quality uses the Hyper volume (HV) quality indicator [13], which is a quality indicator that calculates the volume (in the objective space) covered by members of a non-dominated set of solutions for problems where all objectives are to be minimized.

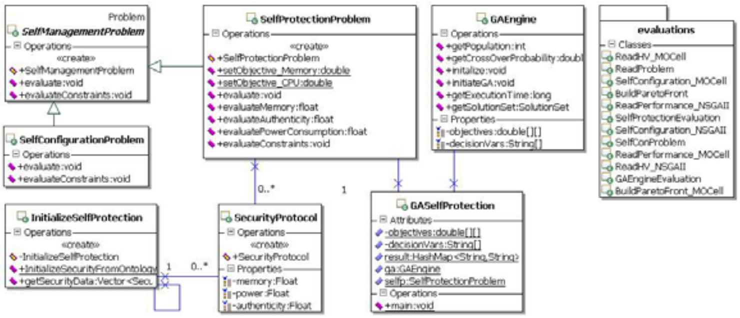

Fig. 4. GAs based Self-management optimization

6.2 Enforcement of Obligations

The enforcement architecture (c.f. Section 2.2) allows adding support for arbitrary obligations at runtime by loading appropriate enforcement plugins. We implemented one enforcement plugin that supports operations on the OSGi platform (such as starting and stopping bundles or setting preferences) and one that

[8] http://sourceforge.net/projects/jmetal/
[9] http://www-new.mcs.anl.gov/otc/Guide/OptWeb/multiobj/

supports the execution of ASL scripts. While for simple obligations such as used in our prototype example, the OSGi plugin provides a fast and direct access to OSGi management, platform-independent ASL scripts are better suited for heterogeneous platforms and more complex architectural restructurings [7]. The sequences of actions that constitute an obligation policy (and an ASL script) is generated by the IPP planner based on the target security configuration found by the GA optimization.

6.3 Performance Measurements and Quality Evaluation

Performance of Genetic Algorithms. For the measurement of performance of obtaining optimal solutions, the following software platform was used: JMetal 2.1, JVM 1.6.02-b06, Heap memory size is 256 Mbytes, Windows XP SP3. The hardware platform was: Thinkpad T61P T7500 2.2G CPU, 7200rpm 100G hard disk, 2G DDR2 RAM. The performance time measurements are in milliseconds.

We have done evaluations of two generic algorithms, NSGA-II and MOCell for their usage in pervasive computing [9]. In this paper, we want to validate whether our recommendations for these two algorithms are valid for different problem (where the problem space is much bigger and fitness evaluation algorithms are different). This time, the parameter settings for GAs are the same as in [9], and we are following the same steps as in [9] for evaluations.

The analysis for this evaluation (procedures as detailed in [9]) shows that our recommendations for parameter settings as in [9] are valid and NSGA-II is recommended for our self-management problems. Table 2 shows randomly chosen runs (from one of 100 runs for every parameter combination) for some of the parameter combinations (as detailed in the legend of Figure 5). We can see that for NSGA-II, which is recommended (and was recommended in [9]) in this case, the population size 64 to 100 with max evaluations of 5000 will have acceptable performance for getting optimized solutions within 342ms to 449ms, and has acceptable quality of solutions as shown in Table 2 and Figure 5. MOcell is not recommended as it has worse HV. We can see this in a direct way in Figure 5: MOCell solutions has many more points far from the Pareto front. We can also see that the *diversity* and *convergence* are satisfactory of NSGA-II, the solutions are spread uniformly along the true Pareto front, and the majority of the points in NSGA-II results are located at the Pareto front.

Table 2. Performance and quality of solutions

GA name	Population size	Max evaluations	cross over probability (CVP)	Avg. HV	Avg. Running Time
NSGA-II	64	5000	0.8	0.566524	342 ms
NSGA-II	81	5000	0.9	0.566524	419 ms
NSGA-II	100	5000	0.9	0.566524	449 ms
MOCell	1444	5000	0.8	0.459411	235 ms
MOCell	1600	10000	0.8	0.494775	576 ms

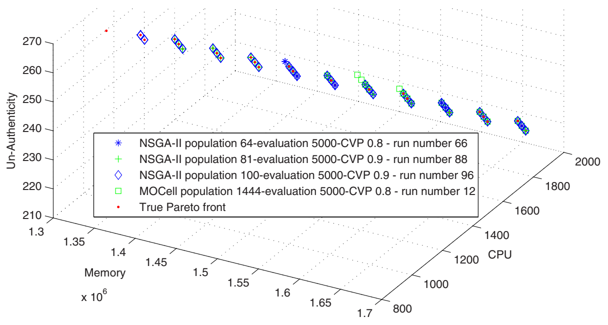

Fig. 5. Visualizing the solution quality

Performance of IPP Planner. We measured the performance of the IPP planner for the plans required in our implementation. With just one security strategy to be set, the planner generates the correct solution in just 10ms (average of 5 measurements, standard deviation 2 ms). In our case, at most four kinds of planning problems can occur, because the steps required to change a strategy depends only on the strategy being activated. Thus in practise the planner can be invoked once for each of these problems to produce a template plan/scheme which is stored in the Change Management layer and available for immediate execution once needed. Thus the test showing an execution time of just 10 ms is the worst case time for planning in our implementation. Other implementations of our approach may require more complex plans to activate a strategy. However, our previous experience with using the IPP planner for general architectural reconfiguration shows that it generates a plan within 100 ms [7].

Performance of Obligation Enforcement. Finally, we measured the performance of the enforcement process, i.e. the process of distributing a single obligation to the OEPs and executing the contained actions. The overall time (omitting network latency) amounts to 70.9 ms (standard deviation 14.21 ms) whereas the plain execution time is almost negligible (0.6%) due to the simple operation we use in our prototype example (changing the configuration of the Core Hydra module). The main computing costs come from signing and verifying the obligation, accounting to over 73% of the overall enforcement time. Another 21.7% is required by Axis 1.4 web service calls.

6.4 Discussion

The critical part of our self-management approach is obtaining the optimized solutions for all the communication channels. The search for the best solutions should be finished in a reasonable time. As we can see from Section 6.3, GAs can accomplish this within acceptable time and satisfactory quality. Combining the performance testing with IPP Planner for generating enforcement plans, and the performance of actual enforcement of security protocols from Section 6.3, in the best case we can get the self-protection ready within 520ms, which is acceptable for enabling the self-protection for the whole Hydra middleware. Even in the “worst” case, where the IPP planner needs to be invoked and the enactment of a strategy change is more complex than in Hydra, this would add less than 100ms or 20% to the execution time.

7 Related Work

In the Willow architecture [14] for comprehensive survivability, security threats are treated as one source of faults that the architecture provides mechanisms to avoid, eliminate, or tolerate. In contrast with our prototype, there is no dynamic adaptation or explicit modeling of the trade-offs involved in providing the protection. The ATNAC framework described by Ryutov et al. [15] detects malicious activity by analyzing failures and behavior patterns of the access control and trust negotiation process. Thus rather than trying to prevent an ongoing attack as such, a detected malicious activity is input to the access control and authorization process which thereby becomes more dynamic. The functionality is at the specific level orthogonal to our work, in that it is concerned with authentication. Further, the adaptation which is provided is focused on improving the accuracy of authentication, rather than on balancing multiple concerns against each other as in our approach. Another approach to multi-objective optimization is followed by the middleware CARISMA. In [16], the authors propose utility functions and auction-based negotiations to agree on an optimized trade-off between security and efficiency. Their decentralized approach however assumes each instance of the middleware acts honestly.

Event-condition-action policies as used in our obligation framework have been used for many policy-based management approaches before, where Ponder2 [17] is one of the most prominent examples. However, self-protection is scarcely considered in such approaches. Finally, a conceptually different approach to self-protection is used in artificial immune systems [18]. This approach is interesting but it is unclear yet how it can be combined with other self-* approaches in order to make acceptable tradeoffs between several different qualitative concerns. In our approach, multi-objective optimization can be used for other self-management features, as we have done for self-configuration [9].

8 Conclusion and Future Work

Self-protection is one of the important self-management capabilities of pervasive service computing. There is scarce reported work providing optimized self-protection, i.e. considering the characteristics of pervasive systems where resources are usually restricted. In this paper, we proposed a Genetic Algorithms-based approach for obtaining optimized security configurations. The optimized solutions can be used to enable corresponding security strategies, based on obligations generated from the IPP planner, and finally the obligation framework will execute these plans and make use of the chosen security protocols. The whole process is evaluated and it was show that our approach is feasible with acceptable performance and satisfactory quality. We will explore auction-based multi-attribute optimization [16], and investigate the replacement of outdated security mechanisms at run time using security ontologies.

Acknowledgments. The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

References

1. Elkhodary, A., Whittle, J.: A survey of approaches to adaptive application security. In: Proc. of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, Washington, DC, USA. IEEE C.S, Los Alamitos (2007)
2. Mitchell, M.: An Introduction to Genetic Algorithms. Bradford Books (1996)
3. Whitten, A., Tygar, J.D.: Why johnny can't encrypt: A usability evaluation of pgp 5.0. In: Proceedings of the 8th USENIX Security Symposium (August 1999)
4. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: International Conference on Software Engineering, pp. 259–268 (2007)
5. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The Many Faces of Publish/Subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
6. Zhang, W., Hansen, K.M.: Semantic web based self-management for a pervasive service middleware. In: Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008), Venice, Italy, October 2008, pp. 245–254 (2008)
7. Ingstrup, M., Hansen, K.M.: Modeling architectural change - architectural scripting and its applications to reconfiguration. In: WICSA/ECSA 2009, Cambridge, England, September 2009. IEEE, Los Alamitos (2009)
8. Koehler, J., Nebel, B., Hoffmann, J., Dimopoulos, Y.: Extending planning graphs to an adl subset. In: Steel, S. (ed.) ECP 1997. LNCS, vol. 1348, pp. 273–285. Springer, Heidelberg (1997)
9. Zhang, W., Hansen, K.: An Evaluation of the NSGA-II and MOCell Genetic Algorithms for Self-management Planning in a Pervasive Service Middleware. In: 14th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2009), pp. 192–201. IEEE Computer Society, Washington (2009)
10. Pretschner, A., Hilty, M., Basin, D.: Distributed usage control. Communications of the ACM 49(9), 39–44 (2006)
11. FIPA Security: Harmonising heterogeneous security models using an ontological approach. Part of deliverable Agentcities. RTD, Deliverable D3.4 (2003)
12. Naval Research Lab: NRL Security Ontology (2007), `http://chacs.nrl.navy.mil/projects/4SEA/ontology.html`
13. Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. IEEE transactions on Evolutionary Computation 3(4), 257–271 (1999)
14. Knight, J., Heimbigner, D., Wolf, A.L., Carzaniga, A., et al.: The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications, Technical Report CU-CS-926-01, University of Colorado

15. Ryutov, T., Zhou, L., Neuman, C., Leithead, T., Seamons, K.E.: Adaptive trust negotiation and access control. In: SACMAT 2005: Proceedings of the tenth ACM symposium on Access control models and technologies, pp. 139–146. ACM, New York (2005)
16. Capra, L., Emmerich, W., Mascolo, C.: CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. IEEE Transactions on Software Engineering, 929–945 (2003)
17. Twidle, K., Dulay, N., Lupu, E., Sloman, M.: Ponder2: A policy system for autonomous pervasive environments. In: The Fifth International Conference on Autonomic and Autonomous Systems (ICAS) (April 2009)
18. Dasgupta, D.: Advances in artificial immune systems. IEEE Computational Intelligence Magazine 1(4), 40–49 (2006)