Trust as differentiator for value-adding home service providers

W. Haerick, J. Nelis, D. Verslype, C. Develder, F. De Turck, B. Dhoedt
Ghent University - IBBT,
Dept. of Information Technology - IBCN, Ghent, Belgium
Email: wouter.haerick@intec.ugent.be

***Abstract*—The openness of OSGi home service delivery platforms enables a service provider to deploy new services which aggregate services from other providers. Without security measures, each service provider is able to probe the home network, modify the configurations of other services, and (ab)use these services in favor of their own services. Premium service providers however prefer to protect their value-added services in an attempt to differentiate with low-cost or free Internet-based service providers. The common home technologies however lack fine-grained security support and do not allow to configure trust-based service adaptation. In this paper, we propose an intelligent residential gateway, with three security components that facilitate trust-based service adaptation in a multi provider environment. Using XACML policies, the collaboration between service components can remotely be modified. Legacy services can be protected and collaboration of services can be made dependent on the outcome of any other service. We compare OSGi virtualisation, embedded OSGi security and security-as-a-service. The latter allows for fine-grained access control on method-level. In a proof-of-concept implementation, we evaluate the performance overhead of transparent service authentication and three policy administration approaches. The results illustrate a minimal overhead to add trust-based service collaboration to any (legacy) service.**

***Keywords*-service collaboration, trust, UPnP, OSGi**

I. INTRODUCTION

In recent studies, promising home service delivery platforms are presented [1], [2], [3], [4] that are based on OSGi [5] and UPnP [6], and allow service providers to remotely deploy their services [7]. Considering a multi-provider context where different service providers–located in the WAN or regional access network–have access to a single home service platform, one service provider could use services from other service providers to build enhanced services (Figure 1). As an example, a SIP proxy provider could interact with a Video-on-Demand service to pause a movie while a SIP call is being established. Similarly, the SIP proxy provider could also verify the current delay in the WiFi network and adapt the encoding parameters accordingly. In the latter case, a SIP proxy service needs to request information at a WiFi-QoS service that might be a premium service, deployed and operated by an Internet access provider. By protecting the premium services, and restricting access to trusted service providers only, premium service providers are able to build aggregated services with best-of-class quality of service. Hence, premium providers are able to differentiate using trust-based access control.

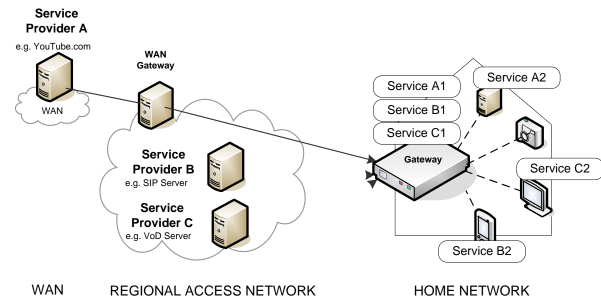

Figure 1. Multi-service provider service delivery architecture

The next-generation service platforms [1], [2], [4] use the UPnP protocol to discover and manage distributed home devices. Locally running services can be discovered by querying the registry of the OSGi service platform. Both technologies however lack support for fine-grained access control and remote configuration of trust relations:

- The permission model of OSGi is a too coarse-grained access control mechanism that is not able to express access control on method-level. The conditional permission concept, which consists of (condition, permission) pairs, does not allow to combine conditions, and as a result is not able to support adaptive service collaboration policy rules.
- The policy language specified in the UPnP Security architecture [8], is a custom language that could be used to protect a UPnP device from being used by an untrusted control point, e.g. a control point from an unknown service provider or remote management provider. The basic language is however not suited for use as generic policy language inside the OSGi service platform: Although it supports access control on method and parameter level, it lacks support for variables and is not able to combine policies that target UPnP devices with policies that target an OSGi service that is not a UPnP service. Trust management is limited to maintaining an ACL list per UPnP device, which includes the names of trusted control points.

This paper therefore contributes to the domain of trust-based service adaptation by comparing three architectures

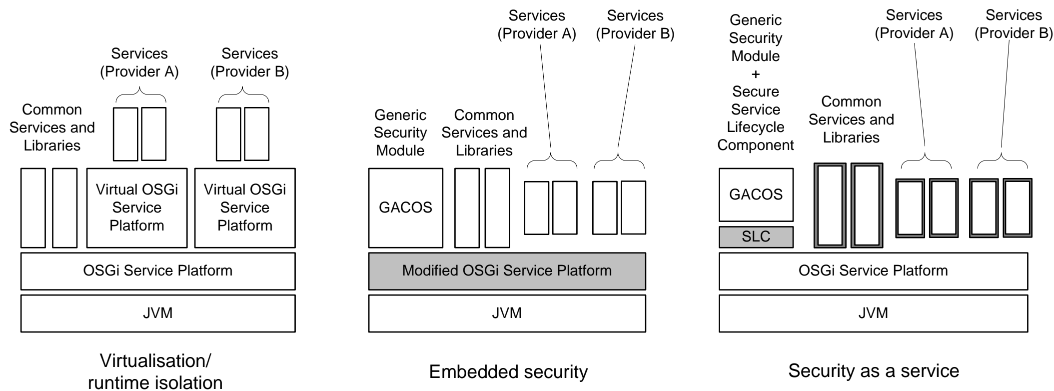

Figure 2. Three architecture to realize trusted service environment

to realize trusted service environments, and specifying the design of an user- and context-aware residential gateway. We have built a proof-of-concept implementation to illustrate the performance overhead for trust-based service adaptation. A remote configuration agent enables premium service providers to update the trust relations and service collaboration policies.

This paper is structured as following: The next section compares three gateway architectures for trust-based collaboration of OSGi services. Section 3 describes the subcomponents of the trust-aware residential gateway, and illustrates the use for three key scenarios: transparent service authentication, trust-based service adaptation and protecting legacy services. Subsequently, we address the management aspects to remotely establish trust and configure collaboration policies. In section 5, we evaluate the performance overhead of transparent authentication versus password or X.509 authentication, and compare the performance of three policy administration approaches. In a last section, we summarize the benefits of the presented security components for premium service providers.

II. Trusted service collaboration architectures

In this section we compare three Java-based architectures to create trusted environments where services can securely interact with each other: OSGi virtualisation, embedded OSGi security and security as an (OSGi) service.

A. *OSGi virtualisation*

A first approach of adding security to an OSGi service platform, is OSGi virtualisation [1]. As services deployed inside the home gateway may contain business-critical code or data, or may give away information on the client base, the authors motivate to isolate service bundles from different service providers. The presented approach is to embed the OSGi framework as a Java service into a core OSGi framework (Figure 2 (left)). The core OSGi instance runs multiple virtual OSGi environments, all running within the same Java Virtual Machine. This virtualisation gives each service provider a separate execution environment. The architecture specifies a separate installation manager per virtual gateway but does not define a mechanism to verify the origin of services at install time.

A controlled way is provided to share common libraries and services, such as a web service or a generic logger. These common services or libraries are hosted by the core OSGi platform, and explicitly exported to all virtual home gateways. By isolating services from a (group of) service provider(s), services are protected of being used by non-trusted services running in the home gateway.

B. *Embedded security*

A second approach to restrict access to premium services, consists of modifying the core OSGi framework (Figure 2 (middle)). As a consequence, this security solution can not be added as a separate bundle by a remote management server. The changes must be applied in the core OSGi framework and therefore should replace the existing OSGi implementation code. As a direct consequence of this integration into the framework, it has the advantage that the security solution can not easily be replaced by another bundle at runtime.

By intercepting all requests for a service object, access permissions can be verified. Two OSGi objects require modifications to implement restricted access to a service: BundleContext and Bundle. The BundleContext object has the global overview on the OSGi registry and thus on all the locally registered bundles and listeners. The Bundle interface defines the actual lifecycle controlling methods: starting, stopping, updating and uninstalling a service. Compliant with release 4 of the OSGi specifications, the origin of the

Criteria	OSGi Virtualization	Embedded security	Security as a service
SERVICE COLLABORATION POLICIES			
Any service with any service	-	Inflexible	✓
Trusted environment	✓	✓	✓
GRANULARITY ACCESS CONTROL			
Method level	-	-	✓
Service level	-	✓	✓
Service provider	✓	✓	✓
TRANSPARENT SOLUTION			
Supporting legacy services	✓	✓	Transparent for service developer
OSGi implementation agnostic	✓	-	✓

Table I
COMPARISON OF TRUST-BASED SERVICE ARCHITECTURES

calling bundle can be verified against the bundle signature or the bundle location.

In [9] a similar architecture is proposed that filters the view a service has on the OSGi service registry. The filtered view is realized by sending UNREGISTERED events to those services that are not allowed to use the service. A service that receives such an event, needs to release the service, in case it was already using it.

C. Security-as-a-service

In contrast with the previous two approaches, the third alternative allows to apply flexible, method-level policy enforcement to any legacy service without requiring changes to the core OSGi framework. Restricted, trust-based service access control is achieved by wrapping all premium services (Figure 2 (right)). The wrapper intercepts each service method call, and evaluates the requestor's identity and applicable XACML policies. Four security-related services, deployed in the residential gateway, are involved in the configuration and enforcement of trust-based service collaboration: the secure Service LifeCycle (SLC) component, the User Awareness (UA) component, the Generic Access COntrol Service (GACOS) component and the Multi Service Provider trust manager. These components are discussed in the next sections.

Table I compares the three architectural options as a function of the ability to execute service collaboration policies, the granularity of the access control mechanism and the transparency with respect to legacy services and current OSGi implementations.

The OSGi virtualisation approach follows an isolation approach that hinders service collaboration across virtual gateways. To deploy services that require the interaction of multiple other services, all these service providers should be deployed in the same virtual gateway. The latter compromises the isolation approach. The embedded security solution, which extends the OSGi permission model, allows to define policies that evaluate custom conditions. However, as no policies can be combined and no variables are supported, a new custom condition need to be defined and programmed for each new collaboration scenario, making it a solution that is not practical. The third architecture, which enforces policies on method level, is suited to define collaboration policies that require other services to be executed before a service is accessed.

With respect to access control granularity, all three solutions allow to restrict access to premium services. In the first and second architecture, a service only discover the subset of trusted services. The third architecture allows service providers to discover all services inside the service platform. However, stringent policies control if a service can be invoked or not by another service. This third architecture also enables a gateway manager to enforce policies on the level of individual methods of a service, taking into consideration the parameters of the method call and other contextual information.

From a deployment perspective, a security solution is required that runs on current OSGi implementations and which allows to protect legacy services without the need to modify these services. The OSGi virtualisation approach meets both requirements. Legacy services need to be reinstalled inside a virtual gateway instead of inside the core gateway. These services might require common services from the core gateway to be imported into the virtual gateway. The second architecture requires to 'hack' an OSGi implementation to embed the security measures, and as a consequence is not OSGi implementation agnostic. Only if the proposed security functionality would become a mandatory part of the OSGi specifications, this architecture is worth considering. The third architecture also supports legacy services and can be added to any OSGi implementation. However, the legacy services require a modification at install time. The service lifecycle component needs to add a policy enforcement function to each of the service methods. As a result, the third architecture is transparent for service developers. However, an automated pre-install process needs to be executed before a service is protected.

III. DESIGNED ARCHITECTURE DETAILS

Given the flexible, fine-granular policy administration and the support for trust-based service collaboration, we have adopted the third architecture–providing security as a service–to enhance OSGi based home service platforms. In this section, we elaborate on the requirements and design of a proof-of-concept implementation.

A. Requirements

We target a security service that on itself can be offered as a premium service. The service should allow any other premium service to be protected against access of non-premium services, to participate in trust-based collaboration scenarios and allow for context-aware adaption. We assume

that all services are hosted on the residential gateway. The security service should comply to the following features:

- Realtime authentication and policy enforcement mechanism on service method level
- Secure access to context information on active, trusted users and shared (network) resources
- Flexible policy administration that allows define conditional service collaboration
- Trust management to configure which premium service providers are trusted

B. Architecture components

We present a modular architecture to realize trust-based service adaptation in a multi-provider environment. The modules can be plugged into the intelligent gateway architecture presented in [10]. As mentioned in section II, the security-as-a-service architecture consists of four security related sub-services: SLC, UA, GACOS and MSP.

1) The SLC component manages secure installation of a service. For each signed service, the following process applies: (1) Verify the identity of the signer against a store with certificates of trusted service providers, (2) add a policy to the policy store that uses local network information or trusted user credentials, (3) add a security wrapper to each of the methods of the service that allows to evaluate applicable policies before granting access to the service.
2) The UA component has a centralized view on all active home users, and offers the interface to administer credentials and policies. Interactions with this component allows trust-based user-awareness of services. The information about active users is retrieved from user management systems that authenticate users when they join a home network, or from IP packet inspection of data packets containing user information. Only services signed with a trusted certificate can query the User Awareness component. From the WAN-side, access to the TR-069 User Awareness Manager is limited to a pre-configured, trusted RMS.
The User Awareness component introduces the concept 'Trust Circle'. A trust circle groups users and devices with identical service permissions. Any user or device can belong to multiple trust circles, and by default a trust circle 'home user' and 'guest' is created. A signed service can find out if a user belongs to any of the (default) trust circles, and restrict functionality or adapt service behavior accordingly. Therefore the following methods are exposed by the User Awareness component:
   - *public boolean isTrustedUser(String user, String trustCircle);*
   - *public boolean isUserOnline(String user);*
   - *public String getIPOfUser(String user);*
   - *public String getUserOfIP(String ip);*
   - *public String getMACOfUser(String user);*
   - *public String getUserOfMAC(String mac);*
   - *public String getTrustCircleOfUser(String user);*

   By invoking these methods as part of the evaluation of a policy, user-aware service adaptation can be realised.
3) The GACOS component is the core security component that (1) performs authentication, (2) gathers contextual information on network, user preferences and user accounts, (3) retrieves matching policies and (4) evaluates if a calling party should get access to the targeted service. The GACOS implementation is based on SUN's libraries for XACML [11]. The evaluation of an access request involves the following components: A Policy Enforcement Point (PEP), an Access Control Service (ACService), a Policy Decision Point (PDP), a Policy Authentication Point (PAuthP), a Policy Information Point (PIP) and a Policy Administration Point (PAP). Figure 4 illustrates the component interactions. Each of the GACOS components has a well-defined responsibility:
   - PEP: guards access to a set of resources and asks the ACService to make an authorization decision considering the origin of the subject of the request.
   - ACService: is the discoverable service that acts

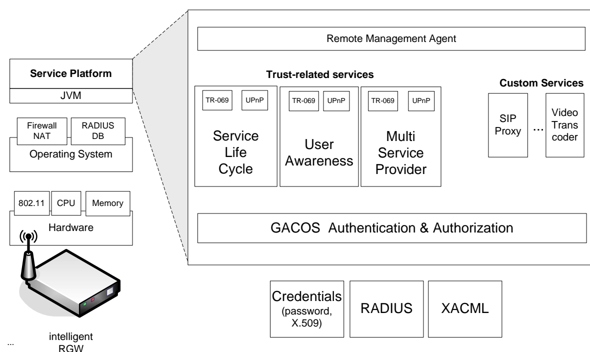

Figure 3. The intelligent gateway: Trust-related components provide awareness of active users, available services and available resources.

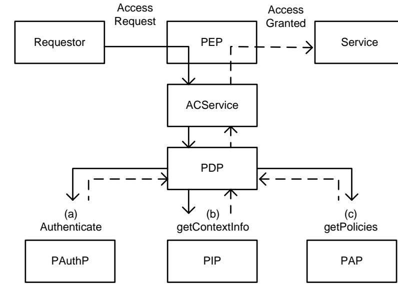

Figure 4. GACOS policy enforcement

as a facade to a PEP, and forwards the access requests to a PDP.
   - PDP: makes the authorization decision using information from PAuthP, PIP and PAP.
   - PAuthP: screens the credentials of the subject, finds a proper authentication module and evaluates the authenticity of the subject.
   - PIP: adds environment information to the XACML request upon request of the PDP.
   - PAP: manages the persistence of XACML policies.

   For the implementation of the PAuthP service, we use the Java Authentication and Authorization Service (JAAS) libraries that offer a framework for pluggable authentication modules (PAM). This framework offers a number of default login modules such as the LdapLoginModule and Krb5LoginModule to authenticate subjects respectively using LDAP and Kerberos. In the scope of a multi-service OSGi platform, we have added a BundleLocationLoginModule, a custom PasswordLoginModule and a X509KeyStoreLoginModule to authenticate respectively OSGi services and end-users or service providers. The BundleLocationLoginModule module verifies the installation location of a service against a list of trusted locations. For this verification, we use the BundleLocationPermission defined in release 4 of the OSGi specifications. The Password Loginmodule allows to verify (username, password) pairs, and the X509KeyStoreLoginModule verifies if the supplied X.509 certificate is signed with the private key of a CA known in the keystore.
4) The MSP component offers the interface for remote management of trust relations between premium service providers as well as for remote updates of service collaboration policies. The management aspects are discussed in the section IV.

C. Key scenarios

The aforementioned, security components enable a number of trust-related scenarios that are key for premium service providers. We elaborate on three scenarios and describe which components are involved.

- **Transparent service authentication:** To enforce access decisions based on the identity of the requesting service, the ACService needs to receive the credentials of that service from the PEP. However, this should not require service providers to change their method signatures to include these credentials as additional parameters in their service requests. As a consequence, the PEP itself should be able to transparently extract the identity of the calling service. This way of transparent service authentication is achieved by querying the stacktrace information to retrieve the class name of the calling bundle. From the class name, we query the bundle context to retrieve the signature from the bundle. With these data, a subject with a bundle credential can be created and passed to the ACService.
- **Trust-based service collaboration policies:** The GACOS component uses the XACML policy language to express trust-based policies. As an example, we consider a SIP Proxy service which only allows home users to access the PSTN network, thus blocking untrusted users. We therefore extended the XACML default functions with an external function that is able to invoke any OSGi service specified. Using this custom XACML function, named $use-osgi-service-with-argument$, we are able to express policies that interact with the User Awareness component. The XACML snipplet below illustrates how we configured a SIP proxy service to block non-trusted users calling to the PSTN network by invoking the method $getTrustCircleOfUser$ of the UA component.

```
<Rule RuleId="PSTNaccess" Effect="Permit">

<Description>Only home users are allowed to call
 to the PSTN network (block guests)</Description>

<Condition>
<Apply FunctionId="urn:oasis:function:string-equal">
<Apply FunctionId="use-osgi-service-with-argument">
<AttributeValue>services.user.UserAwarenessManager</>
<AttributeValue>UserAwarenessManager</AttributeValue>
<AttributeValue>getTrustCircleOfUser</AttributeValue>

<SubjectAttribute ... AttributeId="sip_subject-id">
</SubjectAttributeDesignator>
</Apply>
<AttributeValue>homeuser</AttributeValue>
</Apply>
</Condition>

</Rule>
```

  The XACML condition is expressed as a nested function: the outer function is a string equation and the inner function queries the User Awareness component. If the trust circle of the SIP user equals ‘home user’ then the effect of the policy is to permit the INVITE SIP message to be forwarded to the SIP-PSTN gateway. Similary, an XACML condition can also verify signatures against an X.509 certificate, or integrate with any other credential data store.
- **Support for legacy services:** We support protection of any legacy service through the AccessControlWrapper component that allows to add a Policy Enforcement Point to any OSGi service. The Service LifeCycle component, which manages the installation of a service into the service platform, is extended with a method that invokes the AccessControlWrapper module of the GACOS component. This wrapper generates a new service implementation that invokes ACService before every functional call to the original service implementation. This new service implementation is added to the Activator class of the service, which is at runtime

recompiled to add these minor changes. A reference to the new service implementation is added to the OSGi registry instead of a reference to the unprotected service implementation.

IV. Managing trust-based collaboration and adaptation

To participate in adaptive service collaboration scenarios, a policy enforcement wrapper is needed around each premium service, and policies and credentials need to be configured to define trust relations and service behavior. The Multi Service Provider (MSP) component covers this remote configuration of trusted providers and flexible policies.

We propose a three-step, remote configuration process: (1) Add a signed certificate of the involved service providers to a key store, (2) configure in a VisibilityMap which service providers are allowed to manage which service, (3) add trust-based method-level access restriction. The following methods, provided by the MSP component, are accessible from the WAN through the SOAP-based TR-069 management protocol:

- *addServiceProvider*: This method takes a signed X.509 certificate as input, and stores this certificate in the certificate store using the UA component of the gateway.
- *setExternalVisibility*: This method takes a service and a service provider as input, and updates the VisibilityMap to allow the service to be managed by the given service provider.
- *shareService*: This method takes as input parameters the certificate from two service providers (the service owner and the configuration provider), and the resource and an action to be protected. After verifying the signatures of the signed certificates, the origin of the resource is verified against the service provider that claims to be the service owner. Finally, the XACML policies are added to allow all services from the configuration provider, to access the given resource and action.
- *installService*: This method allows to install a protected service. During installation a PEP is added, the VisibilityMap is updated to allow the owning service provider to configure his service, and the policies are updated to allow the service to use the GACOS component.

Figure 5 depicts Service Provider A which configures a trust relation with another service provider. Firstly, service provider A exchanges its credentials during a two-way SSL handshake with the main TR-069 Management Agent that forwards the configuration messages to the TR-069 MSP component. A Policy Enforcement Point around the MSP component verifies if service provider A should get access to the User Awareness component that has a central view on all users, credentials and policies. In each of the three configuration steps, security-related data is persisted, respectively in a certificate keystore, the VisibilityMap–this map defines which service provider may configure which services–and an XACML policy store to hold the service behavior policies. As an example, service provider A may allow service provider B to configure and use the services from Service Provider A. The initial trust with Service Provider A should be manually configured by a home user, or could be added by the trusted internet access provider that operates the connectivity with the gateway.

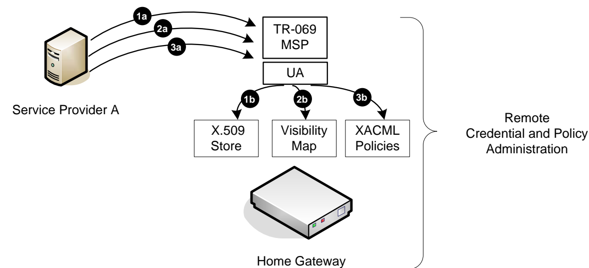

Figure 5. Three-step trust and policy configuration

Figure 6 illustrates the realtime trust verification and policy enforcement process, performed by the GACOS component. GACOS is involved in:

- remote configuration to verify if Service Provider B is allowed to access the Management Service of Service Provider (left side). The VisibilityMap is queried to identify trusted service providers.
- trust-based policy enforcement of service A3 trying to access service B2 (right side). Depending on the XACML policy added in the third step of the remote configuration, the GACOS component may invoke the User Awareness component, or any other (contextual) services, to verify if the requesting services is trusted. In case the context information does not match with the policy, access to service B2 is denied, or some of the service parameters are changed. This allows to configure a collaboration scenario where a service first verifies the delay in the home network, and subsequently adapts the voice codec to be used in a SIP conversation.

V. Performance evaluation

In section II, we have compared three alternative security architectures—OSGi virtualisation, OSGi embedded security and security-as-a-service—from a functional point of view. We motivated the selection of the third option which allows for flexible, fine-grained access control on method level. However, from a performance point of view the third architecture is the only one that introduces a performance overhead on the level of a service call. Additionally, the requirement of transparent service authentication requires a PEP to query the stack trace, which consumes additional time. In this section, we therefore evaluate the performance overhead of transparent service authentication and three different approaches to administer XACML policies.

A. Evaluation setup

The test setup consists of an OSGi service platform from Knopflerfish, installed on a desktop PC with a 2.4Ghz processor. All trust-related components are added during startup of the platform: SLC, UA, GACOS and MSP. For test purposes we developed a $MovieService$ that offers the method $getMostPopularMovie$ to a test client. This service returns a string value with the name of a popular movie, and is not protected. Depending on the test, we will add another security wrapper to the service and measure the performance overhead of authentication the requestor and evaluating the collaboration or adaptation policies.

B. Evaluation results

We evaluate the performance overhead of transparent service authentication and three different ways of managing complex collaboration policies.

During transparent service authentication, the requesting service is identified by querying the stack trace to retrieve the class name, and the OSGi bundle context to gather the bundle credentials (trusted location or signed X.509 certificate). This method of grabbing subject credentials introduces a performance overhead. Through a performance test, we would like to compare this overhead to traditional authentication of passwords or signed certificates.

In a first test, we deploy the MovieService without a PEP. In a second test, a PEP is added to the MovieService that transparently adds the bundle credentials. In a third test, the signature of the getMostPopularMovie method is extended by the AccessControlWrapper to include a username and password, or X.509 certificate, in the service request. The tests are performed on a desktop computer with an AMD 2.4Ghz processor.

Figure 7 shows a response time of 1.8 ms to execute the MovieService without authentication. The second bar in the graph shows an execution time of 5.1 ms in case authentication is performed using an empty login module.

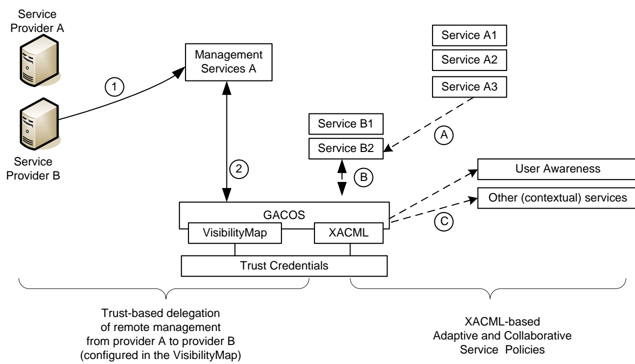

Figure 6. Realtime trust verification and policy enforcement during remote configuration (left side) and service collaboration (right side)

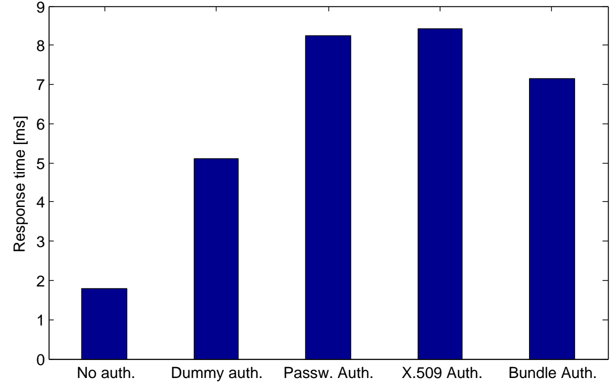

Figure 7. Performance overhead of transparent OSGi service authentication compared to default password and X.509 authentication

The overhead of the PAM framework is thus 3.3 ms. The next three bars show the service execution time for respectively password, X.509 and bundle authentication. With a total execution time of 7.1 ms for bundle authentication and 8.2 ms and 8.4 ms for password and X.509 signature authentication, the performance overhead of transparent bundle authentication is limited to 2 ms (compared to 3.1 ms and 3.3 ms to verify a password and a signature).

The GACOS component leaves the remote management provider three choices to maintain complex policy files:

- use default XACML functions
- use the custom XACML function that executes a single OSGi service that holds the complete policy
- combine multiple custom XACML functions, each executing an OSGi service that hold a subpolicy

From a functional point of view, the first option allows to keep all the logic inside the policy file, whereas in the second and third option the policy details are hidden in OSGi services. However, the first option is less flexible as it requires adaptations to the GACOS components every time an additional environment variable is needed to describe a policy. While the third option allows re-use of subpolicies, the second option offers the advantage of changing a policy at a single location. Another criterion that remote management providers might consider, in choosing one of the three approaches to maintain policies, is the performance overhead.

The performance test consists of a test client contacting the MovieService, which has an access policy that consists of multiple conditions. These conditions are string equations using the service name as parameter. The conditions are logically combined with the OR-relation. The number of conditions is increased from 10 to 100. The policy is implemented using default XACML functions, using a single OSGi service with the complete policy, and using a policy that repeatedly invokes a unitary OSGi service to evaluate the string equation.

Figure 8 shows the performance results. Expressing a policy using default XACML functions or using a single OSGi service performs equally. Evaluating 100 rules also shows a similar performance compared to evaluating 10 rules. The third alternative approach, which repeatedly invokes a basic OSGi service, however has a dependence on the number of conditions. The performance degradation is a result of the performance overhead of the *UseOsgiServiceWithArgument* class. For every call to an OSGi service, the OSGi service registry is queried and a method call is constructed from the parameters supplied in the policy.

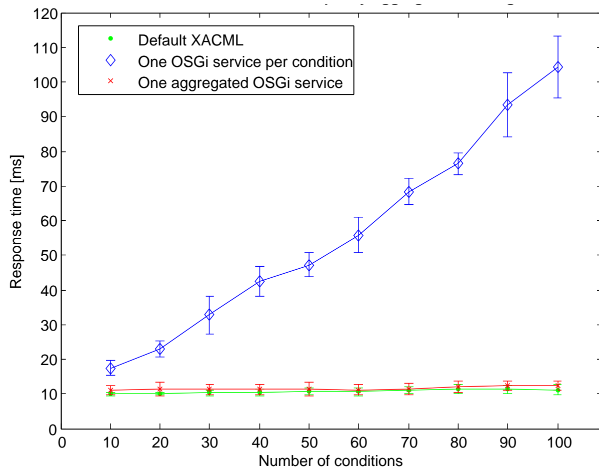

Figure 8. Performance overhead of three policy administration approaches

VI. Conclusions

In this paper we have compared three alternative architectures to create a trusted home service environment: OSGi virtualisation, embedded security and security as a service. Only the latter supports fine-grained access control on method level and parameter level, and allows to define policies to orchestrate services from other service providers. As a result, access to a service can be made dependent on services that check network conditions, contextual (user) information or any other (legacy) service. From a performance point of view, security as a service introduces an overhead on method level. In a proof-of-concept, we have evaluated the overhead of transparent service authentication, which proved to be less than the performance overhead for password and certificate signature verification. We have also compared different approaches to configure trust-based policies. Policies with default XACML rules and policies that implement all rules in a single OSGi service, show equal performance. Policies that refer for each rule to a separate OSGi service, show a performance with degrades fast for an increasing number of rules. However, the latter allows for modular, re-usable policy administration. We have also defined a remote interface that enables a service provider to delegate the management of a specific service to a trusted service provider.

With flexible trust management and fine-grained access control, the presented security components allow premium service providers to aggregate only trusted services. Hence, premium service providers can differentiate with low-cost service providers by remotely configuring secure collaboration of trusted, value-added services.

Acknowledgment

C. Develder is supported by the Research Foundation – Flanders (FWO–Vl.) as a postdoctoral fellow.

References

[1] Y. Royon and S. Frenot, “Multiservice home gateways: business model, execution environment, management infrastructure,” *Communications Magazine, IEEE*, vol. 45, no. 10, pp. 122–128, October 2007.

[2] C. Wu, C. Liao, and L. Fu, “Service-oriented smart home architecture based on osgi and mobile agent technology,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 37, pp. 193–205, 2007.

[3] D. Zhang, H. Lee, and X. Ni, “A new service delivery and provisioning architecture for home appliances,” *IEEE International Conference on Consumer Electronics*, vol. 40, pp. 378–379, 2003.

[4] D. Kang, “Upnp av architectural multimedia system with a home gateway powered by the osgi platform,” *IEEE transactions on consumer electronics*, vol. 51, pp. 87–93, 2005.

[5] OSGi Alliance, “Osgi server platform release 4,” 2005.

[6] Universal Plug and Play Forum, “http:// www.upnp.org/.”

[7] B. Yoon-Hak, “Nom 2.0: innovative network operations and management for business agility,” *Communications Magazine, IEEE*, vol. 46, pp. 10–16, 2008.

[8] Universal Plug and Play Forum, “Upnp device architecture v1.0. 2002, http:// www.upnp.org/.”

[9] N. Goeminne, G. De Jans, F. De Turck, B. Dhoedt, and F. Gielen, “Service policy enhancements for the osgi service platform,” *Proceedings of the 9th International Component-Based Software Engineering Symposium*, 2006.

[10] W. Haerick, N. Goeminne, K. Cauwel, G. De Jans, F. De Turck, B. Dhoedt, P. Demeester, S. Bracke, W. Acke, and C. Bouchat, “Success in home service deployment: Zero-touch or chaos?” *Journal of the communications network*, vol. 4, no. Part 3, pp. 36–43, jul-sep 2005.

[11] “Sun’s xacml implementation,” *http://sunxacml.sourceforge.net/*.