Using JASON to Secure SOA

Łukasz Chmielewski [1] Richard Brinkman [1] Jaap-Henk Hoepman [1,2] Bert Bos [3]

[1] Digital Security group
Radboud University Nijmegen
the Netherlands
{lukaszc,r.brinkman,jhh}@cs.ru.nl

[2] TNO ICT
the Netherlands
jaap-henk.hoepman@tno.nl

[3] Chess IT
the Netherlands
Bert.Bos@chess.nl

ABSTRACT

Nowadays business applications closely collaborate with other business applications by sharing one or more services. Unfortunately, opening your business application to the outside world also sacrifices security. There is quite a number of standards that aims at protecting these services. However, most of these standards require special knowledge about security and are cumbersome to use. Our JASON[1] framework aims at simplifying the task of securing services. A programmer simply annotates his code with appropriate keywords and our tools will generate the security related code. The programmer can simply concentrate on the business application, while the JASON framework does the necessary cryptography.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

Keywords

middleware, SOA security, secure remote management

1. INTRODUCTION

Recent application architectures have become increasingly complex. Starting from mainframe centric, client / server, distributed computing, loosely coupled architecture, they resulted in Service Oriented Architecture (SOA). Each step of this evolution increased the architecture's complexity.

Service Oriented Architecture [8] is an architectural style for designing and utilizing business processes, and defining the infrastructure that allows different applications to participate in business processes. In the SOA model, separate services (that may run on separate servers) are combined to form the business application. These services communicate with each other by passing data.

There are many systems that aim at providing security for SOA architectures. Unfortunately those systems tend to be very complex. A programmer must understand all the security standards, to build even a simple application. Hence, it is crucial to develop solutions that would simplify implementing the security of SOA systems in a generic way.

Such a generic system might be the JASON system. JASON is our JavaCard As Secure Objects Networks platform [2] which was originally designed for smartcards (additionally, analysis of the applicability of the JASON system for M2M systems is in [1]). The aim of this paper is to show that the same principles can be applied to SOA, as well. We also analyzed how to incorporate JASON into SOA. We investigated several ways how JASON could provide the security for SOA in the most flexible manner. The JASON architecture consists of: the JASON compiler (a tool to extract security requirements from annotated Java code) and the JASON framework (a performer of the security actions). The implementation of these tools has not been finished at the time of publication of this article.

We also have improved the JASON compartmentalization mechanism. A JASON compartment limits the access to the service resources by using Java sandbox. Additionally, to limit the access to public methods of objects from various sandboxes on the same JVM, we use a multiple class loaders mechanism (as described in Section 7.1).

Firstly, in Section 2 we give an overview of a work related to JASON. Later (in Section 3) we describe preliminaries: the "old" JASON and the SOA concepts. In Section 4 we present different ways of cooperation between JASON and SOA, and conclude by a description of the best choice for our purpose. JASON annotations are described in Section 5. Subsequently, we present a multiple policies feature (Section 6) and the JASON framework (Section 7) together with the new investigation on JASON compartmentalization. In Section 8, we present security standards of web services that are used within JASON. We finish the paper with the conclusions in Section 9.

2. RELATED WORK

A very interesting approach for distributed policy specification for service-oriented computing is presented in [16]. These system provides various solutions for developing busi-

This research is supported by the research program Sentinels (`www.sentinels.nl`) as project 'JASON' (NIT.6677). Sentinels is being financed by Technology Foundation STW, the Netherlands Organization for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs.

[1] `http://www.cs.ru.nl/jason`

1st International Workshop on Middleware Security (MidSec 2008) December 2, 2008, Leuven, Belgium

ness services, and also enforcing various policies. For the policy specification and enforcement a language PSEL is introduced, together with the necessary framework. Although the system provides means for specifying security aspects, it is not the main part of the system, and for example, separation of services, on the same machine seems not to be considered.

A related work on the security in large-scale distributed systems was performed in [14] for the Globe system. Globe[6] is a distributed system based on distributed shared objects (DSO). The notion of a DSO stresses the property that objects in Globe are not only shared by multiple users, but also physically replicated at possibly thousands of hosts over a wide-area network. Thus, many security issues occur for this system, and they are solved in [14]. This security cannot be applied directly to SOA, due to: model differences (e.g., SOA does not consider objects / services replication), and GLOBE does not use SOA communication standards.

As a related work can be also considered the compartmentalization within the operating system SELinux [9] and the Xen virtualization mechanism [4]. These mechanisms can be used to ensure separation within JASON, however, we decided to use the Java separation mechanism (details can be found in [1]).

Other approaches than the one we propose in this paper do exists, like J2EE (WSIT/XWSS), Spring (Acegi) and JBoss, but none of them are targeted to programmers who lack a thorough understanding of the underlying security protocols and cryptographic techniques.

3. PRELIMINARIES

3.1 JASON Targeted Towards JavaCards

JASON is our JavaCard As Secure Objects Networks platform [2] and was originally designed for smartcards. Lately, we have performed an investigation on the applicability of the JASON concept for M2M (machine to machine) systems [1]. JASON realizes the secure object store paradigm where objects (that are written in Java) are stored on devices and back office systems. Devices may be pervasive, highly mobile, computationally weak, communicationally weak, etc. The JASON platform is a middleware layer which securely interconnects an arbitrary number of smartcards, embedded devices, terminals and back office systems over the Internet. It is important to mention that recently embedded devices has been increasingly equipped with SOA (described e.g., in [7]), and therefore, JASON should be SOA-aware.

The JASON platform supports secure deployment and remote management of secure pervasive systems which run applications from various parties. A JASON application consists of a collection of objects with role-based access, where membership of a role corresponds to the knowledge of a key. In the distributed object model that JASON follows, all objects are separate entities running on separate nodes. Objects interact by requesting remote methods or services from each other. The method invocations are transparent and are performed using secure protocols. Objects do not necessarily know whether its requested method is executed remotely.

A very important concept in JASON is the separation of concerns: the security requirements and the implementation. Programmers only have to specify the security requirements (formulated as an Java annotations), not to implement them. The JASON platform translates these requirements into a secure implementation (based on RMI). At runtime, the JASON platform provides a secure environment and secure communication protocols, according to the specified requirements.

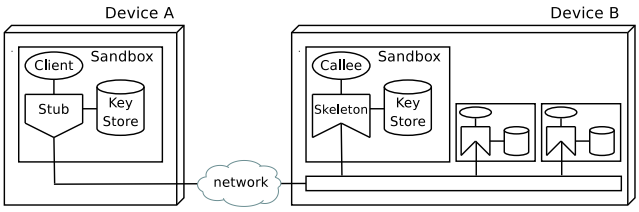


Figure 1: Sandboxing and communication in JASON

Figure 1 conceptually shows how two objects can communicate via stubs and skeletons. The object which calls a remote method on a remote object is identified as the caller, while the remote object is the callee. In this model, stub and skeleton are Java codes produced by the JASON compiler, used by the caller and callee respectively. They provide transparent access to remote methods. The caller locates the interface of the callee and issues a request which is passed to the stub. The stub establishes the connection to the (skeleton at the) callee over the public network using standard protocols and formats. The callee authenticates the caller using the keys in the keystore and evaluates the request. Security requirements for returned values such as authenticity, confidentiality, etc., can be specified. The JASON platform enforces these security properties during the execution of the call.

In JASON, all objects are separate entities, running on separate nodes. The compartmentalization mechanism allows different objects to be safely and securely run on *one* hardware platform. To this end we studied several compartmentalization approaches [1] and their applicability to the JASON platform (see Figure 1). In the end, we have chosen to combine the Java sandbox (for restricted access to resources) and multiple class loaders (to separate services).

3.2 Service Oriented Architecture

Service Oriented Architecture (SOA) [5] defines an infrastructure in which business processes are spread over a number of services. The different services may be owned by different (business) parties, run on different operating systems and written in different programming languages. These services communicate with each other by passing data from one service to another. SOA is often seen as an evolution of distributed computing or modular programming.

Services are relatively large units of various functionality, e.g.: filling out an online application for an account, viewing an online bank statement, placing an order for an online book, or ordering an airline ticket, etc.

To describe the communication between services and their interoperable characteristics, flexible standards are necessary. Web Services based SOA implementations (almost all SOAs are WS based) use XML based protocols to exchange data (like SOAP) and describe the service itself as a WSDL [3] description. A good description of Web Services security standards is in [15].

SOA Building Blocks.

In SOA we can distinguish three types of building blocks (as shown in Figure 2). The Service Provider creates a Web service and publishes its interface and access information

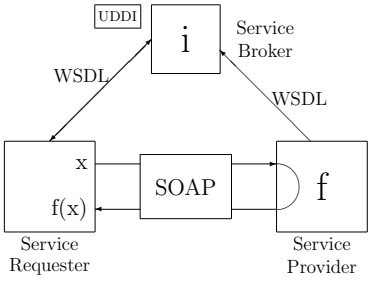


Figure 2: SOA Building Blocks

to the service registry. He should make trade-offs between security and easy availability, and decide how to price the services, etc. The Service Broker (also called service registry) is responsible for making the Web service interface and access information available to any potential service requester. The Service Broker uses the Universal Description Discovery and Integration[2] (UDDI) specification which defines a way to publish and discover information about Web services. The Service Requester is a client that first gets the location of the desired service information from the Service Broker. After that he can contact the Service Provider in order to invoke one of its Web services.

Sometimes in SOA there are also other components involved (e.g., Service Bus), but due to the space constraints we do not describe them in this paper.

JASON does not aim at improving the security of Service Broker now, but it is considered as a future work.

4. JASON CONCEPT AND SOA

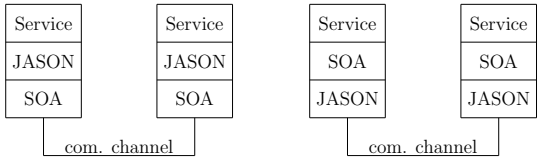
Originally, JASON and SOA aimed at different goals. JASON was meant to be a middleware layer between a PC and a JavaCard making it easy for a programmer without knowledge of cryptography, to build secure smartcard applications. As it turns out (much to our own surprise) the same principle of the original JASON, i.e. the strong separation between secure communication and the core business implementation, can be applied in the broader context in which SOA operates. On the other hand, today lots of embedded devices are equipped with SOA, and thus, JASON should be able to cope with that. Therefore, JASON and SOA can supplement each other.

This paper seeks for an answer, how to achieve that. There are a few obvious ways that can be considered: JASON can be seen as an extension to SOA (JASON on top of SOA) – Figure 3a, SOA can be on top of JASON (here JASON takes care of communication) – Figure 3b, JASON can be a plugged into SOA (shown in Figure 3c), or that JASON can be put alongside SOA (JASON next to SOA; shown in Figure 3d). The first two approaches are not very useful from our point of view because they limit the usefulness of JASON.

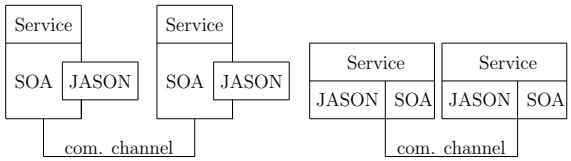
4.1 JASON within SOA

In this section we describe our JASON within SOA approach, which combines JASON next to SOA with JASON plugged into SOA approaches. At the beginning we considered the JASON next to SOA approach as the most suitable. We assumed that separate JASON services should

[2]`http://uddi.xml.org/`



(a) JASON on top of SOA (b) SOA on top of JASON



(c) JASON plugged in SOA (d) JASON next to SOA

Figure 3: Ways of complementing JASON and SOA

implement security issues, but also "talk the same language" (use the same communication standards, e.g., WSDL, WS-Security, etc.) as SOA services. Then programmers using JASON would be able to write secure services without huge effort (e.g., implementing security action, like encryption, etc.). Moreover, because JASON secure web services would comply to existing standards, they would be able to communicate with existing secure SOA services.

These properties are extremely useful and we decided that they are necessary for our new JASON. JASON next to SOA fulfills them, however, this approach has an important drawback: a large part of SOA would have to be reimplemented in JASON. This would significantly slow down the implementation process of JASON. Therefore, we started to look for another concept (or just some modification) that fulfills the aforementioned properties but avoiding this drawback.

We have decided to slightly modify the JASON plugged into SOA approach, but in the JASON next to SOA setting. In plain JASON plugged into SOA, the whole JASON system is a security module of SOA and therefore, the architecture of SOA would be minimally modified. This JASON plugin is called the JASON framework (Figure 5) in our final system, but it is only a part of the whole JASON system.

The other JASON functionality that has to be taken care of is the specification of the JASON security interface. We have decided that the most the most flexible way is to let the programmer specify the security requirements directly in the source code of a program. This way he can precisely specify the security requirements as well as the SOA requirements concerning communication (Web Services annotations). The JASON annotations are taken care of by our JASON compiler – the second part of the JASON system.

The JASON compiler extends the Java compiler in the following way. It translates the JASON annotations and WS annotations to a WSDL file that contains the security requirements expressed in the WS-SecurityPolicy standard.

Figure 4 shows a sample program in which the programmer has specified methods with relevant JASON security annotations (they express: confidentiality, authenticity, etc.) and the web service annotation. The program defines a class named `HomeEmergencyDoor` that represent a hypothetical door device at a Home Control Box[3] (HCB). How to use JASON in the HCB context is described in [1]. The first JASON annotation is `@AvailablePolicies` (a list of JASON annotations is in Section 5) which define two dif-

[3]http://www.homecontrolbox.com/

```
package jason.test;

import jason.annotation.AccessibleTo;
import jason.annotation.AvailablePolicies;
import jason.annotation.Authentic;
import jason.annotation.Confidential;
import jason.annotation.Logged;
import jason.annotation.Policies;
import jason.annotation.Policy;
import jason.annotation.Roles;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
@AvailablePolicies({"OldVersion", "NewVersion"})
@Roles({"OWNER","AMBULANCE"})
public class HomeEmergencyDoor {
  @WebMethod
  @AccessibleTo ({"OWNER","AMBULANCE"})
  @Confidential
  public boolean checkDoorStatus() {
    // code that checks if the door is open
  }

  @WebMethod
  @AccessibleTo({"OWNER"})
  @Authentic
  public void changeDoorStatus
  (@Confidential @Authentic boolean status) {
    // code that opens / closes the door
  }

  @WebMethod
  @Policies({
    @Policy(name="OldVersion",
      accessibleTo=@AccessibleTo({"AMBULANCE"})),
    @Policy(name="NewVersion",
      accessibleTo=@AccessibleTo({"AMBULANCE"}),
      logged=@Logged,
      authentic=@Authentic)
  })
  public void openDoor () {
    // code that opens the door
  }
}
```

Figure 4: A sample program showing the flexibility of specifying both the security requirements and the web service parameters.

ferent security behaviours (that are described as policies): `"OldVersion"` and `"NewVersion"`. These two policies can be used by clients to call the service (as explained in Section 6). It is useful to have more than one policy because, for example, sometimes updating policies on many clients takes much time, and then, temporarily, the old policy might be accepted as well. Subsequently, `@Roles` defines which roles are allowed to access the web service: the Owner's role (this role corresponds to the application of the house owner, that can be stored on the mobile phone), and the Ambulance's role, which roughly corresponds to secret keys or certificates, that are stored by the JASON framework that works on the `HomeEmergencyDoor` Web Service. In the class `HomeEmergencyDoor` three methods are specified:

checkDoorStatus – this method checks if the door is open, and is only accessible to (`@AccessibleTo`) the Owner and the Ambulance. The result of the method is confidential (`@Confidential`), and therefore is sent encrypted to the caller. The security requirements of this method are contained (by default) in both policies.

changeDoorStatus – this method opens or closes the door. Thus, this method is only accessible to the Owner. The Owner's decision `status` should be confidential and signed (`@Authentic`). The result is of type `void`, so it may sound strange to mark it as `@Authentic`. However, it just means that the confirmation of the method's invocation is signed and sent by the service. The security requirements of this method are contained in both policies.

openDoor – this method opens the door (and is similar to `changeDoorStatus`), and is accessible to the Ambulance. The `@Policies` annotation shows how the service can provide different policies. This may be necessary to smoothly upgrade to a new policy version while still accepting old clients. Details of multiple policies and a description of this method are in section 6.

Mixing JAX-WS annotations with JASON annotations gives the programmer more flexibility at the cost of a slightly more difficult interface specification. We have chosen this approach for implementing JASON due to its generality: many instances of JASON and SOA can inter-communicate in a secure way without any changes to the existing security standards used in SOA (e.g., XML encryption). The details of our design decisions for JASON next to SOA are presented in Sections 5, 6, 7, 8.

5. JASON ANNOTATIONS

In this section we describe the JASON annotations. The annotations are based upon our investigation described in Sections 4.1 and 6. We divide them into 4 groups: annotations that consider a whole class (they are placed just before the description of the class), method specific ones, parameter specific ones, and the most complex annotation: `@Policies`. The class annotations are:

1. `@Roles (String[] list)` declares all the roles that can be used with the other JASON annotations.
2. `@AccessibleTo (String[] list)` defines the list of roles that can access the class. If omitted, everybody can access the class.
3. `@AvailablePolicies (String[] list)` lists the names of policies. If omitted, then the default policy is used.

The following annotations can be placed in front of a method:

1. `@AccessibleTo (String[] list)` defines the list of roles that can access the method. An `@AccessibleTo` annotation in front of a method restricts the behaviour of the `@AccessibleTo` annotation in front of the class.
2. `@Logged` specifies that the access pattern should be logged.

These JASON annotation can be specified for parameters and method results:

1. `@Confidential (String[] encryptedBy)` defines that the parameter or the method's result should be encrypted before sending; `encryptedBy` defines the list of roles that can encrypt the parameter; by default it is set to the list from the method's `@AccessibleTo` annotation.
2. `@Authentic (String[] signedBy)` defines that the parameter should be authenticated; works analogically to `@Confidential`.
3. `@Integrity` defines that the parameter should be send in unchanged form.

Additionally, a `@Policies` annotation can be placed at each of the above places to specify a list of different `@Policy` annotations. Each `@Policy (String name, <@Annotation> <annotation>, ...)` defines one policy instance. Parameter `name` specifies the name of the policy, which has to be declared in the `@AvailablePolicies` annotation. The rest of the parameters link to the other JASON annotations. For instance, the `authentic` parameter can be set to `@Authentic(signedBy="...")`. An annotation within a `@Policy` affects only that policy. The parameters that are allowed are dependent on where the encapsulating `@Policy` is placed, following the same rules as specified above.

The syntax for the `@Policies` and the `@Policy` annotation could have been less verbose if Java annotations would have allowed multiple annotation with the same name (but with different parameters) or having an array of different annotation types.

6. MULTIPLE POLICIES

In this section we describe an important new feature of JASON: support for multiple policies. In the original JASON, the compiler produced the secure implementation directly from the security interface (which was just an extension of the Java interface). Hence, if a programmer updated the security interface, the code was compiled into a new executable (every time when the interface was changed, it was necessary to recompile the source code). Therefore, to run some functionality with a different security specification it was necessary to produce (using the JASON compiler) a new skeleton and a new stub, and replace the old versions. To avoid this problem, dynamic policies are used, which can be changed at runtime. Therefore, it is enough to produce a new policy file and deliver it to the JASON framework.

The main consequence of decoupling the policy from the executable code, is the ability to run the code with different policies. This ensures easy transition from one policy to another. Instead of updating the policies of the clients all at once, the service may allow connections from both clients running with the old policy as well as clients who switched to the new policy.

Figure 4 shows a Java class that allows two different policies, which are named `"OldVersion"` and `"NewVersion"` by the new `@AvailablePolicies` annotation. The first and the second method, defined in the class, do not contain any information about policies, and therefore, their security annotations affect both policies. The last method `openDoor()` shows how to specify the different security requirements for various policies. The annotation `@Policies` contains a list of `@Policy` annotations. The first policy, named `"OldVersion"`, defines the method to be accessible to `Ambulance`, while the second one also requires logging the access (annotation `@Logged`) and signing the response message (annotation `@Authentic`).

Although a programmer can write the policy directly in WSDL/XML, we recommend to write the security requirements as annotations in the source code. This makes it easier for the programmer to keep the policy and the code in sync. A compiler tool can read the annotation and generate the WSDL policy as an XML file.

7. JASON FRAMEWORK

Keeping the policies separate from the implementation not only allows us to use the code with multiple policies, also the JASON framework can perform security actions separate from the business application. The JASON framework minimally consists of a JasonHandler which has access to a key store and the policies. Figure 5 shows a simplified model for Web Services (similar one can be made for e.g., RPC). The JasonHandler can be seen as a kind of firewall or gate keeper. It checks the incoming messages against the access control rules in the policy. When access is granted the JasonHandler checks whether the message is appropriately signed or encrypted. If everything is according to the policy it decrypts the encrypted parts and checks and removes the signatures. The result of this process is a standard SOAP message which man be handled by other handlers, and eventually parsed by a Web Service Handler into a method call.

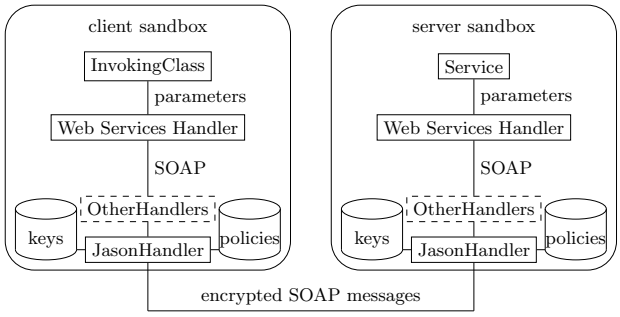


Figure 5: JASON framework for Web Services

The same JasonHandler is used for outgoing messages. It ensures that outgoing messages are correctly signed and/or encrypted before they leave the sandbox. In the implementation of JasonHandler we use many WS-security standards, which are described in Section 8.

7.1 Compartmentalization

In JASON, every object is placed in a JASON compartment. Each compartment consists of a Security Manager to control access to resources (files, databases, socket connections, etc.), and a ClassLoader to prevent an object to invoke methods of another object that is loaded by a different ClassLoader. The JASON Handler creates the compartments.

To really separate two services that run on the same JVM, a standard Java sandbox is not enough, since a sandbox only limits access to resources, like files and network sockets, but not to other classes. Two services in different sandboxes can still call the public methods of each other (this is not the case on a JavaCard, which has a firewall built in). To provide real separation we load each service by a different ClassLoader. A class loader defines a new name space for classes that are loaded by that class loader. Therefore, classes loaded by different ClassLoaders cannot access methods of each other. This approach has proved to be successful in, for example, the implementation of Apache's Tomcat server[4].

8. SECURITY STANDARDS

SOA security involves a daunting number of security standards. JASON is not meant as yet another security standard. Instead JASON strives to simplify the task of writing a secure application, while being standards compliant. The JASON annotations are a hint to our JASON compiler to use the correct security standards. In our view a program-

[4] `http://tomcat.apache.org/`

mer is not required to dick into all the security standards before he can write a secure program.

The JASON annotations form the security contract which both the service and the invoking client have to obey. The most natural place to put this information is the WSDL description. The WS-Policy standard allows us to extent the WSDL with any kind of policy. There are a lot of Web Service related security standards that can be used as WSDL policies. The WS-SecurityPolicy standard describes how the requirements of a security standard can be specified as a consistent WSDL policy. For instance, it can describe which part of a SOAP message has to be encrypted and which part has to be signed. It also specifies which standard to use. WS-SecurityPolicy can reference a lot of other security standards. Here, we list only those useful for JASON:

WS-Security [10] describes how to encrypt / sign part of a SOAP message. Basically it will use XML Encryption and XML Digital Signatures. (Closely linked with the `@Confidential` and `@Authentic` annotations).

WS-SecureConversation [12] describes how to set up a secure session. (Closely linked with the `@Roles` and `@AccessibleTo` annotations).

WS-ReliableMessaging [11] deals with integrity (`@Integrity` annotation).

WS-Trust [13] describes how to handle trust relations and right delegations.

Security Assertion Markup Language (SAML) can be used to exchange authentication and authorisation information between different parties.

9. CONCLUSIONS

Securing web services involves many standards, which are most often cumbersome to use. A programmer should be an expert in security and SOA standards to get all the settings files right. JASON aims at reducing this complexity. A programmer defines security settings by declaring them, not by implementing them. Encrypting and signing parameters and method result is as simple as inserting annotations in the source code. The annotations are used by the JASON tools to generate definition files which can be used by the JASON framework.

Our main goal has been to achieve good synergy from combining the JASON concept and the SOA architecture. We believe that we achieved this goal and the resulting system (JASON within SOA) is described in Section 4. Therefore, our future research has been concentrating on writing the JASON compiler and the JASON framework. We will also design a key management system and a policy distribution system. Eventually, we also consider performing a security validation of the JASON system.

10. REFERENCES

[1] B. Bos, Ł. Chmielewski, J.-H. Hoepman, and T. S. Nguyen. Remote management and secure application development for pervasive home systems using Jason. In *In 3rd International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing*, pages 7–12, Istanbul, Turkey, July 2007.

[2] R. Brinkman and J.-H. Hoepman. Secure method invocation in Jason. In *USENIX Smart Card Research and Advanced Application Conference (CARDIS)*, pages 29–40, San Jose, CA, USA, Nov. 2002.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. Technical report, W3C, 2001.

[4] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[5] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[6] P. Homburg, M. V. Steen, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7:70–78, 1999.

[7] S. Karnouskos, O. Baecker, L. M. S. de Souza, and P. Spiess. Integration of soa-ready networked embedded devices in enterprise systems via a cross-layered web service infrastructure. In *Proceedings of 12th International Conference on Emerging Technologies and Factory Automation*. IEEE Computer Society, 2007.

[8] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[9] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42. The USENIX Association, June 2001.

[10] OASIS. *Web Service Security: SOAP message security 1.1 (WS-Security 2004)*, February 2006. `http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf`.

[11] OASIS. *Web Service Reliable Messaging (WS-ReliableMessaging 1.1)*, June 2007. `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf`.

[12] OASIS. *WS-SecureConversation 1.3*, March 2007. `http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.pdf`.

[13] OASIS. *WS-Trust 1.3*, March 2007. `http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.pdf`.

[14] B. Popescu, M. van Steen, and A. S. Tanenbaum. A security architecture for object-based distributed systems. In *Proceedings of 18th Annual Computer Security Applications Conference*, pages 161–171. IEEE Computer Society, 2002.

[15] J. Rosenberg and D. Remy. *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption*. Pearson Higher Education, 2004.

[16] W. T. Tsai, X. Liu, and Y. Chen. Distributed policy specification and enforcement in service-oriented business systems. In *ICEBE '05: Proceedings of the IEEE International Conference on e-Business Engineering*, pages 10–17, Washington, DC, USA, 2005. IEEE Computer Society.