Remote Management and Secure Application Development for Pervasive Home Systems Using JASON

Bert Bos, Łukasz Chmielewski, Jaap-Henk Hoepman, Thanh Son Nguyen

***Abstract*—In a modern house, the number of electronic devices keeps increasing. More and more of these devices become interconnected, to provide new services. And they start disappearing in the environment, forming a truly pervasive home system.**

The embedded devices in such pervasive networks are usually owned and managed by several entities with conflicting interests. This makes secure remote management of such devices a challenging task. Software development to implement such services need to solve complex security problems and need to be aware of the large spread of capabilities among the different devices.

We present ongoing research to apply and extend the JASON architecture [1] to handle such remote management and secure software development issues.

***Index Terms*—pervasive systems, home automation, middleware, sandbox, security, remote management**

I. INTRODUCTION

In a modern house, the number of electronic devices keeps increasing. Houses may be equipped with many modern appliances, ranging from hi-fi sets, tv's, pc's, smart fridges and washing machines, doorbells, light switches, thermostats and the like. Sensors record activity in the house, to automatically switch off the light in empty rooms for instance. They could also serve as burglar alarms. Surveillance cameras may monitor the doors. Electronic energy meters, gas meters and water meters record and report usage periodically and automatically without human intervention.

These devices get networked. They synchronize and cooperate to provide better services. The surveillance camera can instruct the heating system to switch on the heater, order the lighting system to switch on the lights in the hall and the living room, and tell the television to play the preferred channel when the owner approaches the front door. It can activate the alarm system when unrecognized people try to break in.

These devices need to be updated and managed from time to time. Each device offers services from different service providers. Such management may necessarily be done by the service providers themselves. Remote management is preferable, since service providers would wish to update or interact with all devices that run their services in a large geographic area without visiting each and every one of them. In the extreme case, we cannot visit these devices, since there are too many of them, and we may not even know where they are.

The above scenario is an instance of a domestic pervasive system. That is, a system consisting of large group of small or embedded devices, interconnected and spread all through a modern family home. Pervasive computing (also known as ubiquitous computing, everyware [5] etc.) refers to the integration of systems into their environment, to build context-aware systems and services that are invisible and blended into the surroundings. It has attracted a lot of academic research. Moreover, several industrial projects have started to use the results of this research to implement actual pervasive systems. The Home Control Box project[1] is one of these.

In our case study we envisage a central home control box (HCB) in a house which connects to all local devices. The HCB connects to service providers over the Internet. Devices can interact with each other and with service providers through the HCB.

It is difficult and error-prone to develop secure applications for pervasive devices, especially because these devices have varying capabilities, and may lack the most basic resources. It would be useful if software developers would have to focus only on implementing the functional aspect of a service, while the security aspects could be specified by the developers and be implemented automatically. The JASON architecture provides this separation of concerns.

It is also important that applications by many different vendors can work safely together on the same multi-application platform. One core issue is that the management is done by different and often competing parties. For instance, two providers of similar services which run on the same HCB, may try to interfere with each other. It is desirable in such cases that management activities of parties do not interfere wih each other, if not allowed by the security policies in force. This requires proper sandboxing of applications.

In this paper we discuss how the JASON approach provides a programming paradigm and a corresponding platform for the development of secure domestic pervasive services, that can be remote managed securely. We show use cases and refinements to the JASON platform according to the learnt lessons.

II. THE JASON APPROACH

JASON is our Javacard As Secure Objects Networks platform [1]. It realizes the secure object store paradigm where objects are stored on different devices and back office systems. Devices can have one or more of the following characteristics: being pervasive, highly mobile, computationally weak, communicationally weak etc. The JASON platform is being developed as a middleware layer which securely interconnects an arbitrary number of smartcards, embedded devices, terminals and back office systems over the Internet.

B. Bos is with Chess IT, the Netherlands, e-mail: `Bert.Bos@chess.nl`

Ł. Chmielewski, J.-H. Hoepman, and T. S. Nguyen are with Radboud University Nijmegen, e-mails: `{lukaszc,jhh,thanhson}@cs.ru.nl`

J.-H. Hoepman is also with TNO, The Netherlands, e-mail: `jaap-henk.hoepman@tno.nl`

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

The JASON platform supports secure deployment of secure pervasive systems which run applications from various parties, and supports remote management of these systems. JASON applies role-based access control. An application consists of a collection of objects which give access to pre-defined roles. 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 request is done using methods provided by the JASON secure communication layer. The method invocations are transparent. Objects do not necessarily know whether its requested method is executed remotely.

One important concept in JASON is the separation of concerns: the security requirements and the implementation. Programmers only have to specify security and remote management requirements in the application's interface description, not to implement them. The JASON platform translates these requirements into a secure implementation. At runtime, the JASON platform provides a secure environment and secure communication protocols, according to the specified requirements.

In the next two sections we present two individual techniques in more details: the Secure Communication Layer and the Sandboxing mechanisms. Then we present few practical use cases to determine the practicality of the JASON approach and to refine the JASON platform according to lessons learnt.

III. Secure Communication Layer

This section describes the communication model of the Jason objects. This Secure Communication Layer (SCL) enables objects to call remote methods and services in a secure fashion. The interface language can be either an extension of the Java interface language, or the extension of WSDL [2]. These extensions allow extra security and remote management keywords to be added to the interface. The Jason platform translates these requirements into a secure implementation.

SCL currently consists of two communication methods: *secure method invocation* (SMI [1]) based on RMI and *secure web services* (SWS) based on web services. Each approach has its own advantage and disadvantage, mainly due to the underlying remote method invocation and web services technologies. Given the overhead of encoding and decoding XML, users can choose the remote method invocation approach instead of web services. On the other hand, web services are more flexible. For a comparison of remote method invocation and web services, see [3].

Figure 1 shows how two objects can communicate via stubs and skeletons in SMI[2]. The object which calls a remote method on a remote object is identified as 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 and evaluates the request. Security requirements for returned value such as authenticity, encryption etc., can be specified. The JASON platform enforces these security properties during the execution of the call.

[2] SWS works in a similar fashion. For clarity we do not describe that in the paper.

IV. Sandboxing

In Section II we explained the JASON approach, assuming all objects were separate entities, running on separate nodes. In this section, we discuss how these different objects can safely and securely be run on *one* hardware platform. To this end we have studied several *sandboxing* approaches and their applicability to the JASON platform (see Figure 1).

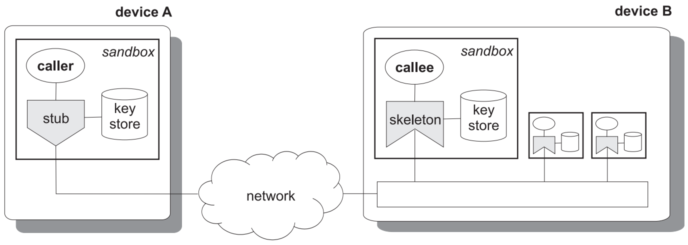

Fig. 1. Sandboxing and SMI in JASON

We have considered three approaches and their representatives. These approaches realize compartmentalizations on different levels. On the *Application Level*, we have the Java VM which provides a sandbox for a Java application from within the Java Virtual Machine [12]. On the *Kernel Level*, SELinux [7] includes means for sandboxing in the kernel, therefore it is possible to enforce each and every application to operate in its own sandbox. On the *Operating System Level*, the Xen [4] approach is to enforce each different operating system in different sandboxes and to allow a number of such sandboxes to run concurrently. Figure 2 shows the ideas behind the three kinds of compartmentalization. Comparison of the properties of these systems is given in Section D.

The name "sandbox" appeared in the literature relatively recently, however, a mechanism of compartmentalization for safety reasons has been known since the 70s [10]. A sandbox (also called compartment) is a secure environment to run possibly untrusted code safely. Its approach is to restrict applications to pre-assigned resources. The network access, the ability to inspect the host system or read from input devices is usually disallowed or heavily restricted within a sandbox. In this sense, sandboxes are a specific example of virtualization. A sandbox

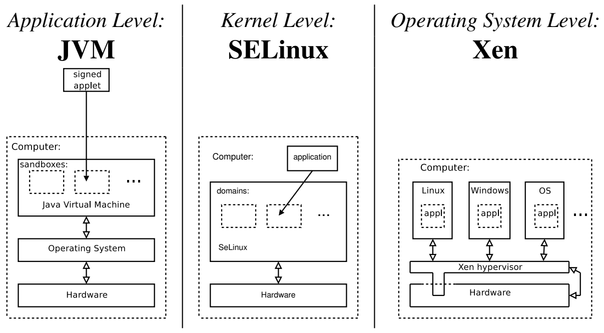

Fig. 2. Different levels of compartmentalization

is often used to execute an untested code, or programs from unverified third–parties and untrusted users.

One can find sandbox mechanisms in a number of popular and widely-used applications: Java 2, SELinux, Xen, Sandbox for Grids (S4G), User Mode Linux (UML), chroot, Denali, VMware, Minix 3, ActiveX, to name but a few. However, we concentrate on the three "fundamentally different" approaches and their representatives to discover the most suitable one for implementing JASON.

A. *Java Sandbox*

Java 2 is a powerful development environment. There are a few fundamental components responsible for its security. The first one is the class loader architecture, which is the program that is responsible for loading Java classes. The second one is the security manager, which is responsible for limiting code to its sandbox environment. Another one is the byte code verifier, that checks if a bytecode, which is about to run, was produced by a proper Java compiler and therefore if it is safe to run it on the Java VM. There are also Java 2 components that are not strongly connected with sandboxing issues, but are important for its security (safety features built into the Java Virtual machine and into the language and the Java crypto API).

Java 2 security mechanisms provide unique features that help writing security sensitive applications. Functionality of signing code and X.509 certificates is also provided. Java Authentication and Authorization Service (JAAS) simplifies writing role-based applications. Furthermore, code verification helps detecting malicious software before running it on a virtual machine. Together with standard sandboxing, these tools make Java 2 very reasonable choice for implementing the JASON system.

B. *SELinux: Security-Enhanced Linux*

SELinux [8][7][11] was a project to port the work of developing a mandatory access control architecture done by the National Security Agency (NSA) and the Secure Computing Corporation (SCC) on the Mach and Fluke OS's to Linux. Now SELinux is an implementation of a flexible mandatory access control (MAC) architecture (called FLASK) into the Linux kernel. The policy for decision making (defined in a policy file) is performed by security server with a general security interface. The general security interface enables implementation of various security models. It requires only to write an instance of a security server and the rest of the system can be unchanged. SELinux provides an example security server that implements a combination of Type Enforcement (TE), Role-Based Access Control (RBAC), and optionally Multi-Level Security (MLS). These security models provide significant flexibility through a set of policy configuration files. However, the possibility of implementing one's own security server is a significant advantage of this system.

C. *Xen*

Xen [4] is a x86 virtual machine monitor (hypervisor) which allows many guest operating systems to share a conventional hardware in a safe and resource managed fashion. This aim is achieved by providing an idealized virtual machine abstraction. Operating systems like Linux, FreeBSD, or Windows can be ported to run on Xen with minimal effort. This approach has an advantage that a sandbox can host any kind of object[3] (but at the "expense" of having to run a full operating system) that could run on the operating system being sandboxed. The decision not to give full virtualization without hardware support (currently supported by Intel VT or AMD-V chips) gives Xen nearly optimal performance.

Since 2005 there has been a tendency to put greater effort into the security aspects of Xen. Such a secure system is sometimes called security enhanced Xen (XenSE) [9]. Since then, research on improving security of Xen has been carried out extensively.

D. *Comparison*

We present an aspect based discussion about the differences and similarities between SELinux compartment, Java sandbox, and Xen guest operating system. The aspects have been chosen with respect to the JASON's security requirements and safety of the implementation.

Due to space constraints we present only major security apects. The minor aspects are presented in Section C in the appendix.

System modification threat. This kind of threat happens when a malicious application manages to "escape" from a sandbox and modifies the working system.

In this aspect Xen and Java are relatively safe tools. Xen is a relatively small system (around 40K of code). Furthermore, some implementations of Java VM have been formally verified [6].

In SELinux every part of the Linux kernel can be attacked (and Linux kernel is relatively big), which enlarges the possibility of a malicious code escaping from a compartment.

Flexibility of security supervisors and policies. This aspect describes how flexible the mechanisms that maintain policies are and how flexible policies can be defined in each system.

An example SELinux security server contains concatenation of RBAC, TE approach, and optionally Multi-level security, which makes it strongly flexible. Moreover, it is possible to develop one's own security server (however it involves a large effort). The SELinux policy language is quite complicated, because it has to cope with properties of the whole operating system.

Java security manager has good flexibility and does not have to concentrate on operating system issues. Java policy language seems to be transparent, simple, and powerful enough.

Xen defines a privileged domain that enforces administrator's policy. The general aims of Xen are slightly different than those of SELinux and Java, but still Xen seems to be fairly flexible with respect to its aims (e.g., its possible to dedicate a device to only one guest operating system). However, for JASON purposes Xen policy seems to be much less flexible than SELinux or Java policies.

Gaining privileges. In this paragraph we describe how a program can get access to the sandbox.

In Java VM code can gain privileges to a sandbox by static assignment (when the application is started) or by various kinds

[3] in contrast to Java VM where only Java objects can be run.

of authentication: signatures or password. Adding new features can be achieved easily. In SELinux a process is matched to some privileged domain by default. Then processes can change its domain if some condition holds. Unfortunately, SELinux has no support for signatures and adding this feature to the system seems to be challenging. In Xen a process is matched to a guest operating system when it starts. Entering guest operating system by a process in other way has to be implemented "outside" of Xen.

Communication between sandboxes. SELinux does not support explicit API for communication between domains (but it can be relatively easily implemented). Xen does not support any transparent way to communicate between many guest operating systems (although they could communicate through operating system's ports). In Java there are many tools that could be used for communication between sandboxes, e.g., remote method invocation (RMI).

Level of development. Java 2 and Xen are the most developed and widely used of the considered approaches. SELinux is still in an experimental phase (however, it is already officially mainstreamed into the Linux kernel).

Supported platforms within a sandbox. In Java sandbox only programs compiled to Java bytecode can be run. This is a serious limitation of the system (however, there exist some Java bytecode compilers for languages other than Java, e.g., Python). SELinux is more universal – any program compiled under Linux can be run within a compartment. Xen is the most universal from the considered systems. Any program can be run within potentially any (mainstream) guest operating system.

Lightweight. In this paragraph we describe the resource costs of using sandboxes under considered systems.

The most lightweight from the considered systems are Java and SELinux. Under both systems creating additional compartment does not cause much work for the security supervisor. The least lightweight is Xen, because every sandbox "takes up" virtualization of one operating system. However, in a situation when not too many sandboxes are used it seems to be a reasonable choice.

E. Conclusions

Java 2 sandbox functionality, lightweight of the system, and cryptography support makes this system a very good choice for implementing the JASON system. The main disadvantage of this system is the limitation that only Java objects can work in the sandbox.

Although SELinux is a promising system we have rejected it for our future research due to its disadvantages (for details see Section D). Firstly, the system is in a relatively early development stage. Secondly, the system is vulnerable to any "normal" Linux kernel vulnerability, which has relatively large code base (in comparison with, e.g., Xen).

Xen virtual operating system seems to satisfy most of the security requirements of the JASON sandbox. It can maintain fairly complex policies and is well developed. The disadvantages are: lack of crypto support, no transparent way of communication between guest systems is provided, and hardware support for full virtualization is necessary. The most important advantage is universality – the fact that almost any application can be run within a compartment. Any application within guest operating system is only bound to the system's limitations.

V. USE CASES

In order to refine the JASON platform, we study the Home Control Box (HCB) system and a few HCB applications as use cases. We study the general security requirements of the applications. Assuming that a HCB participates in the JASON network, we discuss the requirements that could be fulfilled with the existing JASON solutions and the additional functionalities that are necessary to add to JASON itself.

A HCB system consists of several HCBs. These are small and powerful computers being placed inside the house. Each HCB connects to various home devices (television, burglar alarm, heating system, electronic door etc.), to HCB providers, to other service providers, to the police and hospitals.

The HCB providers have control over the HCBs and keep track of these in their databases. Service providers need agreements with the HCB providers to use the HCBs. For the fact that service providers need authorization from HCB providers first before privileges to use HCBs are granted to them, and the fact that the HCBs are regularly tracked by the HCB providers, we also use the term Back Office System (BOS) to refer to the HCB providers. Note that this BOS is different from the back office systems that each service provider may maintain.

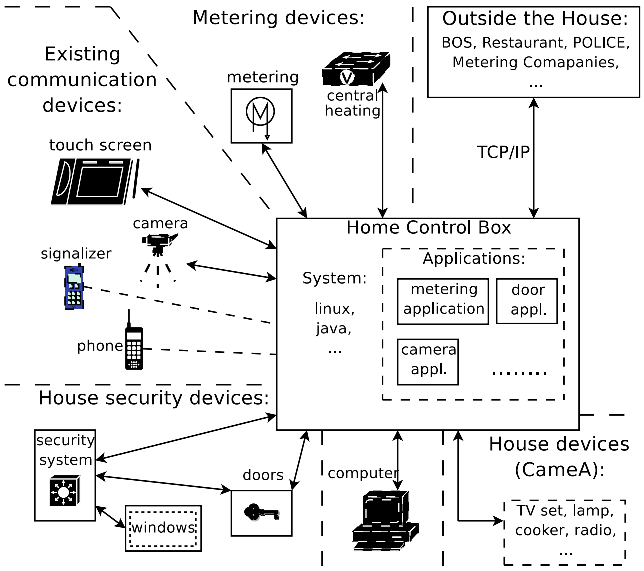

Fig. 3. the Home Control Box and home devices

From inside the house, devices are connected in some way to the HCB, e.g. by a cable or a wireless connection. From the outside, service providers locate their devices through the corresponding HCB. Since it is a generic container, typically service providers need to install an application on the HCB to control their devices. For instance, there is an application to control the surveillance camera, to receive instructions from the safeguard company as well as to send a burglar alarm to them. The HCB however only allows to install applications if the BOS approves it, which typically means they have a contract. The HCB runs these applications, forwards requests and responses back and forth.

We show two HCB use cases. Our goal is to see the practicality of JASON in different situations, therefore we select the use cases such that they cover more and different uses of the HCB. One case (the Health Alarm) involves a situation where emergency overrides security checks, while the other case (the Energy Meter) involves strict policy checks before an action is performed. We show their security requirements and whether those requirements can be fulfilled by the JASON platform.

A. Health Alarm

The health alarm is a service that allows users to warn the hospital in case of an emergency. The goal is to help elderly people to live on their own, without constant nursing. In case of a health emergency, the health alarm provides an easy and informative way to call hospitals, easier than using phones. The user holds a small device which can be integrated into a wristwatch. In case of illness or emergency, the user can press the signaler which sends signals (carrying heart beats, blood pressure etc) to the HCB, which in turn sends that to the hospital. By means of an indoor camera, a video connection is established between the house and the hospital. The doctor can see the situation and decide whether an ambulance is necessary.

Several security requirements can be identified. Consider a few function calls by the HCB, the signaler and the hospital. The requirements are as follows (Figure 4). Next to each requirement, $+$ means that it can be implemented by JASON, $-$ means that it currently cannot.

Figure 4 shows a situation when the signaler requests the method "health alarm" on the HCB to report a health emergency, together with medical data of the owner. The signaler first connects to the HCB and issues the method call. Before executing the method, the HCB authenticates the signaler to avoid abuse (caller authentication)($+$). Because of the emergency, the HCB does not check immediately whether the security policies allow this action but will log and audit later (logging and emergency overrides)($-$). The HCB sets high priority for this emergency call so it can be executed before other methods, say reporting energy usage (setting high priority for the call)($-$).

The HCB forwards the alarm by calling the method "health alarm" on the hospital and passes the medical data along. Security requirements are as follows. The HCB authenticates the hospital (callee authentication)($+$[4]) and sets up a secure communication link to it (link encryption)($+$). The hospital authenticates the HCB (caller authentication)($+$) and the signaler to see exactly which signaler went off. This is to prevent abuse and to send the ambulance to the right address. Then it sets up the secure communication link (link encryption)($+$), bypasses security policies to solve the emergency case (logging and emergency overrides)($-$) and sets high priority for this call over other calls ($-$).

[4] This currently need to be done "by hand" as part of the object look up process, meaning that the developer need to implement it in the function call. This reduces the "separation of concerns" (between functionality and security of the implementation) objective of the JASON platform. It is therefore useful to extend JASON with this.

The hospital requests to "initialize video connection" so the doctor can see the situation in the house. The HCB authenticates the hospital (caller authentication)(+). It streams the video content to the hospital securely (and only to the hospitals, not to thieves, say) (secure streaming of video)(-). The hospital needs to authenticate the service user (Callee authentication)($+$). Connection should be established with high priority so the case can be resolved quickly (setting priority for the method call)($-$).

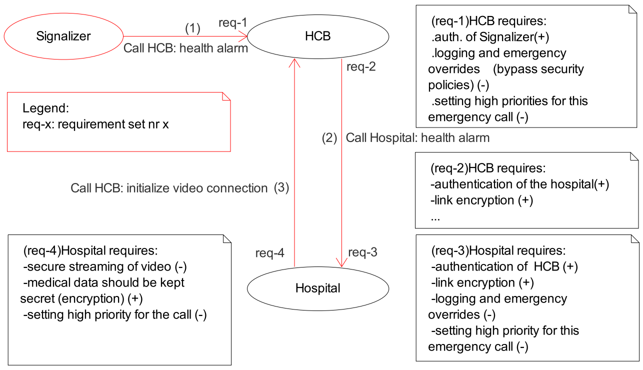

Fig. 4. Requirements for the health emergency alarm

Logging and "emergency overrides" is useful in several cases. In practice, in case of an emergency, the doctor treats the patient first before checking the papers. Similarly, in case of a method request with emergency, the callee can authenticate the caller and allow emergencies override security policies. The callee however logs this fact and investigates later. This functionality is currently missing from the JASON platform.

Setting priority for roles and method calls is useful. For instance, a method call to the police should be processed before a call to the energy provider. The JASON platform still lacks this capability.

B. Energy Meter

Many modern houses are equipped with electronic energy meters. The energy meter monitors and reports energy usage automatically and periodically. If the user has his own, but not constant power supply such as solar power, he can even sell back to the energy provider, through the energy meter. Usage data is sent to the energy provider through the HCB. Under normal circumstances there is no need for the energy provider to physically visit and collect data from the meters anymore.

Similarly to the health alarm case, consider a few function calls and their security requirements. However, due to space limit, we do not repeat the security requirements that already followed from the previous use case but refer to the Figure 5 instead.

The energy meter requests the method "report energy usage" on the HCB to send meter data to the HCB, which in turn sends that to the energy provider. There is a mutual distrust between the energy provider and the service user[5] , therefore the energy provider authenticates the HCB (caller authentication)(+) and

[5] However, they both have to trust the energy meter to some extent.

the energy meter (nested caller authentication)(-), checks for the integrity and authenticity of the meter data (data integrity)(+) and updates its database.

Occasionally, the energy provider requests to change the calibration information of the energy meter. It requires that whether all new calibration parameters are updated, or none (atomicity of the transaction)(-). Since there is a mutual distrust between the service user and the service provider, the service user requires the calibration parameters to be logged as well on the HCB as evidents for future cases (logging parameters)(-).

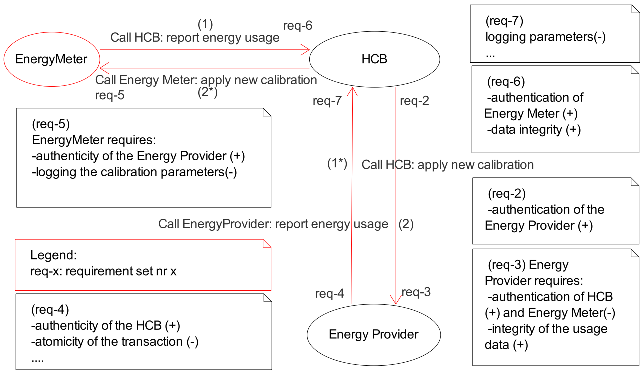

Fig. 5. Requirements for the Energy Meter

C. New Requirements

We have seen the use cases and the new requirements for the JASON platform. Based on this experience we have built a generalized M2M scenario that is the aimed scenario of new JASON (described in Section A in the appendix).

The JASON platform need to be extended with the following functionalities: automatic authentication of the callee, setting priority for the calls, new security requirements for sending parameters or results of method calls (e.g., integrity), atomicity of transactions, logging and "emergency overrides" and secure streaming of video.

Several consequences were identified relating to these new requirements. In case of emergency overriding security policies, we have to create a log so tracing back is possible later. Issue connected with "emergency overrides" is asynchronous method's calls (situation when a caller does not wait for receiving a result from a callee). How and where the log data is kept so that it is secure and still easily accessible to the service providers. Identify the types of policies that can or cannot be overridden and determine how far the emergency can override. We might also want to back up some pieces of essential data before executing the emergency request. In this way we can fall back safely in case the log audit concludes that the request was malicious.

Another important issue is remote management in which more than two parties are involved. This situation is shown in Figure 5. Here the energy meter sends report to the energy provider through the HCB. It is desired that the report should be signed by the energy meter and by HCB. It is feasible that in general recursive remote management properties are desired (e.g., multiple signing data, forwarding an encrypted data, signing an encrypted data etc.).

VI. Conclusions and Further Research

It is difficult and error-prone to develop secure applications for pervasive devices, especially because these devices have varying capabilities, may lack the most basic resources, are owned and managed by several entities with conflicting interests, leading to complex security policies. In this paper we have presented an ongoing research to apply and extend the JASON platform to handle such remote management and secure software development issues. To this end, JASON provides a programming paradigm and a corresponding platform for the development of secure domestic pervasive services, that can be remote managed securely.

Several research challenges were identified. For instance, how to combine the advantages of the sandboxing mechanisms, how to define a clear interface language for security requirements, requirements for new security keywords and implementations to the JASON platform, how to guarantee secure interfaces and to resolve conflicts between keywords, and developing support to automatically translate the security requirements into secure implementation. Further research also takes into account naming and publishing consequences, communication consequences, key management, logging, auditing, and transaction support.

References

[1] Richard Brinkman and Jaap-Henk Hoepman. Secure method invocation in jason. In *USENIX Smart Card Research and Advanced Application Conference (CARDIS)*, pages 29–40, San Jose, CA, USA, November 2002.

[2] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Technical report, W3C, 2001.

[3] William R. Cook and Janel Barfield. Web services versus distributed objects: A case study of performance and interface design. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 419–426, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[5] Adam Greenfield. *Everyware : The Dawning Age of Ubiquitous Computing*. New Riders Press, March 2006.

[6] Pieter H. Hartel and Luc Moreau. Formalising the safety of java, the java virtual machine and java card. Technical report, University of Twente, 2001.

[7] Peter Loscocco and Stephen 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.

[8] Peter Loscocco and Stephen Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, pages 115–134, July 2001.

[9] Carlos Rozas. Intels security vision for xen. http://www.xensource.com/files/XenSecurity_Intel_CRozas.pdf.

[10] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.

[11] Stephen Smalley. *Configuring the SELinux Policy*, Ferbuary 2005. NSA technical report.

[12] Robert F. Stark, Joachim Schmid, and Egon Borger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag Berlin Heidelberg, 2001.

APPENDICES

A. THE GENERALIZED M2M SCENARIO

The JASON platform was born with secure network of javacards as application. The JASON concepts, however, are equally applicable to a secure network of objects of any kind. Besides domestic pervasive systems such as the HCB system, JASON is being extended to support more general architectures with more complex security features as well. An example is the Machine-to-Machine (M2M) architecture with its various instances (payment terminal, park and pay system, monitoring of fuel and gas distribution systems, to name a few). M2M refers to data communications between machines. The architecture usually involves a middleware layer which interconnects myriad of sensors, computers, back office systems, service providers and the like.

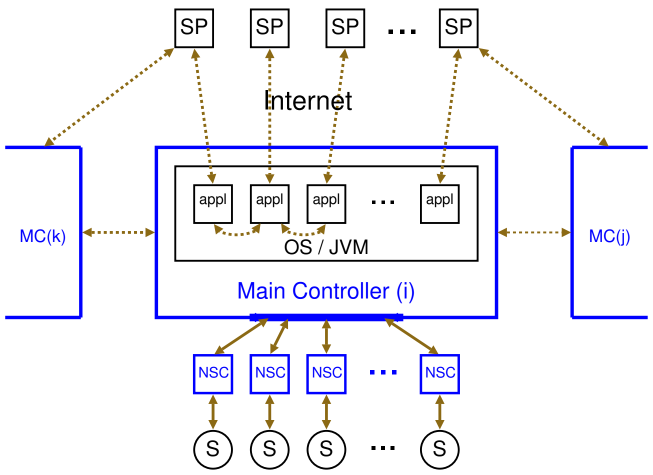

Fig. 6. Generalized M2M architecture

The Figure 6 presents a M2M network with Main Controller (MC) to which several Net of Subcontrollers (NSC) connect to. Applications on these nodes are in separate sandboxes. These sandboxes can communicate through predefined interfaces with respect to the user-specified security policies. Service Providers (SP) manage their services in the network by communicating with the responsible controllers.

The JASON project aims at improving the security level of M2M-like systems. We have kept this fact in mind when refining the JASON security capabilities based on the case studies. Overall, the secure object network paradigm is designed with practical applications being taken into account.

B. EXAMPLE APPLICATION OF THE JASON PLATFORM

This example demonstrates the application of the JASON functionalities on HCB applications. Figure 7 shows a HCB with only two applications: the energy metering and the door applications which control the electronic energy meter and the door respectively. Each application runs in a separate sandbox. Communication between these two sandboxes is disallowed. The metering application periodically queries the meter, sets up a secure link to the electricity provider and sends the report. The door application accepts orders going from the owner's mobile phone through the BOS. The policy regarding when to open the door can be bypassed by hospitals or the police in case of an emergency. This act however is always logged.

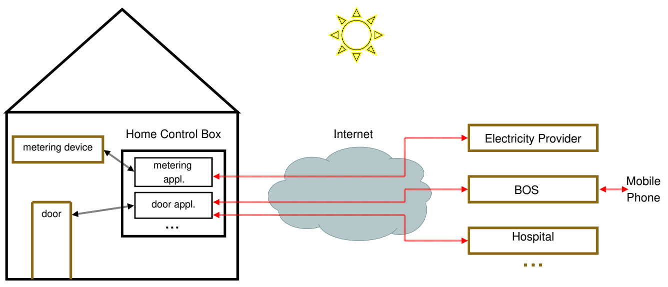

Fig. 7. JASON for Home Control Box

The HCB exposes an interface through which the domestic appliances can be controlled [Figure 8]. The interface in this example is an extension of the Java interface[6]. It contains entries about the roles, the methods to be called and the security keywords.

```
public  interface HomeEmergencyDoor
{
  roles   OWNER AMBULANCE;

  accessible to   ALL
  confidential authentic   bool checkDoorStatus ();

  accessible to   OWNER
  authentic   void
  changeDoorStatus (confidential authentic   bool
  status);

  accessible with logging to   AMBULANCE
  authentic   void openDoor ();
}
```

Fig. 8. JASON code for Home Control Box

The keyword `roles` specifies the entities to which this interface is available. For each method, a list of authorized entities is given by the keyword `accessible to`. When a method is `accessible with logging to` a specific role, any entity with that role can invoke the method without checks (bypassing certain security policy), but the action will be logged for auditing. The keyword `authentic` expresses the requirement to verify the sender (e.g. by signature), while `confidential` requires that the data being exchanged must be kept secret (e.g. by encryption).

The method `checkDoorStatus()` checks whether the door is opened. This method with the requirement `accessible to ALL` is accessible to all roles which are listed in the interface and not to anyone else. The result is required to be encrypted and signed by the door application (`confidential authentic`).

The method `changeDoorStatus()` opens or closes the door according to the `status` parameter. It is accessible only

[6] Users can choose to specify the interface in this way or using WSDL with keyword extensions.

to the owner. The parameter should be sent encrypted and signed. The result being `authentic void` means that the caller is informed whether the operation has completed by a signed message.

The method `openDoor()` is similar to the method `changeDoorStatus()` except that it is dedicated to the ambulance in case of an emergency, with logging and auditing.

Once the interface is specified in this way, the JASON compiler generates a secure implementation based on the given requirements.

New requirements following from the generalized M2M scenario is broadcasting methods. It may be interesting to built in Jason flood (broadcast) of method calls. This situation happens when the main controller sends the same information to many sensors. Instead sending this information separately to every sensor it might be desired that a sensor would propagate method call to its peers.

C. Sandboxing

Here we present a discussion about minor security aspects of sandboxing and Table I, which summaries all of the security aspects.

Invasion of privacy. This threat happens when a malicious code manages to "look outside" the sandbox and read "private" data of the system. The systems behave similarly like for the system modification threat with respect to the invasion of privacy.

Denial of service. Java VM does not provide any countermeasure against this threat. SELinux can be patched to limit usage of devices or processor. In Xen limiting resources (CPU time) can be done through policies.

Code verification. Java VM support simple and static way of bytecode verification. Aim of this checks is to reject bytecode that is produced by malicious compilers. SELinux and Xen do not support code verification.

Role Base Access Control. SELinux and Java 2 (JAAS) provide Role Base Access Control (RBAC) model. It is really convenient because this kind of access control is flexible and useful for JASON's purposes. Xen does not support RBAC.

	System		
Aspect	Java	SELinux	Xen
System modification threat	+	+/−	+
Invasion of privacy	+	+/−	+
Denial of service	−	+/−	+/−
Code verification	+	−	−
Flexibility of security supervisor and policy	+	+	−
Role Base Access Control	+	+	−
Gaining privileges	+	+/−	−
Communication between sandboxes	+	+/−	−
Resistance to operating system weaknesses	−	NA	NA
Level of development	+	+/−	+
Supported platforms within a sandbox	−	+/−	+
Lightweight	+	+	−

TABLE I

COMPARISON OF COMPARTMENTALIZATION MECHANISMS