Int J Wireless Inf Networks (2010) 17:11–25
DOI 10.1007/s10776-010-0116-y

A Security Architecture for Reconfigurable Networked Embedded Systems

Gianluca Dini · Ida Maria Savino

Received: 31 July 2008 / Accepted: 11 June 2010 / Published online: 1 July 2010

Abstract Nowadays, networked embedded systems (NESs) are required to be reconfigurable in order to be customizable to different operating environments and/or adaptable to changes in operating environment. However, reconfigurability acts against security as it introduces new sources of vulnerability. In this paper, we propose a security architecture that integrates, enriches and extends a component-based middleware layer with abstractions and mechanisms for secure reconfiguration and secure communication. The architecture provides a secure communication service that enforces application-specific fine-grained security policy. Furthermore, in order to support secure reconfiguration at the middleware level, the architecture provides a basic mechanism for authenticated downloading from a remote source. Finally, the architecture provides a rekeying service that performs key distribution and revocation. The architecture provides the services as a collection of middleware components that an application developer can instantiate according to the application requirements and constraints. The security architecture extends the middleware by exploiting the decoupling and encapsulation capabilities provided by components. It follows that the architecture results itself reconfigurable and can span heterogeneous devices. The security architecture has been implemented for different platforms including low-end, resource-poor ones such as Tmote Sky sensor devices.

Keywords Security · Reconfigurability · Middleware · Networked embedded systems · Sensor networks

G. Dini (✉)
Dipartimento di Ingegneria dell'Informazione: Elettronica, Informatica, Telecomunicazioni, University of Pisa, Via Diotisalvi 2, 56100 Pisa, Italy
e-mail: g.dini@iet.unipi.it; gianluca.dini@ing.unipi.it

I. M. Savino
Divisione Difesa, Spazio e Ambiente, Elsag Datamat S.p.A., Via Laurentina 760, 00143 Rome, Italy
e-mail: ida.savino@elsagdatamat.com

1 Introduction

The convergence of communication, computing and control and recent technological advances in low-power, low-cost communication miniature computing devices and sensors make it possible to build NESs that can be deeply embedded in the physical world including home appliances, cars, buildings, and people [4, 17, 44]. These large-scale, widely distributed, heterogeneous, pervasive systems include autonomous and interconnected units, which not only have capabilities of sensing, but also those of acting in and on the environment [2]. Networked Embedded Systems are traditionally designed and built to perform a fixed set of predefined functionalities in a well-known operating environment, and, after deployment, in the vast majority of applications, their functionality is not expected to change during their lifetime. However, nowadays this design approach can no longer be pursued. In order to be cost-effective and operational over time, NES are required to be reconfigurable in order to be customizable to different operating environments and/or adaptable to changes in the operating environment. Unfortunately, *reconfigurability* acts against *security* as it introduces new sources of vulnerability. Given the interactive and pervasive nature of NES, a security breach can result in severe privacy violations and physical side effects, including property damage, injury and even death.

Security in NES tends to be a more difficult long-term problem than it is today in desktop and enterprise computing [21]. In fact, the drive to provide richer functionality, increased customizability and flexible reconfigurability of NES requires the ability to remotely download software after they have been deployed [39, 40]. However, downloading malicious software (including viruses, worms, and Trojan horses) is by far the instrument of choice in launching security logical attacks. The magnitude of this problem will only worsen with the rapid increase in the software content of embedded systems. Furthermore, NES often use wireless communication to simplify deployment and increase reconfigurability. So, unlike a traditional network, an adversary with a simple radio receiver/transmitter can easily eavesdrop as well as inject/modify packets in a wireless network. Finally, cost reasons often cause embedded devices to have limited energy, computation, storage, and communication capabilities. This leads to constraints on the types of security solutions that can be applied. To further worsen this scenario, embedded systems often lack adequate physical/hardware support to protection and tamper-resistance. This, together with the fact that NES can be deployed over a large, unattended, possibly hostile area, implies that each embedded device is exposed to the risk of being compromised. Compromised devices have to be, at least, logically removed from the network communication. Usually, the ability to logically remove compromised devices from the network translates into the ability to revoke keys [5]. In fact, cryptographic algorithms do not expose keys so that secret keys can only be compromised by compromising the device. It follows that by revoking all keys of a compromised device, it is possible to remove the logical presence of that device from the system.

In this paper we present a security architecture that supports secure communication and secure reconfiguration in NES. The architecture integrates, enriches, and extends a component-based middleware by means of security abstractions and services. The proposed architecture is general and flexible. It is general because it has been designed from the ground up to be implementable on a wide range of devices, comprising low-end ones, and the abstractions it provides can be used to build applications and higher-level services. Finally, it is flexible as it accommodates different implementations of the security services according to the specific application requirements and constraints. Several middleware systems for low-end, resource constrained NES have been designed and implemented [1, 9, 10, 16, 25, 28, 29, 41, 46–48] and a lot of research on security has been done for this type of embedded systems [6, 15, 20, 24, 27, 30, 36, 37, 40, 49, 50]. However, only a few middleware systems consider security [32]. Our security architecture provides a stride towards filling this gap.

The architecture comprises three basic services: Secure Communication, Authenticated Downloading, and *Rekeying*. *Secure communication* implements secure channels according to the application-specific security policies and requirements. The Secure Communication service allows an application developer to define and implement fine-grain, application-specific secure communication policies. The service allows also dynamic negotiation of security protocols so that a device can adapt to the changed operating conditions. *Authenticated downloading* guarantees that software components are remotely downloaded from trusted sources. It is a key service for secure reconfiguration in NES because it prevents an attacker from modifying or installing a rogue one. Authenticated downloading in NES is particularly challenging because it must also be efficient in terms of communication, storage and computation in order to be sustainable on low-end resource-poor devices. Finally, *Rekeying* is explicitly devoted to key distribution and revocation. Key distribution establishes and refreshes secure channels, as dictated by good cryptographic practices. However, the ability to revoke keys is equally important as it translates into the ability to logically remove a device from the communication system. Therefore, although embedded devices may lack any preventive physical measure against tampering, however the architecture provides at least a reactive mechanism to logically remove compromised devices.

The architecture achieves its flexibility by providing the above services in terms of *component frameworks*, i.e., collections of well-defined middleware components with well-defined interfaces and well-defined mutual relationships. One component framework is defined for each basic service. The architecture does not commit to a specific implementation of component frameworks. In contrast, component frameworks constitute a flexible software fabric that allows an application programmer to implement the above security services in the most suitable way for the specific application. In order to instantiate the architecture, the application developer chooses the secure communication protocol, the authenticated downloading protocol, and the rekeying distribution scheme that better fit the application requirements and constraints, and encapsulate them in the corresponding component frameworks.

The paper is organized as follows. Section 2 provides a review of related works. Section 3 gives an overview of the system model at the middleware layer. Section 4 describes the proposed security architecture in terms of services and related component frameworks. Section 5 describes a possible implementation of the architecture aimed at showing its generality and flexibility. As to generality, we have implemented the architecture on the

RUNES middleware [9, 10] running on Contiki [13] on Tmote Sky sensor nodes [33] to show the architecture ability to scale down to low-end embedded devices. As to flexibility, we have implemented two specific rekeying and authenticated downloading protocols (and their related secure communication protocols) mainly for exemplifying purposes. Of course, an application developer can make different choices according to the application requirements and constraints. Finally in Sect. 6 we draw our final remarks.

2 Related Work

During the past few years, researchers have devoted much effort in designing and developing middleware for heterogeneous, resource-constrained NES such as wireless sensor networks. Relevant examples include [1, 9, 10, 16, 25, 28, 29, 46]. However, security has been largely ignored in the current generation of this type of middleware [32, 41, 47, 48]. A few relevant exceptions include Matè [25], Impala [28] and ZUMA [45]. The architecture we propose in this paper fits in with this research strand, it has been conceived for this type of middleware and, in particular, it has been implemented within the RUNES middleware [9, 10] on Contiki [13].

With reference to closer works, the ZUMA middleware provides efficient end-to-end secure communication by means of Kilavi [43], a security protocol for home-automation applications. ZUMA secure communication services can be implemented within our architecture by properly implementing the corresponding component frameworks. In other words, ZUMA secure communication architecture may be a possible instantiation of our architecture. Matè is a tiny virtual machine running on TinyOS [18]. Matè helps programmers to develop expressive and concise sensor network applications. It supports reconfiguration in terms of infection of small program capsules through the network, and it is resilient to buggy or malicious capsules so preventing applications to crash the system. Impala is a middleware system and API for sensor application adaptation and updates. Impala has some resemblances to Matè although Impala's security checks are more oriented to unfortunate programming errors than malicious attacks. Our architecture is complementary to Matè and Impala from both the communication and reconfiguration standpoint. Actually, our architecture focuses on secure communication which is not an issue in Matè and Impala. Furthermore, while Matè and Impala control the run-time behaviour of components, our architecture includes an authenticated downloading service that verifies the authenticity of components coming from remote sources.

It is important to notice that security threats, vulnerabilities and related countermeasures in heterogeneous, resource-constrained NES have been intensively investigated [6, 36, 40]. Furthermore, theoretical models and protocols addressing specific problems have been proposed. For instance, secure communication and key management are the focus of an industrial consortium, IEEE standard, and research community [15, 20, 24, 27, 30, 37, 49, 50]. Finally, specific security architectures encompassing both secure communication and key management have been proposed [38, 42]. Notwithstanding, as stated above, these models, protocols, and architectures are still largely ignored in practice by most middleware systems of the current generation [32]. Therefore, our architecture can be considered a stride towards filling this gap. Furthermore, several of these solutions can be implemented within the component framework we provide. Section 5 provides a few examples of that.

Finally, with reference to secure reconfiguration, we can notice certain similarities between our architecture and DeLuge [14], a scheme for authenticated downloading of software that has been conceived for TinyOS [26]. However, DeLuge only allows large-grain downloading of the whole memory image of an embedded device, so requiring device rebooting upon software reconfiguration. In contrast, we support authenticated downloading at a finer grain, namely at the level of single software component, so removing the constraint of device rebooting.

3 System Model

We consider an *heterogeneous* system composed of both general-purpose mobile devices and embedded devices capable of sensing and acting on the environment. Mobile devices fulfill tasks, either in cooperation or isolation, and interact with embedded devices to sense the environment or act on it. In order to do that, devices communicate through a wireless network. Possible applications are, for example, monitoring the environment, detecting and pursuing a target, providing support to first rescue team and so forth [8].

In these application scenarios, operational conditions may change unpredictably. Therefore, devices must be able to reconfigure themselves in order to comply with the changed conditions. Reconfiguration may concern a device functionality, e.g., conditions change from "normal" to "exceptional", or a different implementation of a given service. For instance, a drastic change in light conditions may require that the localization service implementation changes from a vision-based one to an ultrasound-based one [3].

In such a scenario, we assume that devices mount a middleware layer that provides a set of reusable programming abstractions that allow configuring, deploying, and reconfiguring software [10]. The middleware hinges on basic runtime units of encapsulation and deployment, referred to as *components*. Components provide services to other components through one or more well-defined *interfaces* and can have *dependencies* on other components. This enables the implementation and deployment of different versions of the same component, each tailored to a specific device type. Furthermore, inter-dependent components are grouped into *component frameworks* to build more complex services. More precisely, a component framework is an encapsulated composition of components that address some area of functionality, and which accepts additional components, which modify and extend the component framework behaviour. A component framework is itself a component and can thus recursively contain other component frameworks.

This component-based approach abstracts away from the actual platform implementation and provides a general design framework for NES. In fact, it supports and promotes encapsulation and modularity of design and implementation and thus makes it possible to integrate devices with hardware and software of completely different origin and makes them safely and securely coexist and collaborate.

In addition, we assume that components can be dynamically added and removed. This makes it possible dynamically reconfigure applications according to the changing operational conditions. Reconfiguration consists in downloading a new component from a possibly remote source, instantiating and/or removing components at runtime, and also dynamically changing the components interconnections.

4 Security Architecture for NES

The proposed security architecture enriches and extends the middleware layer with abstractions and mechanisms for secure reconfiguration and secure communication. The security architecture is designed and implemented as a collection of components frameworks encapsulating the security aspects with well-defined interfaces as well. In this way, the security architecture results itself reconfigurable and can span heterogeneous devices. Furthermore, for the heterogeneity requirement the architecture must be accessible also to very simple devices with possibly low computational and data storage capabilities. Hence, components have been implemented taking into account possible technological limitations of the device involved in the scenario.

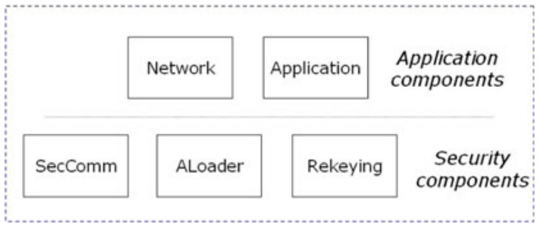

Fig. 1 The security architecture

As shown in Fig. 1, the security architecture hinges on the following components frameworks: the *SecCom* component framework that provides the secure communication service, and the *Rekeying* component framework that performs key revocation and distribution, and the *ALoader* component framework that guarantees authenticated download for secure reconfiguration.

Some scenarios could require the secrecy, integrity and/or authenticity of communication among remote components. So, *SecCom* provides the secure communication service by enforcing application-specific, fine-grained communication security policy. A security communication policy defines the set of protocols that have to be applied in order to protect the communication among components in different physical devices. A protocol is a set of cryptographic transformations that have to be performed on the messages sent through the network. In order to support reconfigurability, *SecCom* could negotiate the secure communication protocol as well as the cryptographic algorithms necessary to implement it.

The *Rekeying* component framework provides the rekeying service by performing the key distribution and revocation. Different instantiations of *Rekeying* can implement different rekeying protocols according to the security requirements. In order to support reconfigurability, *Rekeying* could be loaded after deployment to adapt to a change in security operating conditions.

Finally, in order to support reconfigurability, a device could load a new component through the network. So, the device has to receive proofs that the component comes from a trusted source (component authenticity) and that the component has not been modified (component integrity). The *ALoader* component framework provides the authenticated loading service by enforcing a security loading policy. Different instantiations of *ALoader* can implement different loading policies according to both the operating environments and the required trade-off between security and performance.

In the rest of the section, we provide a detailed description of components, interfaces, and services provided. We describe the component interfaces in the Interface Description Language (IDL). For each operation provided by an interface, IDL allows us to specify the name and the type of the arguments the operation takes as input (`in` parameters), and the type of values the operation returns as output (`out` parameters).

4.1 The Secure Communication Service

The *SecCom* component framework fulfills the communication security requirements in terms of confidentiality, integrity and authenticity. Different instantiation of *SecCom* can provide different security communication protocols. The protocol can be pre-deployed in the device (*Local Protocol*), or can be negotiated and retrieved from a remote trusted part (*Remote Protocol*). In this case, *SecCom* is able to dynamically negotiate security protocols so that the device is able to adapt to the change of the operating conditions. In order to protect communications of an application, the *SecCom* component has to be inserted between the application and the Network component. The *SecCom* component framework provides the application components with the same interface as the *Network* component (Fig. 2). So doing, *SecCom* can be inserted without affecting the application component from a functional point of view. Furthermore, this allows us to remove *SecCom* without affecting the application components and reconfigure the software stack of a device by using *SecCom* only when needed. Hence, *SecCom* can be inserted and removed so that software can be transparently reconfigured for security purposes.

Operationally, *SecCom* intercepts incoming/outgoing messages and applies them the cryptographic transformations specified by the security communication protocol. The actual specification and implementation of the protocol depends on several factors including the kind of embedded computing device and the hardware and software platform on which *SecCom* is deployed. The component can be implemented at software. However, if a hardware cryptographic device is present, the component can encapsulate and abstract the cryptographic services offered by that device.

4.1.1 Secure Communication Protocol

A security communication protocol is defined as a set of *rules*, each of which consists in a *transformation*, a *cryptographic suite*, and a set of fine-grained *selectors*.

A transformation specifies the set of cryptographic processing to be applied to messages in order to protect communication and guarantee their confidentiality, integrity, and authenticity. In other words, a transformation specifies how to process a message before sending/receiving it to/from the network. A transformation can be either a cryptographic primitive or a combination of primitives. Cryptographic primitives can use cryptographic keys.

A cryptographic suite specifies the actual cryptographic primitives, and the related keys, to be used in a transformation. Keys are specified by a key unique identifier. In the simplest devices with limited capabilities, the cryptographic suite only includes symmetric ciphers, one-way hash functions, and HMACs. In more advanced devices, a cryptographic suite may include digital signatures.

Finally, selectors are a fine-grained mechanism that specifies which messages a transformation has to be applied to. Selectors include at least the type of message (e.g., the port), the destination address, the source address, whether the message is incoming or outgoing and so forth.

For example, let us assume that for messages of type `T` we have to guarantee both confidentiality and integrity. One way to achieve these goals is to send a message `m` after it has been processed according to the transformation `t: encrypt(m||hash(m))` where `encrypt` specifies the symmetric encryption, `hash` specifies the hashing operation and || is the concatenation operation. If we would like to use `SHA-1` as hash function and `RC5` as symmetric cipher keyed by the `K` key, then we specify the cryptographic suite `c: (encryption=RC5, keyid=K; hash=SHA-1).` Finally, the selectors specifying the relevant messages are `s: (msgType=T, direction=outgoing).` It follows that the secure communication protocol is specified by the rule `r=(s, t, c).`

At this stage we do not specify the implementation of protocol rules. Such an implementation depends on several factors including the kind of device the implementation is for.

4.1.2 The SecCom component framework

In case of a Local Protocol, *SecCom* includes the *StaticSeC* component, that actually implements the secure communication protocol. More in detail, when *Application* sends a message to Network, *StaticSeC* intercepts the message, retrieves the matching security rules and processes the message accordingly. If the performed algorithms need cryptographic keys, *StaticSeC* retrieves them from the *KeyDB* component (Fig. 3).

In case of a Remote Protocol, *SecCom* also includes a Negotiator component, that negotiates the secure communication protocol. Before the *Application* component connects to the *Network* component, the *Negotiator* negotiates the secure communication protocol, instantiates the

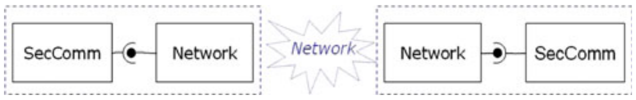

Fig. 2 The *SecCom* component framework

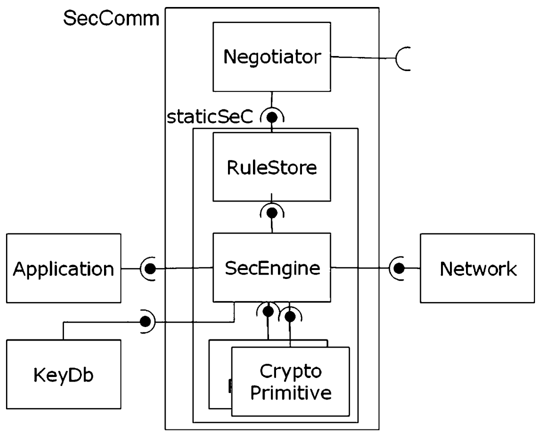

Fig. 3 Internal structure of *SecCom*

StaticSeC that implements the negotiated protocol, and inserts it between *Application* and *Network*.

At this stage we do not specify the implementation of these components but we limit to specify their functional behaviour and their interface in the rest of the section. Such an implementation depends on several factors including the kind of device the implementation is for. We will discuss some examples of implementation in Sect. 5.

The *StaticSeC* Component

The *StaticSeC* component includes a *RuleStore* component storing the protocol rules, a *SecEngine*component that is responsible for applying the transformations specified by the rules, and *CryptoPrimitive* components implementing the specific cryptographic transformations. If the performedalgorithms need cryptographic keys, *StaticSeC* retrieves them from the *KeyDB* component.

The *SecEngine* component fulfills two tasks. First, it provides *Application* with the same interfaceas *Network*. Second, it implements the secure communication protocol by processing theincoming and outgoing messages according with the security rules. For each message, *SecEngine*retrieves the security rule associated with the message from the *RuleStore* component, and appliesthe message transformation specified by that rule. In doing that, *SecEngine* exploits theservices provided by the *CryptoPrimitive* components as needed. If a transformation requires acryptographic key, *SecEngine* retrieves the key from the *KeyDB* component. Finally, *SecEngine*forwards the resulting message to the next component, i.e., *Network* or *Application* according towhether the message is outgoing or incoming. The *SecEngine* component could be implemented as a rule interpreter. However, this implementation choice could be pursued only for more capable embedded devices. In contrast, in case of the simpler devices, the *SecEngine* can be implemented ad-hoc for one or more protocols.

The *Rulestore* component contains the security rules that have to be applied to messages. Upon receiving a request from *SecEngine*, rulestore searches and returns the matching rules specifying the crypto transformations to be applied to the message. A matching is found according to the rule selector.

The *CryptoPrimitive* component implements a cryptographic primitive, such as a cipher (i.e., SkipJack [35], RC5 [22]), a one-way hash function (i.e., SHA-1 [34]), or a hMAC. The *CryptoPrimitive* components do not preclude the use of the public key cryptography if it is available on the device. These components can be implemented at software, or if present they can embed cryptographic services provided by hardware devices.

The *KeyDB* component stores the cryptographic keys used to perform the cryptographic primitives. Every key is associated with a key identifier that uniquely identifies the key within the *KeyDB* component.

The *Negotiator* component

In case of Remote Protocol, *SecCom* includes the *Negotiator* component that is responsible for negotiating the secure communication protocol and instantiating the staticsec component.

The *Negotiator* communicates with a trusted site through the network following a secure communication protocol implemented, in its turn, by another *SecCom*. The *SecCom* component framework connecting *Negotiator* to *Network* implements a local protocol and thus it includes only a *StaticSeC* (Fig. 4).

Protocol negotiation occurs when an *Application* connects to the Network component. In this case, *Negotiator* creates a *StaticSeC* by performing the following operation:

1. remote downloading, if necessary, of the *Rulestore*, *SecEngine*, and *CryptoPrimitive* components that implement the required secure communication protocol
2. instantiation of the *Rulestore*, *SecEngine*, and *CryptoPrimitive* components,
3. connection between *Rulestore* and *Negotiator*, *Rulestore* and *SecEngine,SecEngine* and CryptoPrimitives,

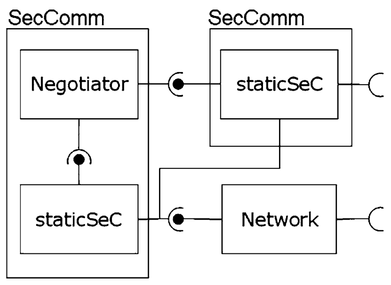

Fig. 4 Remote protocol

4. connection of both *Application* and *SecEngine* and *SecEngine* and Network.

If, as a consequence of changed operational conditions, *Application* needs no secure communication, *Negotiator* removes the instantiated staticsec and directly connects *Application* and *Network* components. Protocol negotiation can also occur in any other moment so that Negotiator has to modify *StaticSeC* in order to perform a different security communication protocol. Usually, negotiation consists in selecting a protocol and inserting the rules of that protocol into the *Rulestore*. The insertion of new rules may cause the instantiation of new *SecEngine* and *CryptoPrimitive*s components. For instance, if the *SecEngine* is implemented as a sort of rule interpreter, negotiation of a protocol only involves the insertion of the protocol rules into the *Rulestore* and, possibly, the instantiation of some *CryptoPrimitive* components. In case of ad-hoc *SecEngine*, negotiation of a new protocol would cause the instantiation of a new *SecEngine*.

4.1.3 Interfaces and Dependencies

In this section, we define the interfaces and the dependencies of components that implement the *SecCom*.

The *SecEngine* component

The *SecEngine* component implements the interface `INetwork` providing the *Application* component with the same interface as Network.

```
Interface INetwork{
     int send (in Message msg, in Address add);
     int receive(out Message msg, out Address
 add);
}
```

The field `Message` includes at least the data `MsgData` and the message type `MsgType`. The field `Address` usually contains the IP address and port of the source and destination application so that it can identify the communication connection.

SecEngine uses the `INetwork` interface, the `IRuleStore` interface, the `IKeyDB` interface and one or more interfaces of CryptoPrimitive components.

The *Rulestore* component

The *Rulestore* component provides the following interface `IRuleStore`:

```
Interface IRuleStore{
     int getRule (in MsgSelector ms, out
 Transformation t,
           out CryptoSuite c);
}
```

The method `getRule` returns the `Transformation` and the `CryptoSuite` associated with the messages described by the `MsgSelector`. In order to dynamically change the rules, the rulestore provides the following interface `IRuleUpdate`:

```
Interface IRuleUpdate{
      int insertRule(in MsgSelector ms, in
 Transformation t,
          in CryptoSuite c);
     int updateRule(in MsgSelector ms, in
 Transformation t,
          in CryptoSuite c);
     int deleteRule(in MsgSelector ms);
}
```

The methods `updateRule, insertRule` and `deleteRule` respectively updates, inserts and deletes the rule associated with the messages described by the `MsgSelector`.

The *CryptoPrimitive* component

The *CryptoPrimitive* component provides a cryptographic primitive, such as a cipher, a one-way hash function or a hMAC. For example, the *CryptoPrimitive* component implementing a cipher provides the other components with the following interface:

```
Interface ICipher {
      int Encrypt (in Byte[] dataIn, in int
 lengthIn, out Byte[] dataOut,
          in int lengthIn, in Key k);
      int Decrypt (in Byte[] dataIn, in int
 lengthIn, out Byte[] dataOut,
          in int lengthIn, in Key k);
}
```

The method `Encrypt` encrypts the message `dataIn` of size `lengthIn` by using the key `k`, and returns the processed message `dataOut` of size `lenghtOut`. The method `Decrypt` decrypts the message `dataIn` of size `lenghtIn` by using the key `k`, and returns the plain message `dataOut` of size `lenghtOut`.

Whether the *CryptoPrimitive* component implements a hash function, it provides the following interface:

```
Interface IHash{
      int getHash(in Byte[] dataIn, in int
 lengthIn, out Byte[] hashOut,
      out Byte[] lenghtOut);
}
```

The method `getHash` returns the hash value `hashOut` of size `lenghtOut` applied to the input `dataIn` of size `LengthIn`.

The *KeyDB* component

The *KeyDB* component provides the following interface `IKeyDB`:

```
Interface IKeyDB{
      int getKey(in IndexKey ik, out Key k);
}
```

The method `getKey` returns the key value `k` with index `ik`.

The *Negotiator* component

The *Negotiator* component depends on the `IRuleUpdate` interface in order to dynamically change the rules associated with messages. Furthermore, the *Negotiator* component uses the `INetwork` interface to receive the secure communication protocol through the network. Usually, this interface is provided by a *StaticSeC* component. This component performs the algorithms and stores the security rules that are applied to the messages transmitted by Negotiator.

4.2 The Rekeying Service

The *Rekeying* component framework performs key distribution and revocation and updates the key repository on the device. Usually the keys are distributed through the network in such a way that both confidentiality and authenticity are guaranteed.

In order to provide the rekeying service, a device, referred to as the *Key Sender* (KS), performs key distribution and revocation. The other devices, referred to as *Key Receiver (KR)*, verify the key authenticity and upload their local key database. Operationally *Rekeying* on KS side generates a new key and performs the cryptographic transformations according to the rekeying protocol in order to guarantee key authenticity and confidentiality. The

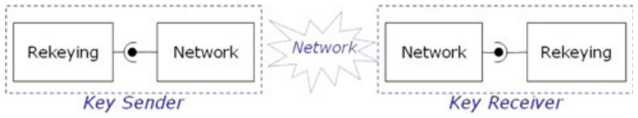

Fig. 5 The *Rekeying* component framework

Rekeying component on KR side performs the complementary transformation and verifies that the key comes from a trusted part, i.e., KS.

Usually, the keys are transmitted through the network so that *Rekeying* is connected to the *Network* component as depicted in Fig. 5.

4.2.1 The Rekeying Component Framework

As shown in Fig. 6, the *Rekeying* component framework on KS side includes a *SecCom* and a *GenerateKey* component, whereas *Rekeying* on KR side includes a *SecCom* and a *AuthKey* component.

The *SecCom* component framework performs the secure communication protocol guaranteeing confidentiality and integrity of rekeying messages. The secure communication protocol enforced by *SecCom* depends on the chosen rekeying protocol. On KR side The *GenerateKey* component is responsible for generating a new key according to the rekeying protocol. The renewed key is passed to *SecCom* component framework that guarantees key confidentiality. On KR side the *AuthKey* component receives the key from *SecCom* and verifies its authenticity according to the rekeying protocol. In case, *AuthKey* updates the *KeyDB* component. The internal structure of *AuthKey* and *GenerateKey* components strictly depends on the chosen rekeying protocol.

It is worthwhile to notice that the *Rekeying* component does not preclude the use of public key encryption. It is

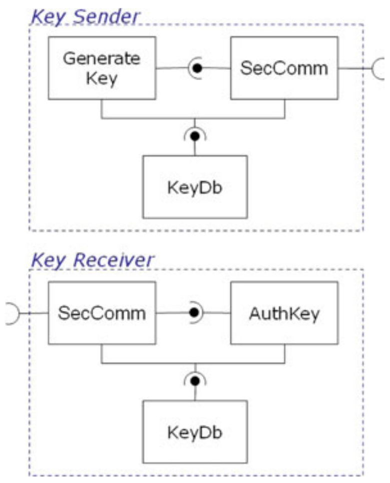

Fig. 6 Internal structure of *rekeying*

only necessary that *CryptoPrimitive* components implementing this form of cryptography are available.

4.2.2 *Interfaces and Dependencies*

In this section, we define the interfaces and the dependencies of components that are included in *Rekeying*.

The *KeyDB* component

The *KeyDB* component provides the following interface `IKeyUpdate`:

```
Interface IKeyUpdate{
      int getKey(in IndexKey ik, out Key k);
      int insertKey(in IndexKey ik, in Key
 newKey);
      int updateKey(in IndexKey ik, in Key
 newKey);
      int deleteKey(in IndexKey ik);
}
```

where the method `getKey` returns the value of the key associated with index `ik`. The method `updateKey` updates the key identified by the index `ik` with the value `newKey`. The methods `insertKey` and `deleteKey` respectively inserts a new key with index `ik` and key value `newKey`, and deletes the key identified by the index `ik`.

The *AuthKey* and *GenerateKey* component

The *AuthKey* and *GenerateKey* components depend on the `IKeyUpdate` interface in order to refresh the key. Furthermore, they use the `INetwork` interface to receive and send the new key respectively.

4.3 The Authenticated Loading Service

In order to support reconfigurability, components can be remotely uploaded into an embedded device after it has been deployed. We consider a scenario in which a device, referred to as the *Component Sender* (CS), provides components for remote uploading. As remote component uploading takes place through the wireless medium, an attacker has an easy game to modify a component or inject a fake one altogether. Therefore, upon remote loading, components need to be authenticated. This means that a component must be accompanied by a proof that the component originates from the trusted CS. Then, the other devices, referred to as *Component Sender* (CR), load only authenticated components.

The authenticated loading service is provided by the *ALoader* component framework both on CS and CR side. More in detail, *ALoader* on CS is responsible for broadcasting the new component and guaranteeing the component authenticity. The *ALoader* component framework on CR is responsible for downloading through the network, verifying the component authenticity, and finally loading the component from the memory buffer. In some applications, *ALoader* has to guarantee the component confidentiality. Usually, the components are broadcast through the network so that *ALoader* is connected to the *Network* component as depicted in Fig. 7.

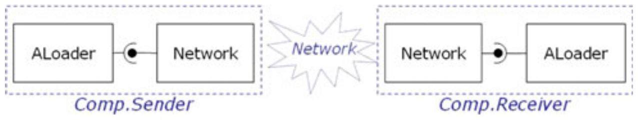

Fig. 7 The *ALoader* component framework

4.3.1 *The ALoader Component Framework*

The *ALoader* component framework includes a *AuthCS* component on CS side and a *AuthCR* component on CR side. The *AuthCS* component is responsible for generating the packets carrying the new component according to the authentication protocol. The *AuthCR* component buffers the component in memory and guarantees the component authenticity. Upon verifying the component authenticity, *AuthCR* calls the primitive of the operating system for loading the component stored in the local memory. The internal structure of both *AuthCR* and *AuthCS* strictly depends on the chosen authentication protocol.

In some scenarios, *ALoader* has to guarantee the confidentiality of packets carrying the component being loaded. So, *ALoader* includes a *SecCom* component as shown in Fig. 8. In order to guarantee the component confidentiality, *SecCom* use the keys stored in *KeyDB* component. Usually, *SecCom* is connected to the *Network* component to send/receive the component being loaded.

It is worthwhile to notice that *ALoader* does not preclude the use of public key encryption both in the *SecCom* and in the *AuthCR* component. It is only necessary that *CryptoPrimitive* components implementing this form of cryptography are available on the devices.

4.3.2 *Interfaces and Dependencies*

In this section, we define the interfaces and the dependencies of components that are included in the *ALoader* component framework.

The *AuthCR* component

The *AuthCR* depends on the *Network* interface and provides the `IAuthLoad` interface.

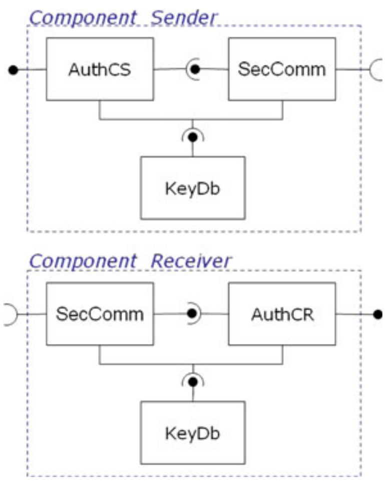

Fig. 8 Internal structure of *ALoader*

```
Interface IAuthLoad{
      load(in String cname, out ComponentType
 t);
}
```

The operation `load` downloads the component whose name is specified by the string `cname` and returns the component type.

The *AuthCS* component

The *AuthCS* depends on the `INetwork` interface and provides the `IAuthBrdcast` interface.

```
Interface IAuthBrdcast{
      send(in String cname, in ComponentType t,
 in Address a);
}
```

The operation `send` broadcasts the component whose name is specified by the string `cname` and the component type `t`. The receivers are specified by the address `a`.

5 Architecture Implementation

In this section we briefly describe two possible protocols for the Rekeying Service (Sect. 5.1) and the Authenticated Loading Service (Sect. 5.2), respectively. Then, we discuss a real implementation of the architecture for low-end, resource-poor TMote Sky devices (Sect. 5.3) [33].

5.1 The Rekeying Protocol

In our implementation we chose S^{2}RP, the Secure and Scalable Rekeying Protocol for devices with low-computational capabilities [11]. S^{2}RP guarantees the key authenticity by using only one-way hash functions that are computationally affordable even by the simplest devices. In short, the key authentication mechanism levers on *key-chains*, a technique based on the Lamport's one-time passwords [23]. A key-chain is an ordered set of symmetric keys so that each key is the hash preimage of the previous one. Hence, given a key in the key-chain, anybody can compute all the previous keys, but nobody can compute any of the next keys. Keys are revealed in the reverse order with respect to creation. Given an authenticated key in the key-chain, the devices can authenticate the next keys by simply applying a hash function.

In order to reduce the communication overhead, the Key Server (KS) maintains a tree structure of keys according to S^{2}RP (Fig. 9). Each leaf is associated with the symmetric *device-key* that the corresponding device secretly shares with KS. For each internal node, KS defines a key-chain and the node is associated with the last-revealed key. Let us refer to the last-revealed key associated with the node j as K_{l_j}, and to the next key that has to be revealed as K_{n_j}. Notice that after broadcasting K_{n_j}, the key K_{n_j} becomes the last-revealed key.

In addition to its device-key, every device stores the last-revealed key K_{l_j} associated with node j if the subtree rooted at the node j contains the leaf associated with the device-key. Hence, the key associated to the tree root is shared by all group members and it acts as the group-key. We call *device-keyring* the set of keys a devices stores.

When a device d leaves the group, KS has to revoke all the keys stored by d in order to guarantee the forward security. More in detail, KS defines the set of compromised internal nodes whose subtree contains the leaving device. Then, for each compromised node j and for each of its child i, KS broadcasts the next key of node j encrypted by using the last revealed key of node i. The keys are revealed in bottom-up order so that KS broadcasts a message containing the key K_{n_j} only if it has already broadcast all the messages containing the renewed keys of j's children

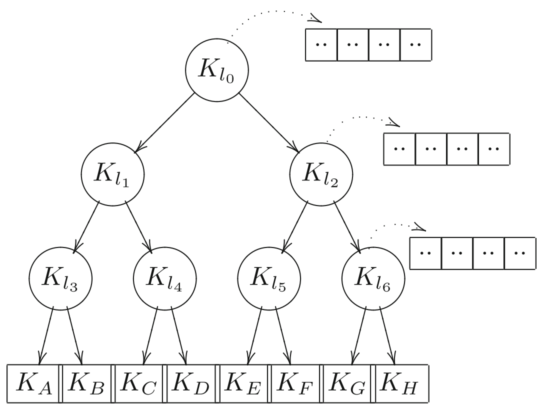

Fig. 9 Hierarchical structure of key-chains in S^{2}RP

nodes. The rekeying protocol is secure because the leaving node does not hold any of the keys used for rekeying. Furthermore it is scalable because KS has to broadcast $\mathcal{O}(\log n)$ messages where n is the number of devices. A deeper discussion about protocol security and performance can be found in the original paper [11].

The *Rekeying* Component Framework implements S^2RP as follows. With reference to Figs. 5 and 6, KS implements the Key Sender component framework whereas every device implements the Key Receiver component framework. In Key Sender, the *GenerateKey* component encapsulates the key tree whereas the *KeyDB* stores all the device-keys and the last-revealed keys of all internal nodes of the tree. The *SecCom* component is responsible to encrypt keying material as required by the S^2RP protocol. In the Key Receiver component framework, the *KeyDB* component stores the device-keyring whereas the *SecCom* is responsible to decrypt messages containing keying material according to the S^2RP protocol. Furthermore, the *AuthKey* component is responsible to authenticate keys according to the key chain mechanism.

5.2 The Component Authentication Protocol

A typical approach to authenticate a component upon downloading consists in authenticating it as a whole. However, this approach requires that a component receiver (CR) receives the entire component before verifying and this can be exploited by an adversary to mount a denial of service attack. More in detail, an attacker can make the device waste resources by making it buffer the whole component that in the end fails the authenticity verification. An alternative approach is based on the observation that a component is typically transmitted in several packets [19]. If every packet is authenticated, a device stores only authenticated material and reduces the risk of denial of service at minimum. Nevertheless, this solution introduces overhead as each packet needs to be authenticated.

A trade-off between security and performance can be achieved by authenticating *bursts* of packets by means of a technique based on Lamport's one-time passwords. A burst contains a fixed pre-defined number N_B of packets. If N is the total number of packets conveying the components, N_B is comprised between 1 and N. Bursts are transmitted sequentially. With reference to Fig. 10, each burst is linked to the next (transmitted) one by a one-way hash function. In fact, CS computes the hash of each burst and transmits the result with the previous burst. The hash value associated with the last transmitted burst is filled with the null value. If the receiver can authenticate the first burst, then it can sequentially authenticate all the subsequent bursts. Upon receiving a burst, the receiver computes the hash and compares it with the hash value conveyed by the previously

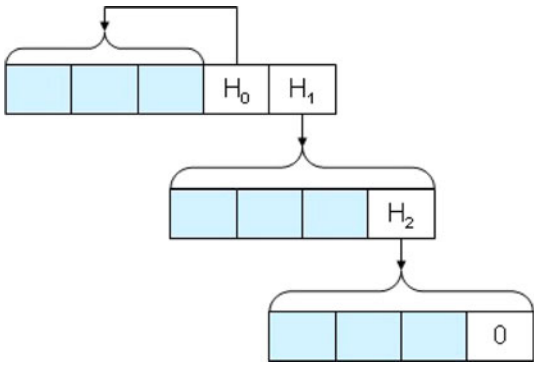

Fig. 10 A chain of bursts. The order of transmission of bursts is from top to bottom

received burst. If the two values are equal, the received burst is authentic.

The authenticity of the first burst must be proven in a different way. In a scenario with many receivers equipped with reduced computing capabilities, the digital signature might be not efficient. Therefore, we prove the authenticity of the first burst by means of a Message Authentication Code (MAC) computed with the pairwise key that a device secretly shares with CS, or with the group-key. Let us consider the example in Fig. 10. Given $N = 9$ and $N_B = 3$, each burst contains two component-packets and the hash of the next burst. The first burst also contains an authenticator constituted by a MAC computed with the secret key of the component receiver.

The proposed authentication scheme is secure because, in order to modify or inject a component, an adversary has to forge a MAC or one or more hashes. Under the assumption that the hash function is one-way, and that the MAC function is computationally secure, such a forgery is practically infeasible [31]. Furthermore, the scheme is also efficient in that it only stores authenticated bursts. Finally, It is also flexible because the number N_B, $1 \leq N_B \leq N$, of hash function computations as well as the authentication method for the first burst (MAC o digital signature) are design parameters.

This authenticated downloading protocol can be implemented in terms of the *ALoader* component framework as follows. With reference to Figs. 7 and 8, the *AuthCS* component is responsible for splitting a component into bursts and compute the related digests. The *AuthCR* component is responsible for verifying bursts authenticity and reconstruct the component from them. The *SecCom* component framework is responsible for the component confidentiality. The *KeyDB* stores the key for the MAC computation.

5.3 The Prototype

In this section we first describe the middleware layer on which we have implemented the security architecture

(Sect. 5.3.1). Then, we provide a performance evaluation of the prototype implementation (Sect. 5.3.2).

5.3.1 CTRK

The Component Run-Time Kernel (CTRK) hinges on the components, that are the basic runtime units of encapsulation and deployment. Components are instantiated at runtime from component types and can be deleted as well. Each component type can itself be dynamically loaded and unloaded at runtime and can be used to create multiple component instances. CRTK provides the basic operations for instantiating and removing a component instance, by invoking the `instantiate()` and `destroy()` operation. CRTK also provides the `load()` and `unload()` operations to load and unload a component type respectively.

Components offer their services through one or more interfaces and can have dependencies on other components. Dependencies are expressed in terms of one or more receptacles. Each component must have each receptacle of its connected to the interface of the corresponding component before it can be executed. Hence, when a components is deployed, all the components that provide the required interfaces are recursively deployed as well. The CRTK provides the `connect()` operation that creates a component, referred to as connector, implementing the connection between a receptacle and an interface. All the components reside inside a capsule which serves as a runtime component container, providing name space functionality. A capsule is typically implemented as an operating system address space.

CRTK has been instantiated on different types of hw/sw platforms. We have implemented our security architecture on two of them. One instantiation is in the Java Programming Language for devices of the laptop class. The other instantiation is on the operating systems Contiki [9, 10, 12] for low-cost, resource-poor devices of the TMote Sky class [33]. In fact, Contiki–CRTK is a lightweight and flexible operating system for tiny networked sensors and has a dynamic structure that allows to replace programs and drivers during runtime.

5.3.2 Implementation on TMote Sky Sensor Nodes

In this section we discuss the implementation of the security architecture on Contiki–CRTK for TMote Sky sensor nodes [33]. These sensor nodes are powered with two AA batteries and equipped with a 16-bit 8MHz MSP430 microcontroller, 48 KB of ROM, 10 KB of RAM, and IEEE 802.15.4 radio interface. We believe that such an implementation is more meaningful than the Java one on powerful devices, such as laptops and PDAs, as it proves that the architecture can scale down to even very low-end devices.

In our prototype, each sensor node implements an early prototype of *SecCom*, *Rekeying* and *ALoader* components on the receiver side. These security components exploit the services offered by the *CryptoPrimitive* components to perform cryptographic primitives, such as a symmetric cipher, a one-way hash function, or a hMAC. Our *CryptoPrimitive* components implement via software SkipJack as symmetric cipher [35], and SHA-1 as hash function [34]. However, another instance of *CryptoPrimitive* embeds and abstracts the cryptographic services offered by the radio chip CC2420 [7]. In fact, CC2420 features hardware IEEE 802.15.4 MAC security operations based on AES encryption using 128 bit keys. It implements CTR encryption and decryption, and Davies-Mayer hash function. In order to implement a cipher, the *CryptoPrimitive* component abstracts the CC2420 operating in the in-line mode by properly configuring it to provide the cryptographic transformations on the incoming/outgoing packets. Furthermore, the *CryptoPrimitive* uses the CC2420 in standalone mode as a cryptographic coprocessor to realize the hash function.

Table 1 reports the total amount of time (in milliseconds) employed by a *CryptoPrimitive* for encrypting a message or performing a hash function. More in detail, in the software implementation the CryptoPrimitive component requires 9.92 ms for encrypting a packet of 48 bytes by using SkipJack as symmetric cipher, whereas it requires 0.203 ms by using the CC2420 inline encryption. Furthermore, the *CryptoPrimitive* component requires 14.3 ms for applying the hash function SHA-1 on a packet of 28 bytes, whereas it requires 4.25 ms by using the CC2420 in standalone mode.

Table 2 reports the total amount of time employed by the security components in order to perform an operation. The second column contains the computational overhead with the software implementation of the CryptoPrimitives, whereas in the third column the CryptoPrimitives are implemented by using the CC2420 security operations. The table shows the computational overhead introduced by *SecCom* to guarantee the communication integrity and confidentiality. More in detail, *SecCom* performs the following transformation on the message `m:encrypt`

Table 1 Computational overhead of *CryptoPrimitive*

CryptoPrimitive	Time (ms)
SkipJack	9.92
SHA-1	14.3
AES	0.203
Davies-Mayer	4.25

Table 2 Computational overhead of security components

Components	SW *CryptoPrimitive*	HW *CryptoPrimitive*
SecCom	24.22 (ms)	4.45 (ms)
Rekeying	32.20 (ms)	–
ALoader	1.84 (s)	0.755 (ms)

`(m||hash(m))`, where `encrypt` specifies the symmetric encryption, `hash` specifies the hashing operation and || is the concatenation operation. The *SecCom* component requires 24.22 ms for a 28 bytes message by using software CryptoPrimitives, and 4.45 ms by using the CC2420. Furthermore, the table shows the time required by Rekeying in order to guarantee the key confidentiality and authenticity in accordance to the S^{2}RP protocol. It requires 32.2 ms by using SkipJack as symmetric cipher and SHA-1 as hash function. This component uses only the software CryptoPrimitives because in our implementation the CC2420 cannot use different cryptographic keys on the basis of the received messages. Finally, the *ALoader* component framework requires 1.84 s in order to authenticate a component of 1,264 bytes by using SHA-1 implemented via software as hash function. On the contrary, it requires 0.755 ms by using the CC2420 in standalone mode to perform the hash function.

6 Conclusions

In our research, we focus on security and reconfigurability in large-scale, heterogeneous NES that strictly inter-operate with the physical world. In order to be cost-effective and operational over time, NES are required to be reconfigurable in order to be adaptable to changing operating conditions. However, reconfigurability of NES introduces new security vulnerability.

With reference to such scenario, we have proposed a security architecture for reconfigurable NES applications. The proposed security architecture integrates and extends a reconfigurable, component-based middleware by means of abstractions and services for secure communication and reconfiguration. The proposed architecture has the following merits. First of all, it identifies a basic set of security abstractions and services, namely secure communication, authenticated downloading, and rekeying, that are fundamental for secure communication and reconfiguration in NES applications. Furthermore, it provides a component framework for each basic service so that every component has a well-defined interface and relationship with the other components. An application developer can instantiate the architecture by properly implementing each component framework according to the application needs. Finally, it can scale down even to very low-end devices such as Tmote Sky sensor nodes.

Acknowledgements This work has been partially supported by CHAT, "Control of Heterogeneous Automation Systems: Technologies for scalability, reconfigurability and security," funded by the European Commission under FP7 with contract number INFSO-ICT-224428; CONET, the Cooperating Objects Network of Excellence funded by the European Commission under FP7 with contract number FP7-2007-2-224053; and by Cassa di Risparmio di Pisa, Lucca e Livorno.

References

1. Tarek F. Abdelzaher, Brian M. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, Tian He, S. Krishnamurthy, L. Luo, Sang Hyuk Son, Jack Stankovic, R. Stoleru, and Anthony D. Wood, Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 582–589, Tokyo, Japan, 23–26 March 2004.
2. I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, Vol. 38, No. 4, pp. 293–422, 2002.
3. K. H. Årzén, A. Bicchi, G. Dini, S. Hailes, K. H. Johansson, J. Lygeros, and A. Tzes, A component-based approach to the design of networked control systems, *European Journal of Control*, Vol. 13, pp. 261–279, 2007.
4. G. Baliga and P. Kumar, A middleware for control over networks. In *Proceedings of the 44th IEEE Conference on Decision and Control*, 2005.
5. H. Chan and Adrian Perrig. Security and privacy in sensor networks, *IEEE Computer*, Vol. 36, No. 10, pp. 103–105, 2003.
6. Haowen Chan, V. D. Gligor, A. Perrig, and G. Muralidharan, On the distribution and revocation of cryptographic keys in sensor networks, *IEEE Transactions on Dependable and Secure Computing*, Vol. 2, No. 3, pp. 233–247, 2005.
7. Chipcon AS, CC2420–2.4GHz IEEE 802.15.4/ZigBee-ready RF Transceiver, http://www.chipcon.com
8. Paolo Costa, Geoff Coulson, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, and Stefanos Zachariadis, Reconfigurable component-based middleware for networked embedded systems, *International Journal on Wireless Information Systems*, Vol. 14, No. 2, pp. 149–162, 2007.
9. Paolo Costa, Geoff Coulson, Cecilia Mascolo, Gian Pietro Picco, and Stefanos Zachariadis, The RUNES middleware: a reconfigurable component-based approach to networked embedded systems. In *Proceedings of the 16th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'05)*, Vol. 2, pp. 806–810, Berlin, Germany, 11–14 September, 2005.
10. G. Dini and I. M. Savino, S^{2}rp: a secure and scalable rekeying protocol for wireless sensor networks. *Proceedings of the 3rd IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS'06)*, pp. 457–466, 9–12 October 2006.
11. A. Dunkels, B. Gronvall, and T. Voigt, Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pp. 455–462, Washington, DC, USA, 16–18 November 2004.
12. Adams Dunkels, Björn Grönvall, and Thiemo Voigt, Contiki—a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local*

Computer Networks (LCN'04), pp. 455–462, Tampa, FL, USA, 16–18 November 2004.
13. P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler, Securing the deluge network programming system. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, pp. 326–333. ACM, 2006.
14. Laurent Eschenauer and Virgil D. Gligor, A key-management scheme for distributed sensor networks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pp. 41–47, New York, NY, USA. ACM, 2002.
15. Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu, Agilla: a mobile agent middleware for self-adaptive wireless sensor networks. *ACM Transactions on Autonomous and Adaptive Systems*, Vol. 4, No. 3, 2009.
16. S. Graham and P. Kumar, editors. In *Proceedings of PWC 2003: Personal Wireless Communication*, Vol. 2775 of *Lecture Notes in Computer Science*, pp. 458–475, Chapter Convergence of Control, Communication, and Computation. Springer, Berlin, 2003.
17. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. Pister, System Architecture Directions for Networked Sensors. In *Proceedings of the 9th Symposium on Architectural Support to Programming Languages and Operating Systems (ASPLOS'00)*, pp. 93–104, Cambridge, MA, USA, November, 2000.
18. J. W. Hui and D. Culler, The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, pp. 81–94, Baltimore, MD, USA, 03–05 November 2004.
19. Chris Karlof, Naveen Sastry, and David Wagner. Tinysec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, pp. 162–175, Baltimore, MD, USA, 3–5 November 2004.
20. Philip Koopman, Embedded system security, *IEEE Computer*, Vol. 37, No. 7, pp. 95–97, 2004.
21. Lamport, L., Password authentication with insecure communication, *Communications of the ACM*, Vol. 24, No. 11, pp. 770–772, 1981.
22. LAN/MAN Standards Committee of the IEEE Computer Society, *IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)*, September 2006, revision of 2006.
23. P. Levis and D. E. Culler, Matè: a Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 85–95, San Jose, CA, 5–9 October 2002.
24. P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, The emergence of networking abstractions and techniques in Tiny OS. In *Proceedings of the 1st Symposium on Networked System Design and Implementation (NSDI'04)*, pp. 1–14, San Francisco, CA, USA, 2004.
25. Donggang Liu and Peng Ning, Multilevel μtesla: broadcast authentication for distributed sensor networks, *ACM Transaction on Embedded Computing Systems*, Vol. 3, No. 4, pp. 800–836, 2004.
26. Ting Liu and Margaret Martonosi, Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, San Diego, CA, USA, 11–13 June 2003.
27. Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, TinyDB: an acquisitional query processing system for sensor networks, *ACM Transactions on Database Systems*, Vol. 30, No. 1, pp. 122–173, 2005.
28. David J. Malan, Matt Welsh, and Michael D. Smith, Implementing public-key infrastructure for sensor networks, *ACM Transactions on Sensor Networks*, Vol. 4, No. 4, pp. 1–23, 2008.
29. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, Boca Raton, 1996.
30. Mohammad M. Molla and Sheikh Iqbal Ahamed, A survey of middleware for sensor network and challenges. In *Proccedings of 2006 International Conference on Parallel Processing—Workshops*, pp. 228–233, Columbus, OH, 14–18 August 2006.
31. Moteiv. Tmote Sky, http://www.moteiv.com.
32. National Institute of Standards and Technology, *FIPS PUB 180-1: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, April 1995.
33. National Institute of Standards and Technology (NIST), *SKIPJACK and KEA Algorithm Specifications*, 1998.
34. Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and Doug J. Tygar, SPINS: security protocols for sensor networks. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networks*, pp. 189–199, Rome, Italy, 16–21 July 2001.
35. Adrian Perrig, John Stankovic, and David Wagner, Security in wireless sensor networks, *Communications of the ACM*, Vol. 47, No. 6, pp. 53–57, 2004.
36. Neeli R. Prasad and Mahbubul Alam, Security framework for wireless sensor networks, *Wireless Personal Communications*, Vol. 37, No. 3–4, pp. 455—469, 2006.
37. S. Ravi, A. Raghunathan, and S. T. Chakradhar, Tamper resistance mechanisms for secure, embedded systems. In *VLSI Design*, 605 pp. IEEE Computer Society, Washington, DC, USA, 2004.
38. S. Ravi, A. Raghunathan, P.C. Kocher, and Hattangady S. Security in embedded systems: design challenges, *ACM Transactions on Embedded Computing Systems*, Vol. 3, No. 3, pp. 461–491, 2004.
39. R. L. Rivest, The RC5 encryption algorithm. In B. Preenel, editor, *Proceedings of the 2nd International Workshop on Fast Software Encryption*, Vol. LNCS 1008, pp. 86–96, Leuven, Belgium. Springer, Berlin, 14–16 December 1994.
40. Kay Römer, Oliver Kasten, and Friedemann Mattern, Middleware challenges for wireless sensor networks. *Mobile Computing and Communications Review*, Vol. 6, No. 4, pp. 59–61, 2002.
41. RUNES Consortium. Reconfigurable Ubiquitous Networked Embedded Systems (RUNES), European Commission, 6th Framework Programme, contract number IST-004536, http://www.ist-runes.org
42. Stefan Schmidt, Holger Krahn, Stefan Fischer, and Dietmar Watjen, A security architecture for mobile wireless sensor networks. In *Proceedings of the European Workshop on Security in Ad-hoc and Sensor Networks (ESAS'04)*, pp. 166–177, Lecture Notes in Computer Science No. 3313, Heidelberg, Germany, 6 August 2004. Springer, Berlin, 2005.
43. Hannu Sikkilä, Mikael Soini, Petri Oksa, Lauri Sydänheimo, and Markku Kivikoski, Kilavi wireless communication protocol for the building environment-security issues. In *Proceedings of the IEEE Tenth International Symposium on Consumer Electronics (ISCE'06)*, pp. 1–6, St. Petersburg, Russia, 28 June–01 July 2006.
44. B. Sinopoli, C. Sharp, Schenato L., S. Schaffert, and S. Sastry, Distributed control applications within sensor networks, *Proceedings of the IEEE*, Vol. 91, No. 8, pp. 1235–1246, 2003.
45. Michael N. K. Soini, Jana Van Greunen, Jan M. Rabaey, and Lauri T. Sydänheimo, Beyond sensor networks: Zuma middleware. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2007)*, pp. 4318–4323, Hong Kong, 2007.
46. Eduardo Souto, Germano Guimarães, Glauco Vasconcelos, Mardoqueu Vieira, Nelson S. Rosa, Carlos André Guimarães Ferraz, Eduardo Souto Judith Kelner, Germano Guimarães, Glauco Vasconcelos, Mardoqueu Vieira, Nelson S. Rosa, Carlos

André Guimarães Ferraz, and Judith Kelner, Mires: a publish/subscribe middleware for sensor networks, *Personal and Ubiquitous Computing*, Vol. 10, No. 1, pp. 37–44, 2006.
47. Miaomiao Wang, Jiannong Cao, Jing Li, and Sajal K. Das, Middleware for wireless sensor networks: a survey, *Journal of Computer Science and Technology*, Vol. 23, No. 3, pp. 305–326, 2008.
48. Yang Yu, Bhaskar Krishnamachari, and Viktor K. Prasanna, Issues in designing middleware for wireless sensor networks, *IEEE Network*, Vol. 18, No. 1, pp. 15–21, 2004.
49. Sencun Zhu, Sanjeev Setia, and Sushil Jajodia, Leap+: efficient security mechanisms for large-scale distributed sensor networks, *ACM Transactions on Sensor Networks*, Vol. 2, No. 4, pp. 500–528, 2006.
50. Zigbee alliance website, http://www.zigbee.org/en/index.asp.

Author Biographies

Gianluca Dini received the Laurea degree in electronic engineering from the University of Pisa in 1990 and a Ph.D. in computer engineering from Scuola Superiore S. Anna, Pisa, in 1995.Since 2000, he has been an associate professor of computer engineering at the University of Pisa.His main research interests are in distributed computing, with particular reference to security andfault tolerance.

Dr. Ida Maria Savino received the Laurea degree in computer engineering in 2004 from the Facultyof Engineering, University of Pisa. She received her PhD in computer engineering in 2008 from the same University. Currently, she is research fellow the Department of Ingegneria della Informazione, Pisa. Herresearch interests are in security of networked embedded systems.