STaR: a Reconfigurable and Transparent middleware for WSNs security

Roberta Daidone, Gianluca Dini, and Marco Tiloca

Department of Information Engineering
University of Pisa, Pisa, Italy
{roberta.daidone, gianluca.dini, marco.tiloca}@iet.unipi.it

Abstract. Wireless Sensor Networks (WSNs) are prone to security attacks. In order to protect the network from potential adversaries, it is necessary to secure communications between sensor nodes. Also, if we consider a network of heterogeneous objects including WSNs, security requirements may be far more complex. In particular, a single application may deal with multiple different traffic flows, each one of which may have different security requirements, that possibly change over time. In this paper, we present STaR, a software component which provides security transparency and reconfigurability for WSNs programming. STaR allows for securing multiple traffic flows at the same time according to specified security policies, and is totally transparent to the application, i.e. no changes to the original application or the communication protocol are required. Also, STaR can be easily reconfigured at runtime, thus coping with changes of security requirements. Finally, we present our preliminary implementation of STaR for Tmote Sky motes, and evaluate its impact on performance, in terms of memory occupancy, communication overhead, and energy consumption.

Keywords: WSNs, security, middleware

1 Introduction

In the recent years, Wireless Sensor Networks (WSNs) have received an increasing amount of attention and have been adopted in many application scenarios, from environmental to healthcare monitoring applications. In such scenarios, sensor nodes typically collect environmental data, and transmit them to a central base station through a wireless network. Sensor nodes are resource constrained devices that are deployed in unattended, possibly hostile environments.

Given the nature of WSNs, it is an easy task for an adversary to eavesdrop messages as well as alter or inject fake ones. It follows that secure communication is vital in order to assure messages confidentiality, integrity, and authenticity.

So far, deployment of WSNs have been used chiefly for scientific purposes, where an adversary has little incentive to attack the sensors [1]. As a result, notwithstanding the large body of academic research on WSNs security, only a few real deployments comprise security solutions.

Now things are changing. Recently, WSNs have been employed in Cooperating Objects Systems (COS) where mobile physical agents share the same environment to fulfill their tasks, either in group or in isolation [7, 22]. In this kind of systems, agents not only sense the environment, being a WSN the interface to the real world, but also act on it. Therefore, these COS become a tempting target for an adversary, because a security infringement may easily translate into a safety infringement, with possible consequences in terms of damages to things and injures to people. Similar considerations hold when WSNs are used in Critical Infrastructures [16, 18]. As an example, in [22] we consider a Highly Automated Airfield scenario where some static sensors monitor the airfield perimeter and communicate with an *Unmanned Ground Vehicle* (*UGV*) which is responsible for further investigations on alarms triggered by sensors. In such a scenario, it is fundamental to guarantee integrity and authenticity of messages exchanged within the network. In the asbsence of any alarm, it is reasonable to use a lightweight authentication mechanism. Instead, in the presence of an attack, it becomes necessary to increase the security level by introducing encryption. To supoport dymanic changes in a scenario like this one, reconfigurability is a mandatory feature for the security middleware.

In this paper, we present STaR, a modular, reconfigurable and transparent software component for secure communications in WSNs. STaR guarantees confidentiality, integrity, and authenticity by means of encryption and/or authentication. STaR is *modular* because it separates interfaces from their implementations. This makes STaR easily portable on different hardware [9, 19], system software [8, 27] and network stacks [14, 30]. Also, modularity makes it possible to easily load/unload different STaR modules to match different security requirements, add new features, or extend existing ones.

By means of STaR, it is possible to protect multiple traffic flows at the same time, according to different security policies. STaR is *reconfigurable* because it allows to change security policies on a per packet basis at runtime. That is, it assures a fine grained adaptability to possible changes in security requirements.

STaR is also *transparent*, because the application can still rely on the communication interface already in use. Also, the application does not have to be redesigned or recoded in order to exploit a certain security policy. This clearly separates the implementation of the application from the STaR component. STaR characteristics allow for easily reusing application components in application scenarios where security becomes relevant. Also, STaR allows unskilled people to secure their applications, by simply selecting security policies to be applied. Besides, application developers need neither to implement complex security procedures, nor to configure unfriendly tools, such as network firewalls.

We present our preliminary implementation of STaR [23] for TinyOS [27], and provide a performance evaluation on Tmote Sky motes [19], in terms of memory occupancy, communication overhead, and energy consumption. However, STaR features a generic architecture, and can be implemented for other hardware platforms and operating systems tailored to WSNs, such as Contiki [8] and ERIKA Enterprise [11].

The rest of this paper is organized as follows. In Section 2 we present some related work. Section 3 presents the STaR architecture. In Sections 4 and 5 we describe STaR interfaces providing communication and configuration services, respectively. Section 6 presents our prototype implementation of STaR for the TinyOS platform, and discusses our performance evaluation on Tmote Sky motes. In Section 7 we draw our conclusive remarks.

2 Related work

Many solutions have been devised for WSNs security, including [4] for secure communication, [3, 13, 15, 25, 29] for key management, and [21, 24] for secure code dissemination. Particular attention has been paid to component-based security architectures tailored to WSNs.

In [20] the authors propose a middleware that includes security. The tool is mostly focused on the opportunity to include security during application development. In [12] a reconfigurable security middleware is presented. The main difference between this work and STaR is that in STaR the security processing is part of the architecture itself, while in [12] security policies are preocessed by a module external to the middleware. Also, our work has been implemented and evaluated in terms of memory occupancy, network performance and power consumption. In [6] SMEPP Light is presented. It features group management, group level security policies and mechanisms for query injection and data collection for WSNs. SMEPP Light is tailored on subscribe/event scenarios. Also, it considers peers configured by means of XML which cannot dinamically change their behavior. Finally, SM-Sens [17] is a secure middleware that helps in bridging the gap between high-level application requirements and WSN. This middleware provides the application with an API to be used when introducing security.

3 STaR architecture

STaR assumes the presence of multiple *traffic flows*, and allows for securing them in different ways, according to different *security policies*. Possible security policies include packet encryption, packet authentication, or both of them.

STaR considers each application packet as belonging to one specific traffic flow. In order to do that, each packet belonging to a traffic flow f is associated to a specific *label* L_f. Also, each label refers to a specific security policy SP_f. Then, STaR processes every packet belonging to flow f according to the security policy SP_f associated to label L_f. As shown in Figure 1, the STaR component stays between the application and the rest of communication stack. STaR intercepts both incoming and outgoing traffic, segments it into flows, and secures them according to the corresponding security policies.

STaR assures *reconfigurability* by allowing users to dynamically change security policies. Furthermore, STaR provides *transparency* of security with respect to the application. That is, STaR exports the same interface as that of the underlying communication stack to the application. Therefore, in order to exploit

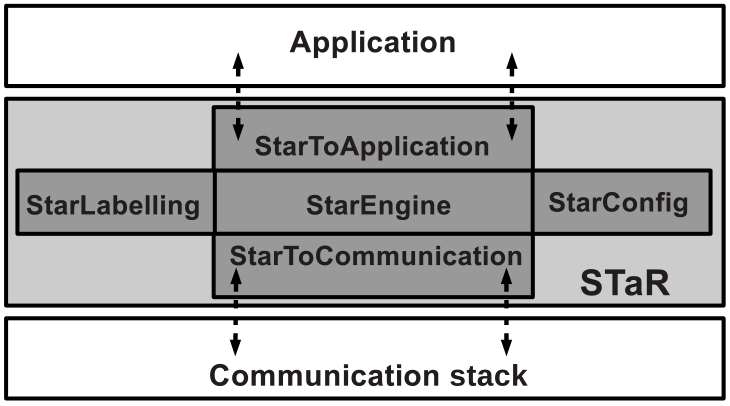

Fig. 1. STaR component overview.

the security services provided by STaR, it is not necessary to either re-design or re-code the application.

Fig. 2. Example of packet processed by STaR.

The STaR component consists in five sub-components, namely *StarToApplication*, *StarToCommunication*, *StarLabelling*, *StarConfig*, and *StarEngine*. The *StarToApplication* component provides the application with the same communication interface exported by the communication stack. The *StarToCommunication* component makes it possible to connect STaR to the underlying communication stack. The *StarLabelling* component classifies packets into *traffic flows*, and determines the associated label. The *StarConfig* component allows users to enable/disable security policies, and change their association to traffic flows, thus providing reconfigurability at runtime. Finally, the *StarEngine* component actually processes packets, according to the security policy associated to the traffic flow they belong to. Figure 2 shows a packet processed by STaR, with the *Label* field prepended to the packet payload. So far, the *StarEngine* component relies on standard security algorithms and does not consider any key management mechanism. We assume that key management is provided by an external component. In other words, STaR focuses on securing communication, assuming keys have been already deployed and are properly managed.

Note that the modularity of STaR simplifies the porting of STaR onto different communication stacks. Actually, a different communication stack requires to customize the *StarToCommunication* and *StarToApplication* components, whereas the other components remain unmodified.

Although the application developer is not required to change the application code and/or behavior, he/she has certain obligations in order to exploit STaR, namely i) implementation of security policieres; ii) traffic segmentation; iii) association of security policies to traffic segments; and, finally, iv) STaR initialization. We deal with these issues in the next sections.

4 Secure communication services

As described in Section 3, STaR assumes that each packet belongs to a specific traffic flow. That is, each packet is logically associated to a specific label that represents a particular traffic flow.

STaR relies on packet labels in order to protect multiple traffic flows at the same time. All packets belonging to a given packet flow can be associated to a common label, and secured before transmission, according to a specified security policy. Conversely, incoming packets can be unsecured upon being received, according to the security policy associated to the traffic flow they belong to.

STaR is responsible for both securing/unsecuring packets and mapping flow labels into security policies. As shown in Section 3, these tasks are totally transparent to the application. In fact, the application can still rely on the original communication interface provided by the available communication stack, and does not require to be modified. In order to manage associations between traffic flows and security policies, STaR maintains two tables: i) a *Security Policy Table* (*SPT*), and ii) a *PolicyDB*.

The *SPT* is formatted as follows. The *Label* field is one byte in size and can range from 0 to 255. That is, STaR is able to manage up to 256 different traffic flows at the same time. The *PolicyID* field specifies the security policy to be adopted for a given traffic flow. *PolicyID* entries in the *SPT* refer to security policies specified in the *PolicyDB* by the specific STaR implementation. Finally, the *Active* field indicates whether the security policy associated to a given label has to be applied or not to packets belonging to such traffic flow. The *Active* field is set to TRUE by default in all entries. Also, *SPT*s of all network nodes are supposed to be initialized in the same way at the network startup. In order to manage the *SPT* at runtime, STaR provides the user with a set of configuration functions, which are described in Section 5. Table 1 shows an example of *SPT*. Note that different labels can be associated to the same security policy. That is, packets belonging to different traffic flows can be secured in the same way.

The *PolicyDB* is formatted as follows. *PolicyID* values in the *PolicyDB* have to match *PolicyID* entries of the *SPT*, in order to correctly retrieve the security policy implementation provided by STaR. The *EntryPoint* field contains a reference to the code section which implements the policy (e.g. a C++ function pointer). If we consider an environment which allows for dynamically loading new security modules [8], the set of available security policies can be changed dynamically. Otherwise, if we consider a static environment like TinyOS [27], also such a set must be statically initialized before devices bootstrap takes place.

Label	PolicyID	Active
0	SP005	TRUE
1	SP013	TRUE
2	SP152	FALSE
...	...	...
254	SP152	TRUE
255	SP020	FALSE

Table 1. Example of Security Policy Table.

PolicyID	EntryPoint
SP005	errCode (*encrypt)(buffer)
...	...
SP152	errCode (*authenticate)(buffer)

Table 2. Example of PolicyDB Table.

Table 2 shows an example of *PolicyDB*. As the *SPT*, *PolicyDB*s of all network nodes are supposed to be initialized in the same way at the network startup. Note that the *EntryPoint* field format changes according to the specific STaR implementation and can be considered the bridge between the *StarConfig* and the *StarEngine* components.

4.1 STaR communication support

STaR provides the user with four communication functions, namely *send*, *receive*, *retrieveLabel*, and *retrievePolicy*. In the following, we describe such functions in terms of their parameters and behavior.

```
bool send(packet, size);
```

Provide *packet* of size *size* to STaR. Return TRUE in case of success.

```
bool receive(packet, size);
```

Provide *packet* of size *size* to the application. Return TRUE in case of success.

These two functions are responsible for communication between the application and the lower layers. Note that the signatures reported above change their implementation according to the adopted communication protocol. As specified in Section 3, the application developer has to implement both traffic segmentation into flows and the association between security policies and traffic flows.

The *retrieveLabel* function implements traffic segmentation whereas the operation *retrievePolicy* implements the association between security policies and traffic flows as follows.

```
int retrieveLabel(packet);
```

Return the label associated to the traffic flow which *packet* belongs to.

```
Policy retrievePolicy(label);
```

Return the security policy associated to the label *label*. Firstly, access the *SPT* to retrieve the *PolicyID* associated to label *label*. Then, access the *PolicyDB* to retrieve the policy. Return an error code in case the *Active* field in the *SPT* is set to FALSE, or the policy is not present in the *PolicyDB*.

The application developer must determine the best security policy to protect each traffic flow, and bind each one of them to a specific label value. Specifically, the *retrieveLabel* function must implement the criteria according to which it is possible to infer which traffic flow a given packet belongs to.

Of course, a base version of *retrieveLabel* can behave according to a default set of criteria. For instance, it can consider all packets as belonging to a single common traffic flow, and associate them a common label value. By doing so, all packets would be protected according to the same security policy.

4.2 STaR communication scheme

In the presence of STaR, the transmission of a packet P takes place according to the following steps. First, the application provides STaR with packet P, through the *send* function. Then, STaR retrieves the label L associated to packet P through the *retrieveLabel* function, and the associated security policy SP through the *retrievePolicy* function.

After that, STaR builds a one byte field, fills it with the label L, and inserts it between the header and the payload of packet P. Then, packet P is secured, according to the security policy SP. Finally, STaR provides the secured packet P to the communication stack, and delivers it to the scheduled recipient nodes.

Note that the additional label byte must never be encrypted, in order to assure that packet P is correctly unsecured at the recipient side. However, the label byte can be authenticated, in order to guarantee that it has been actually generated by the STaR component. Figure 3(a) shows how an outgoing packet is processed in the presence of STaR.

Conversely, the reception of a packet P takes place according to the following steps. STaR receives the secured packet P from the communication stack, and retrieves the label L from the additional label byte, which can then be removed.

After that, STaR retrieves the security policy SP associated to label L, through the *retrievePolicy* function, and unsecures packet P, according to SP. Finally, STaR provides the unsecured packet to the application, that receives it through the *receive* function. Figure 3(b) shows how an incoming packet is processed in the presence of STaR.

5 STaR configuration services

STaR allows users to dynamically change security settings at runtime, and provides a specific *configuration interface* aimed at changing how security policies are used, as well as their association to traffic flows.

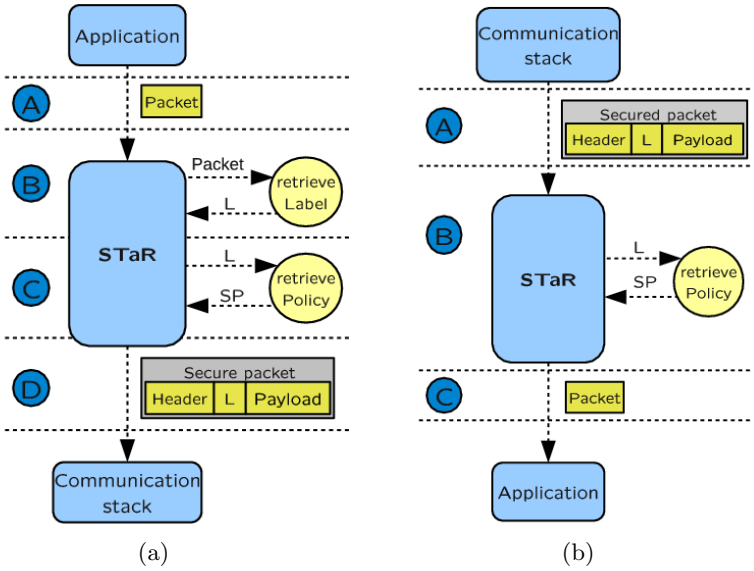

Fig. 3. a) Outgoing packet processing and b) incoming packet processing.

If the operating system allows for changing some program modules at runtime, it is possible to add and remove policies and traffic flows, in order to match new security requirements more effectively. STaR provides seven configuration functions, namely *enablePolicy*, *disablePolicy*, *changePolicy*, *addPolicy*, *removePolicy*, *addFlow*, and *removeFlow*. In the following, we describe such functions as to their parameters and behavior.

```
void enablePolicy(label);
```

Set to TRUE the *Active* field of the *SPT* entry related to label *label*. Then STaR starts applying such security policy to all packets belonging to the traffic flow associated to label *label*.

```
void disablePolicy(label);
```

Set to FALSE the *Active* field of the *SPT* entry related to label *label*. Then STaR stops applying such security policy to all packets belonging to the traffic flow associated to label *label*.

```
void changePolicy(label, newPolicy);
```

Write *newPolicy* in the *PolicyID* field of the *SPT* entry related to label *label*. The *Active* field of the *SPT* entry remains unchanged.

Thanks to the STaR configuration interface, the application is allowed to change security settings at runtime. To change a security policy, a specific control message is needed. Such a message is injected into the network as a common

message, but belongs to a specific flow used for STaR management. Messages belonging to this flow are processed by STaR in a special manner and not forwarded to the rest of communication stack, because their purpose is to change the behavior of the *StarEngine* component.

Also, STaR can dynamically change its behavior even in software platforms which does not allow for dynamically loading/unloading modules, such as TinyOS [27]. This is possible by filling all the implemented *SPT* entries and activating/deactivating them, by simply calling the configuration interface functions.

The following four functions can be implemented only if the operating system allows for dinamically changing the program loaded on sensor nodes.

```
void addPolicy(PolicyID, EntryPoint);
```

Add the policy identified by *PolicyID* to the *PolicyDB*. *EntryPoint* specifies the code section to be executed to apply the specified policy.

```
void removePolicy(PolicyID);
```

Remove the policy identified by *PolicyID* from the *PolicyDB*.

```
void addFlow(label);
```

Add a flow with ID *label* to the *SPT*. Firstly, verify it is not a copy of another flow, then set the *PolicyID* field to UNDEFINED and the *Active* field to FALSE. These fields will be set by a policy association.

```
void removeFlow(label);
```

Remove the flow identified by *label* from the *SPT*.

6 STaR TinyOS implementation

We implemented the STaR component [23] for TinyOS 2.1.1, which is currently available at [27]. We have implemented the security features described in Section 4 and Section 5, with reference to the Tmote Sky motes [19] and the CC2420 chipset [26]. At the moment, we have implemented the *Skipjack* encryption module [28] and the *SHA-1* module for integrity hashing [10], and are working to allow users to choose among more standard security protocols.

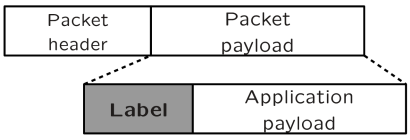

Fig. 4. Application packet modified by STaR.

Figure 4 shows an example of secured application packet. STaR inserts the *Label* field in the application packet, between the packet header and the original payload. Note that, thanks to STaR, application developers can secure application packets without having a deep security knowledge. In fact, they are just required to i) know the meaning of each policy; ii) associate each traffic flow to the chosen policy; and, of course, iii) include the STaR component in the source code. As a consequence, STaR allows security non-experts to secure applications in a simple and reconfigurable manner, without changing either the application or the communication protocol.

6.1 STaR memory footprint

The amount of ROM memory available on Tmote Sky motes is 48 kB, and may represent a severe constraint while dealing with complex modules like those composing STaR.

In order to evaluate memory consumption on Tmote Sky motes, we considered the TinyOS image size wiring the STaR submodules separately.

- S is the image size in bytes of the original TinyOS stack and the *RadioCountToLeds* application, which is one of the demo applications provided with TinyOS.
- $\hat{C} = S + C$ is the image size in bytes of the original TinyOS stack and the *RadioCountToLeds* application (S), wired to the *StarConfig* module (C). $C = \hat{C} - S$ is the memory occupancy of the *StarConfig* module.
- $\hat{E} = S + C + E$ is the image size in bytes of the original TinyOS stack and the *RadioCountToLeds* application (S), wired to the *StarConfig* module (C) and the *Skipjack* submodule of the *StarEngine* module (E). $E = \hat{E} - \hat{C}$ is the memory occupancy of the *Skipjack* submodule of the *StarEngine* module.
- $\hat{A} = S + C + A$ is the image size in bytes of the original TinyOS stack and the *RadioCountToLeds* application (S), wired to the *StarConfig* module (C) and the *SHA-1* submodule of the *StarEngine* module (A). $A = \hat{A} - \hat{C}$ is the memory occupancy of the *SHA-1* submodule of the *StarEngine* module.

	Memory occupancy (B)	**Memory occupancy (%)**
Application and TinyOS stack	13372	27.86
StarConfig	854	1.78
StarEngine (Skipjack)	2046	4.26
StarEngine (SHA-1)	3900	8.12
Available memory	27828	57.98

Table 3. Detailed memory occupancy.

Table 3 provides more detailed information on memory occupancy. As explained above, we considered the original TinyOS stack together with the *RadioCountToLeds* application, each STaR module separately, and the amount of memory still available. If we sum the contributions of the *StarConfig*, *Skipjack* and *SHA-1* modules, we observe that our STaR implementation totally requires the 14.16% of the overall memory available on a Tmote Sky mote. Since the application together with the TinyOS stack requires the 27.86% of the available memory, we have the 57.98% of 48 kB still available for other uses. We believe that the amount of memory required by STaR is reasonable with respect to the available memory. Also, STaR modular implementation allows for saving memory by loading only a few of the available modules, provided that the specific software platform makes it possible, and it is well known what modules are needed.

6.2 STaR performance evaluation

In our analysis, we assumed that STaR is operating on top of the 2.4 GHz physical layer, with a 250 Kb/s bit rate [26]. Then, we modeled the impact of security considering two main aspects: i) the network performance degradation due to security processing and extra trasmissions overhead, and ii) the extra energy consumption, due to extra processing and extra transmissions.

We evaluated the security processing overhead by means of experiments, while the extra transmission overhead has been computed analytically, considering the bit rate and the packet size. Also, energy consumption has been evaluated analytically, considering single energy contributions.

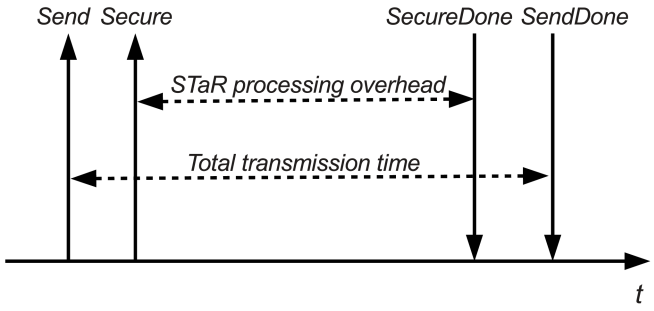

Fig. 5. Send and Secure events nesting.

Figure 5 shows the sequence of events which take place when we transmit a secured packet with STaR, according to the TinyOS Send/SendDone schema. The time interval named *STaR processing overhead* is the extra time required to process the packet according to the chosen security policy. This time has been evaluated experimentally.

Policy	$\mathbf{d_{proc}}$ (μs)	**Standard deviation** (μs)
NONE	142	0
ENC	1239.70	1.96
HASH	32853.50	2.53
ENC + AUTH	33948.65	3.01

Table 4. STaR d_{proc} contributions overview.

In our experiments, we observed one sender device at a time transmitting secured packets whose payload is 8 bytes in size. In order to increase the accuracy of our results, we performed 10 repetitions of 20 transmissions for each experiment. The results shown in Table 4 are averaged over all the different repetitions. We also report the standard deviation we derived from the independent replication method.

The first line shows the delay associated to the *NONE* policy, i.e. packets just cross the STaR component, which adds the label corresponding to neither encryption nor authetication. In this case we have a constant delay because STaR always performs the same simple sequence of operations.

The second line shows the delay associated to the *ENC* policy. In this case packets are labelled in order to be encrypted. Then, the label is recognized by the *StarEngine* module, which encrypts packets using *Skipjack*. We notice a considerable increase in the delay due to the block encryption operations. If we consider *Cipher-block chaining* (*CBC*) *Skipjack*, we can use this algorithm also for authentication.

The third line shows the delay associated to the *HASH* policy. In this case, packets are labelled in order to be hashed. Then, the label is recognized by the *StarEngine* module, which applies *SHA-1* on packets.

Finally, the fourth line shows the delay associated to the *ENC + AUTH* policy. In this case, packets are labelled in order to be encrypted by *Skipjack* as well as authenticated by means of *SHA-1* hashing. This delay should be close to $d_{ENC} + d_{HASH} - d_{NONE}$ because it combines the behaviors of the encryption policy and the hashing policy. We subtract the delay due to labelling (d_{NONE}), because we consider it twice when summing d_{ENC} and d_{HASH}, but it is actually performed just once.

If we consider values in Table 4 we have $d_{ENC} + d_{HASH} - d_{NONE} = 33951.2\ \mu$s with a standard deviation of $3.20\ \mu$s. Thus, we think the reported delay is coherent, and the error acceptable. Note that considerable delays are due to the standard encryption and authentication algorithms, while the actual STaR contribution to the processing delay is just 142 μs. This is the additional time required to add the label field to packets, thus assuring that each one of them is correctly processed according to the associated security policy. We believe that this delay is negligible if compared to the ones introduced by standard cryptographic computations, such as those performed by *Skipjack* and *SHA-1*.

The transmission overhead has been evaluated analytically, with a bit rate equal to 250 Kb/s [26]. Specifically, we have considered the time required to transmit the additional bytes added by STaR, according to the specific security policy. The size of the original application packet, including the header and the *Cyclic Redundancy Check* (*CRC*) is 21 bytes. We computed the time d_{tx} required to transmit the original application packet as the ratio between the packet size in bits and the bit-rate: $d_{tx} = \frac{21 \times 8}{0.250} = 672\ \mu s$.

Policy	$\mathbf{d_{tx}}$ (μs)	**Increase (%)**
NONE	32	4.76
ENC	32	4.76
HASH	672	100
ENC + AUTH	672	100

Table 5. STaR d_{tx} contributions overview.

Table 5 provides an overview of the transmission overhead, considering different security policies. Note that the extra transmission delay of the *ENC* policy is 32 μs if the packet payload is a multiple of the encryption block size (8 bytes, in our case). In this case, the packet is just extended with the one byte label. Instead, if we use a block encryption scheme which requires padding and the payload size is not a multiple of the encryption block size, the packet payload has to be padded in order to have an overall size which is a multiple of the encryption block size. The maximum padding size is 8 bytes, which would result in an additional 256 μs delay. In order to avoid this, it is possible to adopt a block encryption scheme like *Ciphertext Stealing* (*CTS*) [2], which does not rely on padding.

As already observed for the processing overhead, the considerable delays of the *HASH* and *ENC + AUTH* policies are due to the standard *SHA-1* hashing output size, which is 20 bytes long. In fact, the actual STaR contribution to the transmission delay is just 32 μs, that is the time required to transmit the one byte label field added to the original packet. We believe that this delay is affordable, since it is due to the increase of just one byte to the original packet size.

On the other hand, if the application developer finds unaffordable a 100% increase in the transmission delay of the *HASH* and *ENC + AUTH* policies, it is possible to define security policies which truncate the hashing field to 4, 8 or 16 bytes in size. This would save 512 μs, 384 μs and 256 μs during packet transmission, respectively. Hash field truncation is a widely adopted method in WSNs, because it allows for increasing performance without serious risks of collisions [5, 14].

As to energy consumption, we considered that each contribution has the form $\mathcal{E}_i = \mathcal{P}_i \times d_i$. We define d_i the delay due to the considered operation i. $\mathcal{P}_i = V_i \times I_i$ is the single power contribution, expressed as the product between

voltage and current of the MSP430 microcontroller and the CC2420 chipset, which are responsible for processing and transmission, respectively [19].

Policy	**Processing** $\mathcal{P}_{\text{proc}} = 1.08\text{mW}$		**Transmission** $\mathcal{P}_{\text{tx}} = 31.32\text{mW}$	
	d_{proc} (μs)	$\mathcal{E}_{\text{proc}}$ (nJ)	d_{tx} (μs)	$\mathcal{E}_{\text{tx}}$ (nJ)
NONE	142	153.4	32	1002.2
ENC	1239.7	1338.9	32	1002.2
HASH	32853.5	35481.8	672	21047.0
ENC + AUTH	33948.7	36664.6	672	21047.0

Table 6. STaR energy consumption contributions.

Table 6 provides an overview of such contributions. Considerable increases in energy consumption per packet are due to the standard encryption and authentication algorithms, while the actual STaR contribution to energy consumption is the one reported in the *NONE* policy entry: $\mathcal{E}_{\text{proc}} + \mathcal{E}_{\text{tx}} = 1155.6\,\text{nJ}$ that is the energy consumed to add the label field to the original packet and transmit it. We believe that this additional energy consumption is affordable, if compared to the contributions introduced by standard security mechanisms like *SHA-1*.

7 Conclusion

We have presented STaR, our security software component for WSNs. It allows for protecting multiple traffic flows at the same time, according to different security policies. STaR is transparent to the application, which can rely on the same communication interface already in use. Also, STaR allows the user to reconfigure security policies and their association to traffic flows at runtime. Finally, we have considered our preliminary implementation of STaR for Tmote Sky motes, and provided a performance evaluation in terms of memory occupancy, communication overhead, and energy consumption. Our results show that STaR is efficient as well as affordable even in the considered resource scarce hardware platform. In fact, the heaviest impact on performance is due to the adopted standard security algorithms, and not to the presence of STaR. Future works will extend STaR interfaces and services, and aim at implementing and evaluating STaR for different architectures and platforms. Also we will include a Key Manager component for distribution and management of cryptographic keys.

Acknowledgment

This work has been supported by the EU FP7 Network of Excellence CONET (Grant Agreement no. FP7-224053); the EU FP7 Integrated Project PLANET

(Grant agreement no. FP7-257649); the TENACE PRIN (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research; and the Regione Toscana POR CReO 2007 - 2013, LINEA DI INTERVENTO 1.5.a - 1.6, BANDO UNICO R&S ANNO 2012, Piattaforma Integrata per la Gestione delle Operazioni Aeroportuali - PITAGORA. We would also like to thank Davide Di Baccio for his help during the implementation phase of our work.

References

1. A. A. Cardenas, T. Roosta and S. Sastry: Rethinking security properties, threat models, and the design space in sensor networks: a case study in SCADA systems. Ad Hoc Networks 7(8), 1434–1447 (2009)
2. A.J. Menezes, P.C. van Oorschot and S.A. Vanstone: Handbook of Applied Cryptography. CRC Press (2001)
3. C. K. Wong, M. Gouda and S.S. Lam: Secure group communications using key graphs. IEEE_J_NET 8(1), 16–30 (February 2000)
4. C. Karlof, D. Sastry and D. Wagner: Tinysec: a link layer security architecture for wireless sensor networks. In: Proceedings of the 2nd international conference on Embedded networked sensor systems. pp. 162–175. SenSys '04, ACM, New York, NY, USA (2004)
5. C. Karlof, D. Sastry and D. Wagner: TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In: Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004). pp. 162–175 (November 2004)
6. C. Vairo, M. Albano and S. Chessa: A secure middleware for wireless sensor networks. In: Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services. pp. 59:1–59:6. Mobiquitous '08, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium (2008)
7. CONET: Cooperating objects network of excellence, european commission, 7th framework programme, grant agreement n. 224053. http://www.cooperating-objects.eu/ (2008)
8. Contiki: Contiki: The open source operating system for the internet of things. http://www.contiki-os.org/ (2012)
9. Crossbow Technology Inc.: MPR-MIB Users Manual (June 2007), http://bullseye.xbow.com:81/Support/
10. D. Eastlake and P. Jones: RFC 3174 - US Secure Hash Algorithm 1 (SHA1) (September 2001), http://tools.ietf.org/html/rfc3174
11. ERIKA Enterprise: Erika Enterprise and RT-Druid (2009)
12. G Dini and I.M. Savino: A Security Architecture for Reconfigurable Networked Embedded Systems. International Journal of Wireless Information Networks 17, 11–25 (2010)
13. G. Dini and M. Tiloca: Considerations on Security in ZigBee Networks. In: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing. pp. 58–65 (June 2010)
14. Institute of Electrical and Electronics Engineers, Inc.: IEEE Std. 802.15.4-2006, 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 (WPANs). New York (September 2006)

15. J. Maerien, S. Michiels, C. Huygens and W. Joosen: MASY: MAnagement of Secret keYs for federated mobile wireless sensor networks. In: Proceedings of the 6th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications. pp. 121–128 (October 2010)
16. L. Buttyan, D. Gessner, A. Hessler and P. Langendoerfer: Application of wireless sensor networks in critical infrastructure protection: challenges and design options [security and privacy in emerging wireless networks]. Wireless Communications, IEEE 17(5), 44–49 (October 2010)
17. L.H. Freitas, K.A. Bispo, N.S. Rosa and P.R.F. Cunha: SM-Sens: Security middleware for Wireless Sensor Networks. In: Proceedings of the Information Infrastructure Symposium, 2009. Global. pp. 1–7. GIIS '09 (June 2009)
18. M. Albano, S. Chessa and R. Di Pietro: Information assurance in critical infrastructures via wireless sensor networks. In: Proceedings of the 4th International Conference on Information Assurance and Security. pp. 305–310. ISIAS '08 (2008)
19. Moteiv Corporation: Tmote iv low power wireless sensor module (November 2006), http://www.snm.ethz.ch/snmwiki/pub/uploads/Projects/tmote_sky_datasheet.pdf
20. N. Matthys, C. Huygens, D. Hughes, S. Michiels and W. Joosen: A Component and Policy-Based Approach for Efficient Sensor Network Reconfiguration. In: Proceedings of the 2012 IEEE International Symposium on Policies for Distributed Systems and Networks. pp. 53–60 (July 2012)
21. P.E. Lanigan, R. Gandhi and P. Narasimhan: Sluice: Secure Dissemination of Code Updates in Sensor Networks. In: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems. pp. 53–62 (July 2006)
22. PLANET: Platform for the deployment and operation of heterogeneous networked cooperating objects, european commission, 7th framework programme, grant agreement n. 257649. http://www.planet-ict.eu/ (2010)
23. R. Daidone, G. Dini and M. Tiloca: STaR source code. http://www.iet.unipi.it/g.dini/download/code/star.zip (March 2013)
24. S. Hyun, P. Ning, A. Liu and W. Du: Seluge: Secure and DoS-Resistant Code Dissemination in Wireless Sensor Networks. In: Proceedings of the 2008 International Conference on Information Processing in Sensor Networks. pp. 445–456 (April 2008)
25. S. Zhong, L. Chuang, R. Fengyuan, J. Yixin and C. Xiaowen: An efficient scheme for secure communication in large-scale wireless sensor networks. In: Communications and Mobile Computing. WRI International Conference on. CMC '09, vol. 3, pp. 333–337 (January 2009)
26. Texas Instruments: Texas instruments cc2420 2.4 ghz ieee 802.15.4 / zigbee ready rf transceiver (2012), http://focus.ti.com/lit/ds/symlink/cc2420.pdf
27. TinyOS Working Group: Tinyos home page. http://www.tinyos.net/ (2012), http://www.tinyos.net/
28. U.S. National Security Agency (NSA): SKIPJACK and KEA algorithm specifications (May 1998), http://csrc.nist.gov/groups/ST/toolkit/documents/skipjack/skipjack.pdf
29. W. Gu, N. Dutta, S. Chellappan and B. Xiaole: Providing end-to-end secure communications in wireless sensor networks. Network and Service Management, IEEE Transactions on 8(3), 205–218 (September 2011)
30. ZigBee Alliance: ZigBee Document 053474r17, ZigBee Specification. ZigBee Alliance (January 2008)