STaR: Security Transparency and Reconfigurability for Wireless Sensor Networks programming

Roberta Daidone[1], Gianluca Dini[1] and Marco Tiloca[1]
[1]*Dipartimento di Ingegneria dell'Informazione, University of Pisa, Pisa, Italy*
{*r.daidone, g.dini, m.tiloca*}*@iet.unipi.it*

Keywords: Security:Wireless Sensor Networks

Abstract: Wireless Sensor Networks (WSNs) are prone to security attacks. To protect the network from potential adversaries, it is necessary to secure communications between sensor nodes. If we consider a network of heterogeneous objects including WSNs, security requirements may be far more complex. A single application may deal with 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 transparent to the application, i.e. no changes to the original application or the communication protocol are required. STaR can be easily reconfigured at runtime, thus coping with changes of security requirements. Finally, we present our implementation of STaR for Tmote Sky motes, and evaluate it in terms of memory occupancy, communication overhead, and energy consumption.

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 collect environmental data, and transmit them to a central base station through a wireless network. Sensor nodes are typically resource constrained devices deployed in unattended, possibly hostile environments. Given the nature of WSNs, it is an easy task for an adversary to eavesdrop messages and alter or inject fake ones. It follows that secure communication is vital to assure messages confidentiality, integrity, and authenticity.

WSNs have been used chiefly for scientific purposes, where an adversary has little incentive to attack sensors (Cardenas, A.A., Roosta, T., and Sastry, S., 2009). Now WSNs are employed also in Cooperating Objects Systems (COS) where mobile physical agents share the same environment to fulfill their tasks, either in group or in isolation (CONET, 2008; PLANET, 2010). Such agents not only sense the environment, but also act on it. COS are a tempting target for an adversary, since a security infringement may easily translate into a safety one, with possible consequences in terms of damages to things and injures to people. Same considerations hold for WSNs in Critical Infrastructures (Albano, M., Chessa, S., and Di Pietro, R., 2008; Buttyan, L., Gessner, D., Hessler, A., and Langendoerfer, P., 2010).

Many solutions have been devised for WSNs security, including (Karlof, C., Sastry, N., and Wagner, D., 2004) for secure communication, (Wong, C.K., Gouda, M., and Lam, S.S., 2000; Su, Z., Lin, C., Ren, F., Jiang, Y., and Chu, X., 2009; Gu, W., Dutta, N., Chellappan, S., and Xiaole, B., 2011; Maerien, J., Michiels, S., Huygens, C., and Joosen, W., 2010; Dini, G., and Tiloca, M., 2010) for key management, and (Hyun, S., Ning, P., Liu, A., and Du, W., 2008; Lanigan, P.E., and Gandhi, R., and Narasimhan, P., 2006) for secure code dissemination. Particular attention has been paid to component-based security architectures tailored to WSNs (Matthys, N., Huygens, C., Hughes, D., Michiels, S., and Joosen, W., 2012; Dini, G., and Savino, I.M., 2010).

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 it easily portable on different hardware (Moteiv Corporation, 2006; Crossbow Technology Inc., 2004) and network stacks (IEEE, 2006; ZigBee Alliance, 2008). Modularity allows for loading/unloading STaR modules to match security requirements, add new features, or extend existing ones.

STaR allows for protecting multiple traffic flows at the same time, according to different security policies.

STaR is *reconfigurable* because it makes it possible 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 *transparent*, because the application can still rely on the communication interface already in use. The application does not require 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 allows for reusing application components in 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 evaluated our STaR implementation (Daidone, R., Dini, G., and Tiloca, M., 2012) for TinyOS (TinyOS Working Group, 2012) on Tmote Sky motes (Moteiv Corporation, 2006), showing it is affordable as to memory occupancy, communication overhead, and energy consumption. STaR features a generic architecture, and can be implemented for other hardware platforms and WSNs operating systems.

The rest of this paper is organized as follows. Section 2 presents the STaR architecture. Sections 3 describes STaR interfaces providing communication and configuration services. In Section 4 we discuss the performance of our implementation of STaR. In Section 5 we draw our conclusive remarks.

2 STaR architecture

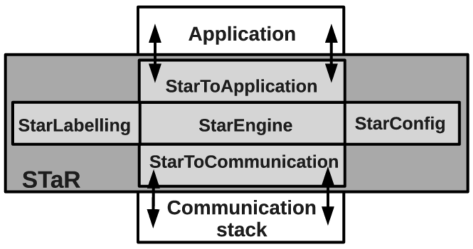

Figure 1: STaR component overview.

STaR allows for securing multiple *traffic flows* according to different *security policies*. Possible security policies include packet encryption, packet authentication, or both of them. The choice of appropriate security policies is related to the specific application and threat model, and is out of the scope of this paper.

As shown in Figure 1, the STaR component stays between the application and the 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, provides *transparency* of security with respect to the application, and exports the same interface as that of the underlying communication stack to the application.

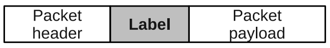

Figure 2: Example of packet processed by STaR.

The *StarToApplication* component provides the application with the same communication interface exported by the communication stack. The *StarToCommunication* component connects 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, providing reconfigurability at runtime. The *StarEngine* component actually processes packets, according to the security policy associated to the traffic flow they belong to. Distribution and revocation of necessary security material are left to other system components, which are not part of STaR and are out of the scope of this paper. Figure 2 shows a packet processed by STaR, with the *Label* field prepended to the packet payload.

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

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

3 STaR security services

As described in Section 2, STaR relies on packet labels 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 being transmitted, according to a specified security policy. 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 securing/unsecuring packets and mapping flow labels into security policies. These tasks are totally transparent to the application. 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. Thus, STaR manages 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. 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. *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 a set of configuration functions, which are described in Section 3.2.

The *PolicyDB* is formatted as follows. *PolicyID* values in the *PolicyDB* have to match *PolicyID* entries of the *SPT* 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).

3.1 STaR communication support

The following communication functions are provided.

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

Provide the packet *packet* of size *size* to STaR. Return TRUE in case of success, FALSE otherwise.

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

Provide the application with the packet *packet* of size *size* coming from STaR. Return TRUE in case of success, FALSE otherwise.

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

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

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

Return the security policy associated to *label*. Return an error code if 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.

Packet *P* is transmitted according to the following steps. 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. Then, 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, to deliver it to the recipient node(s). The label must never be encrypted, in order to assure that packet *P* is correctly unsecured at the recipient side. However, the label can be authenticated, in order to guarantee that it has been actually generated by the STaR component. Figure 3 shows an outgoing packet processed by STaR.

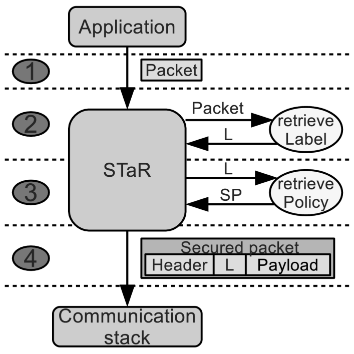

Figure 3: Outgoing packet processing.

Packet *P* is received according to the following steps. STaR receives the secured packet *P* from the communication stack, and retrieves the label *L* from the additional label, which can then be removed. Then, 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 4 shows an incoming packet processed by STaR.

3.2 STaR configuration services

STaR allows users to change security settings at runtime, and provides a specific *configuration interface* aimed at changing how security policies are used, and their association to traffic flows. In the following, we describe the configuration functions.

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

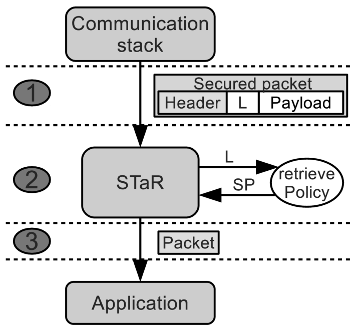

Figure 4: Incoming packet processing.

Set to TRUE the *Active* field of the *label SPT* entry.

`void disablePolicy(label);`

Set to FALSE the *Active* field of the *label SPT* entry.

`void changePolicy(label, newPolicy);`

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

4 STaR TinyOS implementation

We implemented the STaR component (Daidone, R., Dini, G., and Tiloca, M., 2012) for TinyOS 2.1.1, which is available at (TinyOS Working Group, 2012). We implemented the described security features with reference to the Tmote Sky motes (Moteiv Corporation, 2006) and the CC2420 chipset (Texas Instruments, 2012). We have implemented the *Skipjack* encryption module (U.S. National Security Agency (NSA), 1998) and the *SHA-1* module for integrity hashing (Eastlake, D., and Jones, P., 2001), and are working to provide more standard security protocols.

4.1 STaR memory footprint

ROM memory available on Tmote Sky motes is 48 kB, and may be 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 provided demo applications.

$\hat{C} = S + C$ 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 1: Detailed memory occupancy.

Table 1 provides information on memory occupancy, and shows the percentages of Tmote Sky ROM occupied by the original application and STaR (sub)modules. If we sum the contributions of the *StarConfig*, *Skipjack* and *SHA-1* modules, we observe 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 27.85% of the available memory, we have the 57.99% of 48 kB available for other uses. Thus, memory required by STaR is reasonable with respect to the available memory.

4.2 STaR performance evaluation

In our analysis, we assumed STaR operates on top of the 2.4 GHz physical layer, with a 250 Kb/s bit rate (Texas Instruments, 2012). We modeled the impact of security considering i) network performance degradation due to security processing and extra trasmissions overhead, and ii) the extra energy consumption, due to extra processing and transmissions. We evaluated the security processing overhead by means of experiments. The extra transmission overhead has been computed considering the bit rate and the packet size. Energy consumption has been evaluated analytically.

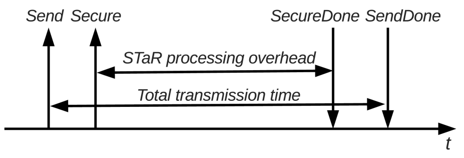

Figure 5: Send and Secure events nesting.

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

Policy	**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 2: 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. Results shown in Table 2 are averaged over all the different repetitions. We also report the standard deviation we derived from the independent replication method.

Considerable delays are due to the standard encryption and authentication algorithms, while the actual STaR contribution to the processing delay is just $142\,\mu s$. This is the additional delay required to add the label field, which is negligible if compared to the one introduced by standard cryptographic computations, such as those performed by *Skipjack* and *SHA-1*.

The transmission overhead has been evaluated analytically, considering a 250 Kb/s bit rate (Texas Instruments, 2012). Specifically, we have considered the time required to transmit the additional bytes added by STaR, according to the specific security policy. The original application packet size, including the header and the *Cyclic Redundancy Check* (*CRC*) is 21 bytes. The time d_{tx} required to transmit the original application packet is the ratio between the packet size in bits and the bit-rate: $d_{tx} = \frac{21 \cdot 8}{0.250} = 672\mu$s.

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

Table 3: STaR d_{tx} contributions overview.

Table 3 provides an overview of the transmission overhead, considering different security policies. 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 $32\,\mu$s, 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 of the original packet size.

We expressed energy consumption contributions as $\mathcal{E}_i = \mathcal{P}_i \times d_i$. Let d_i be the delay due to the considered operation *i*. $\mathcal{P}_i = V_i \times I_i$ is the single power contribution, i.e. the product between voltage and current of the MSP430 and CC2420 components.

Policy	**Processing** $\mathcal{P}_{proc} = 1.08$mW		**Transmission** $\mathcal{P}_{tx} = 31.32$mW	
	d_{proc} (μs)	$\mathcal{E}_{proc}$ (nJ)	d_{tx} (μs)	$\mathcal{E}_{tx}$ (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 4: STaR energy consumption contributions.

Table 4 provides an overview of such contributions. Considerable increases in per packet energy consumption 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: $\mathcal{E}_{proc} + \mathcal{E}_{tx} = 1155.6\,$nJ that is the energy consumed to add the label field to the original packet and transmit it. The extra energy consumption is affordable, if compared to contributions due to standard security mechanisms as *SHA-1*.

5 Conclusion

We have presented STaR, our security software component for WSNs. It protects 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. STaR allows users to change 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, add services such as key management, and compare STaR implementations on different architectures.

Acknowledgment

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

PLANET (Grant agreement no. FP7-257649). We also thank Davide Di Baccio for his precious help.

REFERENCES

Albano, M., Chessa, S., and Di Pietro, R. (2008). Information Assurance in Critical Infrastructures via Wireless Sensor Networks. In *Proceedings of the Fourth International Conference on Information Assurance and Security*, pages 305–310.

Buttyan, L., Gessner, D., Hessler, A., and Langendoerfer, P. (2010). Application of wireless sensor networks in critical infrastructure protection: challenges and design options. *IEEE Wireless Communications*, 17(5):44–49.

Cardenas, A.A., Roosta, T., and Sastry, S. (2009). 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.

CONET (2008). Cooperating Objects NETwork of excellence, European Commission, 7th Framework Programme, Grant Agreement n. 224053. http://www.cooperating-objects.eu/.

Crossbow Technology Inc. (2004). MPR/MIB User's Manual.

Daidone, R., Dini, G., and Tiloca, M. (2012). STaR implementation source code. http://www.iet.unipi.it/g.dini/download/code/star.zip.

Dini, G., and Savino, I.M. (2010). A Security Architecture for Reconfigurable Networked Embedded Systems. *International Journal of Wireless Information Networks*, 17:11–25.

Dini, G., and Tiloca, M. (2010). Considerations on Security in ZigBee Networks. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pages 58–65.

Eastlake, D., and Jones, P. (2001). RFC 3174. http://tools.ietf.org/html/rfc3174.

Gu, W., Dutta, N., Chellappan, S., and Xiaole, B. (2011). Providing End-to-End Secure Communications in Wireless Sensor Networks. *IEEE Transactions on Network and Service Management*, 8(3):205–218.

Hyun, S., Ning, P., Liu, A., and Du, W. (2008). Seluge: Secure and DoS-Resistant Code Dissemination in Wireless Sensor Networks. In *Proceedings of the 2008 International Conference on Information Processing in Sensor Networks*, pages 445–456.

IEEE (2006). *IEEE Std. 802.15.4-2006*. IEEE, Inc., New York.

Karlof, C., Sastry, N., and Wagner, D. (2004). TinySec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded Networked Sensor Systems*, pages 162–175.

Lanigan, P.E., and Gandhi, R., and Narasimhan, P. (2006). Sluice: Secure Dissemination of Code Updates in Sensor Networks. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, pages 53–62.

Maerien, J., Michiels, S., Huygens, C., and Joosen, W. (2010). 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*, pages 121–128.

Matthys, N., Huygens, C., Hughes, D., Michiels, S., and Joosen, W. (2012). 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*, pages 53–60.

Moteiv Corporation (2006). Tmote iv Low Power Wireless Sensor Module.

PLANET (2010). 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/.

Su, Z., Lin, C., Ren, F., Jiang, Y., and Chu, X. (2009). An Efficient Scheme for Secure Communication in Large-Scale Wireless Sensor Networks. In *Proceedings of the 2009 WRI International Conference on Communications and Mobile Computing*, volume 3, pages 333–337.

Texas Instruments (2012). CC2420 2.4 GHz IEEE 802.15.4 / ZigBee ready RF Transceiver. http://focus.ti.com/lit/ds/symlink/cc2420.pdf.

TinyOS Working Group (2012). TinyOS Home Page. http://www.tinyos.net/.

U.S. National Security Agency (NSA) (1998). SKIPJACK and KEA algorithm specifications.

Wong, C.K., Gouda, M., and Lam, S.S. (2000). Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30.

ZigBee Alliance (2008). *ZigBee Specification*.