SecLooCI: A comprehensive security middleware architecture for shared wireless sensor networks

Jef Maerien*, Sam Michiels, Danny Hughes, Christophe Huygens, Wouter Joosen

iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium

ARTICLE INFO

Article history:
Received 23 December 2013
Received in revised form 23 September 2014
Accepted 30 September 2014
Available online 14 October 2014

Keywords:
Security middleware
Wireless sensor networks
Security
Shared infrastructure

ABSTRACT

WSNs are trending towards dynamic environments that enable multiple parties to concurrently deploy and exploit multiple applications on shared nodes. The node owners want to share their nodes’ capabilities in order to increase return on investment, provide value added services, and easily share sensor node services. These concepts drive an evolving view on the software support that is required to service all stakeholders. For example, trust establishment, access control and security policy enforcement must be addressed. The node middleware must be extended to enable such shared usage of nodes while ensuring security. This paper presents the SecLooCI WSN middleware, which enables secure multi-party interactions on top of resource constrained sensor nodes. A prototype implementation for AVR Ravens running the Contiki OS shows the feasibility of the model for this low power micro-controller class of devices. This demonstrates that resource constrained sensor nodes are able to support secure node sharing.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Wireless Sensor Networks (WSNs) have many uses in a variety of fields such as industry [1], agriculture [2], logistics [3] and domotics [4]. These settings provide us with many and often dissimilar requirements. However, this paper identifies three similarities: WSNs are multi-party, multi-application and dynamic environments [5].

Firstly, WSN ecosystems are multi-party [6,7]. In all mentioned use cases, the ecosystem comprises multiple parties or administrative domains that at the very least want to share sensor data. Many WSNs provide data for interested third parties, be it for habitat monitoring [8], river monitoring [9], or supply chain monitoring [3]. In most current cases, users can only access a data repository made available by the WSN owner. In future use cases, users have direct access to the sensor data to ensure integrity and freshness, and deploy custom code to enable node local data processing and formatting. These users then reimburse the WSN owner, increasing the owner’s return on investment.

Secondly, WSNs are multi-application [10–12]. A real world sensor network is seldom single purpose. A sensor network consists of many nodes, some of which might have a single function. The entirety of the network however hosts multiple applications, sharing node and network resources. For example, a building sensor network executes a HVAC application, a person tracking application, and a fire detection application.

Lastly WSNs are dynamic in both physical and logical composition [13]. The physical composition of sensor networks changes due to node mobility and node failure: new nodes enter the network, and old nodes disappear because they stop functioning or move out of network range. The network’s logical composition, meaning the applications...
executing, is also dynamic: new applications are added, old applications updated, and redundant applications removed, as illustrated by multiple papers on application management in WSNs [14–16]. This node evolution is often divergent: nodes have different and evolving configurations and code installed to meet changing context and requirements [17].

Node sharing is a crucial enabler of such multi-party, multi-application, and dynamic use cases. As argued above, sharing sensor nodes allows increased node utility with relatively little additional cost or overhead. Some systems exist that allow insecure node and service sharing [18–20]. Once one party has physically installed a node, other parties can use the sensing and actuation services of this node. Platform Owners enable sharing because this allows them to offer value added (and paid) services. Node users on the other hand want to use shared nodes since this allows usage of the WSN’s capabilities without having to deploy nodes, significantly reducing cost and start-up delays.

Node sharing significantly affects the security requirements of sensor systems. To ensure node availability and integrity, node owners require that all data flows involving their nodes are secure. Since many of these data flows involve multiple parties, they must be updated to allow multiple parties to securely communicate with the sensor node with varying degrees of access. To maintain security in this multi-user environment, this paper identifies five data flows which require revision: (1) network initialisation, (2) application deployment, (3) application management, (4) application service usage, and (5) application communication.

While current WSN security middleware solutions, such as Zigbee [21], LEAP [22], and TinySec [23], provide secure application communication, the currently proposed solutions have significant shortcomings with regards to enabling shared usage of WSNs. These solutions are targeted at the classic single owner, single application, static network paradigm and often only look at specific features rather than a comprehensive solution. The multi-party paradigm requires that ad hoc networks can be formed consisting of nodes, owned by multiple different parties, and that multiple parties can securely deploy, manage and monitor sensor applications on these shared sensor nodes.

Some of our previous work proposes point solutions for certain sub-problems, for example secure network creation [24], secure application deployment [25], secure application management [26], and network monitoring [27]. This paper adds to this work by analysing the application lifecycle, identifying all security requirements with regards to sharing to create a comprehensive security middleware, mapping these security requirements to both existing work and novel work and integrating all systems into the comprehensive SecLooCI secure middleware architecture. A proof of concept shows the feasibility of this comprehensive architecture.

The contribution of this paper is a comprehensive sensor middleware solution that enables secure multi-party sharing of resource-constrained WSN devices: SecLooCI. An extensive problem motivation and requirement analysis shows the necessity of a security middleware for sensor nodes. The middleware architecture proposes an implementation independent solution for multi-party node usage. An implementation is created on AVR Ravens running the Contiki OS and LooCI component based middleware. The evaluation shows the feasibility of the architecture for even resource constrained wireless sensor nodes.

The remainder of this paper is structured as follows: Section 2 presents the context using a logistics use case and identifies the different roles in the system. Section 3 identifies the five security requirements, proposes the attacker model and lists additional requirements. Section 4 analyses currently available technology. Section 5 presents the SecLooCI middleware architecture. Section 6 describes the prototype, which is evaluated in Section 7. Finally, Section 8 formulates the main conclusions.

2. Context

This section presents the problem context. A logistics use case shows the necessity of multiple parties to share wireless sensor network infrastructure. The role model defines these different roles in the system and identifies their drivers and concerns. The lifecycle section identifies the necessary interactions between and requirements of the different roles to ensure a secure shared wireless sensor network.

2.1. Use case

Representative scenarios, for instance in logistics, show the importance of dynamic multi-party environments [6,3]. Multiple parties own part of the network and share their resources with each other and other third parties. These third parties reuse existing sensor deployments, significantly reducing the effort required by these end users. The Platform Owners recover some of the cost of installing a sensor network by charging these third-party users.

For example, logistics providers install a fairly heavy weight sensor in their containers with performance similar to a smart phone to perform supply chain monitoring, with smaller sensors and actuators across the container. These sensors allow visibility of container status for all parties in the supply chain, assuming the logistics providers shares the sensor node data.

Many parties want to interact with the sensors of the containers: (1) the owners of the goods want to know the containers location and to ensure the goods are transported in a correct manner (limited shocks, no extreme temperatures, etc.), (2) harbour owners and customs require node access to enable localisation, monitor container access and ensure correct handling of goods, and (3) governments require access, temperature and location data for security reasons: in order for easy customs processing, it is necessary to prove container integrity and ensure supply chain visibility, as for example required by the US C-TPAT treaty (Customs-Trade Partnership Against Terror) [28] or the European Authorised Economic Operator certificate [29]. All these parties prefer live data to ensure freshness, integrity and the ability to immediately
respond to potential issues. Retrieval of this data is assimilated in the sensor network to ensure the required freshness and integrity of data, requiring deployment of custom configurations and multi-party direct node access.

2.2. Role model

This logistics use case, and other similar use cases, show a drive toward network and node sharing. This section describes the different roles in a multi-party WSNs which result from separating ownership of platform, network, and application as shown in Fig. 1.

The Application Owner (AO) wants to use the WSNs to perform actuation or gather sensor data. He has certain functional requirements, such as receiving temperature data of nodes in a certain location, and non-functional requirements, such as the level of security and quality of service required of the communication channels. In the classical case, the AO would have to purchase sensor nodes, develop or buy an application, deploy a sensor network and provide networking capabilities in order to be able to deploy his application. This is a costly endeavour and the sensor data provided by those sensors might not be ideal due to the inability to install nodes on all locations and on new mobile entities.

By reusing existing sensor nodes, these parties can significantly reduce cost and effort of realising desired functionality. An AO can use such a shared sensor network by deploying a custom distributed application to enact the desired functionality. The applications require certain resources from the sensor node, such as sensing and storage capabilities. The AO can reimburse the PO for using those shared resources.

The Platform Owner (PO) is the owner of the sensor node platforms. In the classical case the PO would deploy his nodes, and provide network and applications himself. Many nodes often would not have access to networking provisions, especially in mobile scenario’s where ad hoc networks are necessary to send messages. Deploying such a system is often not sound from a cost-benefit perspective.

By sharing his nodes, he can get a faster Return on Investment since other parties will pay to use the node’s services. This reduces the PO’s investment risk of deploying a WSN. Additionally, by sharing node capabilities with other partners they can create more accurate or federated services, such as localisation, theft detection and environmental monitoring. Another aspect is that other parties may provide network connection in return for access to sensor node services. Clearly it is advantageous for the PO to share his nodes’ resources.

However, the PO’s primary concern is the continued and secure operation of his nodes. To ensure this continuity, he requires that other parties can only access limited functionality on his nodes, and that he can impose additional restrictions on the usage of these services. For example he wants that other parties can only configure their own data subscriptions, and edit their own preferences on the node.

The Network Owner (NO) manages the wireless network. He provides network and Internet capabilities to the local nodes. The NO sets up shared networks as a service to users of his physical location, similarly to many organisations that offer WiFi networking to visitors. For example the harbour owner offers Internet to customers using the harbour’s facility as a service. Additionally the NO offers a node repository service, allowing AOs to query which nodes are currently present in the network, and which services they offer.

Combining roles Each party can perform one or more of these roles depending on the situation. For example in the harbour context: the PO of the container nodes is likely also an AO since he will have an application monitoring the current state of the containers. The harbour authorities likely fulfil all three roles simultaneously: they provide networking to all containers currently in the WSN (NO), they have some static node infrastructure to allow for example localisation services (PO), and they have a monitoring application running on both their own nodes and foreign nodes to track all containers currently present in the harbour (AO).

2.3. Application lifecycle

This section provides an overview of a sensor network application lifecycle to enable shared wireless sensor networks, as shows in Fig. 2. This overview identifies the additional requirements by the different roles with regards to securely setting up and using a shared wireless sensor network. The application lifecycle lists the different stages an application and network go through from creation to removal of application, following a create, enact, run, remove lifecycle. The lifecycle starts with the declaration of policies and the creation of the sensor network application. Next, the application is enacted and executed. Finally, the lifecycle ends with the removal of the application. Section 5.2 illustrates the lifecycle on the logistics use case proposed in Section 2.

The first phase, the create phase, is divided in smaller sub-phases based on role interactions and time of occurrence: (1) creation of security policies: (all roles individually); (2) creation of the application: (AO only); (3) setup
of the network: (AO and PO); and (4) creation of application instantiation: (AO only). Sub-phase 1 and 2 are independent of each other and can happen in any order, 3 must happen after 1, and 4 must happen after all others, as shown in Fig. 2.

The first step of the multi-party lifecycle is the **policy declaration**, which is done by all roles: PO, NO, and AO. The POs declare which parties are trusted to use which resources: POs need to be able to specify who can deploy applications, use network and other services, to which extend and at which cost. The POs also specifies the minimal security level for data produced by their sensor nodes. NOs need to specify which parties they allow use of their network infrastructure. AOs need to specify which parties or nodes they trust for application deployments.

Next comes the **application creation** phase. To enable easy application reuse, the AO requires that an application is specified separately from the actual deployment, and offers some point of configuration to tune the application to the exact deployment conditions. This paper assumes that applications are declared using the Service Component Architecture [30] standard. This standard states that an application is a collection of parameterizable components. Components offer services to other components, and can use services provided by other components, the operating system and the execution environment on which they run. Current research has shown that sensor networks can be efficiently programmed using components [18,19,14]. The life cycle however is independent of the exact application composition model used: while the implementation might differ, the life cycle will remain the same.

Next is the **network setup**. POs require NOs to create a network, providing the necessary gateway nodes connecting the WSN with the Internet. Once the initial network is established, new nodes from trusted POs can join the network and use the networking capabilities to communicate with server infrastructure. The PO and NO might require to negotiate cost and quality of service. At the end of the phase, the node can communicate securely with other nodes in the network, as well as users on the Internet. This stage assumes that nodes of different POs can communicate with each other and with the infrastructure of the NO. While currently there is still significant research in the MAC, routing and networking protocols to be used in WSNs, it is necessary for standards to emerge to ensure interoperability.

Once deployed, the **application runtime** phase starts. During this phase, applications execute and communicate using messages, and use system services. The AO may require his application traffic to be secured if so specified

![Fig. 2. Overview of the lifecycle of a distributed multi-party sensor network application. At each phase, the role(s) relevant to that phase are mentioned. Interaction is required in multiple phases to ensure safe cooperation.](image-url)
during the application instantiation phase. The PO requires that the application components only use the number of specified services, and only for a set amount to prevent overuse. Additionally, the AO may need to change the application during this phase. For many use cases, the AO only learns the exact environment by sensing it, and subsequently wants to adapt his application based on such sensing, or as the environment changes, for example due to nodes leaving and new trusted nodes arriving. Naturally the AO and POs require that this adaptation happens securely. During the runtime phase, the PO needs to monitor the node system to ensure that all applications and services operate as expected. This requires a flexible infrastructure monitoring problems, such as network attacks, platform error or application service overuse.

Once the application has performed the necessary sensing and/or computing tasks, the application removal is done by AO. To remove applications, an AO must be able securely manage the application. The PO should also be able to remove application components and configurations, for example when the PO detects that a application component uses too many system resources.

The final phase is application chargeback. Using the data gathered from the application monitoring and other runtime data, the PO can calculate a cost of running the applications, and charge the AO for his applications on the shared nodes.

3. Requirements

The context has shown a clear need for sharing of embedded sensor nodes. To enable this scenario, the security provisions of wireless sensor nodes and networks need to be updated. This section identifies the data flows impacted by sharing, presents the attacker model and lists the non-functional requirements of the security middleware.

3.1. Feature requirements

This paper identifies 5 data flows which involve significant interaction between the different roles and the node during the lifecycle as presented in Section 2.3. The data flow approach is recommended by a.o. the SANS institute [31]. This paper focuses only on the five data flows which are significantly affected by node sharing.

3.1.1. Network initialisation

The first data flow is network initialisation. During the network initialisation, the node has to register with the Network Owner (NO). During the initialisation the PO of the node joining the network requires that the NO is known to be trustworthy. The NO on the other hand also requires that the PO and the node are trustworthy before allowing the node to join the network. This trust must somehow be established, and deployed onto the node. This trust exchange is crucial to establish a shared network.

Once trust is established, and key material is exchanged, the new node potentially needs to be configured with additional network configuration and enable routing. However, we consider routing and network configuration out of scope of this paper, since this is not significantly impacted by sharing once key material and trust has been set up. There is a large body of work that attempts to solve these issues, which can be transferred to the current system.

3.1.2. Application deployment

Once the node has joined the network, and network configuration has been established, the node needs to be updated with custom code during the application enactment phase. The PO requires that the new code is deployed by a trusted party, and that the code is approved. A PO likely wants to first review all code, before it is deployed onto the node, to ensure the code does nothing malicious. The PO must then be ensured that this code is deployed unaltered. Additionally the PO can have additional security policies associated with the code that also need to be deployed. The AO also requires that his code is deployed without alterations, so that no other parties can abuse his code. Potentially the AO requires that his code is deployed confidentially.

3.1.3. Application management

After code has been deployed on the node, the application runtime starts, and the AO must manage the application. To manage his applications, the AO sends management commands to the node to either reconfigure or introspect. The AO requires that only users that he trusts can manage his applications. The PO requires that AO’s can only manage their own applications and configuration, while not threatening the integrity of the platform, nor the configurations of other AOs.

3.1.4. Application service usage

During the runtime phase, the application uses services provided by the node, such as communication service, and sensing services to gather information, enact change or communicate. The PO requires that this service usage is monitored and restricted. This ensures that an application component can only use a limited amount of node services. For example an AO should only be able to send a limited amount of messages, as to reduce the communication overhead of the component.

3.1.5. Application communication

Once the application has gathered data, it needs to communicate this data remotely. Both the PO and AO require that they can express security policies which regulate which information must be encrypted or authenticated. These policies must then be negotiated and deployed on the nodes, to ensure the most strict security policy is enforced. For example an AO wants to deploy a localisation application onto a node, and requires that the localisation data is transmitted with message authentication. The PO however requires that all localisation data is transmitted confidentially. These policies must be resolved, the most restrictive one selected (confidentiality), and then deployed on the node and enforced on all future communication.
3.2. Attacker model

The middleware must be able to handle attacks from multiple different angles. This paper differentiates three types of attackers: network attackers, physical attackers and insider attackers. This section lists the resources available to these attackers, the goals they try to achieve, and what the middleware must be able to prevent them from doing.

**Network attackers** only have access to the messages exchanged over the network, in accordance with the Dolev–Yao [32] model. A network attacker can intercept, manipulate, duplicate or create messages, but he cannot break the underlying cryptographic primitives. He aims to gain information on, or some kind of control over the network. This includes: gaining the key material of one or more nodes, gaining information on the applications that are currently installed, or the data contained and produced by the sensor nodes. Alternatively he may attempt to corrupt the sensor nodes and applications by performing network level attacks such as denial of service attacks. The node security middleware must prevent network attackers from being able to manipulate the sensor nodes in any way possible and to gain any information that is considered confidential. While a sensor node cannot prevent a network attacker to deny correct reception, he should not be able to do anything else.

**Physical attacker** are attackers who gain access to all information contained in one or more nodes through physical means. Standard sensor nodes are currently considered incapable of resisting an attacker who can physically probe the memory, unless the sensor node is physically secured, by either being encapsulated in tamper proof hardware, or in a physically secure environment. He aims to gain control over or disable the network using the credentials gained from one or more hacked sensor nodes. While it is impossible for software to prevent hardware tampering, capturing one node should not allow the physical attacker control over other nodes. The node middleware must minimise the effects of node capture. Networks must be able to recover from the capture of any node once detected. The middleware must also prevent a physical attacker from being able to manipulate nodes other than the one he has probed.

**Insider attackers** are attackers that have credentials to access some limited functionality of the system. They use these credentials to attempt to gain more privileges. These attacker have the same limitations as networked attackers, but additionally possess valid credentials to access partial functionality of the system. Their goal is to gain additional information contained in the system or additional control over the system. The node middleware must ensure that the system and information remains confidential and integer, and must ensure that potential breaches are registered.

3.3. Non functional requirements

As discussed in the previous subsections, the secure WSN middleware needs to be able to handle a variety of different usage patterns and be able to adapt to many potential situation. This section gives a general overview of the non-functional requirements of the security middleware:

3.3.1. Evolvability

The middleware must be able to support sensor network evolution, be it evolving node compositions, application compositions, application requirements or trust relationships.

3.3.2. Performance

The middleware targets low resource environments and thus requires resource consumption to be as low as possible. This includes communication overhead, memory overhead (ROM and RAM), and processing overhead.

3.3.3. Transparency

Applications and application developers should be oblivious to the fact that security is present on the platforms. The security middleware must transparently provide the necessary security features to allow all applications to be secured. Such model is dictated by clean software development to ensure separation of concerns: application developers are seldom security experts, and thus are often incapable of creating and maintaining bug-free secure communication protocols [33]. The middleware must allow AOs to specify security concerns in a policy driven fashion to allow easy declarative description of required security provisions, for example in a policy language such as the XML based XACML [34].

3.3.4. Compatibility

WSNs have multiple possible usage patterns. For example some WSNs have a continuous connection to the Internet, while others have a sporadic or very delayed connection to the Internet. Furthermore WSNs use many different communication patterns such as one-to-one, many-to-one, or opportunistic communication [35]. The security middleware must be compatible with these usage patterns.

3.3.5. Flexibility

Due to the heterogeneous and resource limited nature of WSNs and multiple different usage patterns, security should be flexible. The security middleware must be able to support multiple levels of security and different cryptographic algorithms.

3.3.6. Shareability

The middleware must support sensor network usage by multiple concurrent users with divergent access rights.

4. Related work

The state-of-the-art has proposed many interesting solutions which potentially address the node security requirements that were presented in Section 3.1. This section presents an overview of the most relevant related work, and identifies that most related work focuses on only one of the required features for a secure node.
environment. Also, most related work offers little to no support for node sharing.

4.1. Network initialisation

Much research has been performed regarding network key distribution and encryption protocols in WSNs [36,37] to secure network communication. The existing research can generally be subdivided in either pure symmetric key approaches, or asymmetric key integration. Some examples of pure symmetric key approaches are: (1) key distribution using group and pairwise keys [38], (2) key pre-distribution schemes [39], (3) time-based key distribution [40], and (4) time based key disclosure [41]. The application of asymmetric key encryption usually involves optimising RSA for usage in WSN either by optimising hardware [42] or software [43], or looking at novel and efficient algorithms such as Elliptic Curve Cryptography [44].

Key WSN examples are Zigbee, LEAP, and TinySec. Zigbee [21] offers a distinction between master, network and link keys yet often fail to offer multi-user interaction or easy network evolution. At network setup, the node is authorised by the Trust Center (TC) by using the master key. Once authorised, the TC can securely deploy network and link keys. Similarly LEAP [22] distinguishes five different types of keys: (1) basestation keys, (2) pairwise node keys, (3) cluster keys, (4) group keys, and (5) master keys. At deployment each node is equipped with the network master key. This key is then used to set up the other keys. Once the setup time is passed, all nodes delete the master key, only leaving the other keys. TinySec [23] offers a flexible link layer security that offers two modes of security: authenticated encryption and authentication only. It can be used with any key management protocol, and is independent of the actual protocol used. Also it is not possible to set up multiple communication channels with different configurations in TinySec, creating a protocol that is either always on or always off. While offering secure communication, none of these protocols offer multi-party integration, easy network evolution, or manageable network security.

4.2. Application deployment

With regards to secure deployment, many protocols have been proposed in the related work, yet most assume single owners nor include deployment of policy metadata. Most algorithms attempt to secure Deluge [16], the TinyOS [45] code dissemination protocol and propose a series of hash chains or hash trees where the start of the chain/tree is signed by the base-stations private key to provide integrity and authenticity, for example Sluice [46] and Seluge [47]. This can either be done using symmetric key encryption or public key encryption. Some propose to encrypt the binary to provide confidentiality [48]. While these approaches offer secure and efficient deployment of code, they neglect the multi-user aspect of code distribution and often have strong assumptions of single user sensor nodes. Neither do they consider the need to deploy resource usage limits for the deployed components.

4.3. Application management

Many protocols offering a secure network layer with authenticated message transmission on a node to node basis, and can thus be considered to also tackle authenticated management communication. Some examples are LEAP [22], which has pairwise shared keys between nodes, allowing unique identification, and TinySec [23], allowing message authentication by using MACs. There are however some additional protocols aimed specifically at authenticating users at certain sensor nodes.

Similarly to secure network communication, there are symmetric key and asymmetric key approaches. An example of a symmetric key approach is saAQF [49]. saAQF assumes a single network user which has the gateway key chain. To authenticate a message, the user calculates a 1 bit MAC with each key, and concatenates all these MACs as message MAC. Each node has a subset of the user's keys, and can verify the bits for which he has the key. This prevents a single node capture to allow physical attackers to send authenticated messages to other nodes. The second category are asymmetric key approaches, which authenticate management messages using certificates. Several protocols have been proposed, that optimise this approach for WSN usage, such as for example using bloom filters for verification [50], single network CA's [51], or ring signature authentication [52]. While the asymmetric key schemes offer multi-user authentication, they are often too heavy-weight for efficient use in resource restrained WSNs. Most schemes also neglect to offer a clear management infrastructure allowing evolution of access rights, or neglect authorisation altogether.

4.4. Application service usage

Related work in application service usage monitoring and enforcement can be divided in two approaches, active node monitoring, and passive node monitoring. Current proposed solution however have significant drawbacks in dynamic multi-user environments and seldom offer node side policy enforcement.

An active node monitoring solution is the Simple Network Management Protocol [53], allowing a host to request information from networked devices. This approach allows users to request monitoring data from nodes, but does so at a very high cost. An active monitoring approach from the WSN field specifically is Sympathy [54]. This debugging tool allows a fine grained metric collection of sensor nodes such as next hops, neighbours and uptime. While being very powerful for debugging network problems during testing, it does not offer capabilities for more fine-grained and evolvable decisions on what information to monitor, and does not allow application components to be limited with regards to node usage.

Passive monitoring approaches typically monitor WSN traffic, interpret messages, and relays them to a server infrastructure using a secondary network, such as Ethernet [55] or Bluetooth [56]. While these approaches can monitor the WSN without any node overhead, they do require the deployment of a costly and redundant secondary network, cannot interpret secure messages, detect node
transmission errors, have problems with node mobility and cannot limit component’s node usage.

4.5. Application communication

Most middleware and protocol approaches do not distinguish between management and application communication, disregarding multi-level content, origin and target driven security. Often you can only have one general security setting, which is assumed to be the only setting in the network [22,23], applied to management communication and all kinds of application communication. There is a however clear distinction between different types application and management communication, and from who this traffic originates. Current related work does not offer such content dependent security policies, where certain application data can be sent in the clear, while other application data is required be encrypted and authenticated. While some related work offers some level of fine-grained security, it does so on a network or link level, rather than based on data type.

4.6. Related work summary

Related work has proposed many partial solutions for a secure node middleware. Yet no comprehensive node security middleware has been proposed offering secure management, network communication, application communication, application deployment and application monitoring. Neither has it been shown that such a secure middleware offering extensive multi-user operations can operate on resource constrained sensor platforms.

5. Architecture

This section proposes the SecLooCl middleware architecture for securing wireless sensor nodes. The architecture extends a generic pre-existing non-secure message and component-based middleware with security extensions. This section first describes a generic high level node architecture for non-secured systems. Next this section introduces the middleware modules required to ensure data flow security as stated in the previous section. An overview of the architecture can be found on Fig. 3. A short list of provided services is shown for each of the components, which authorised users can use to manage their assets. The list of services only shows the add and set function for brevity. Retrieval and removal of configurations is also present.

A typical reconfigurable node system is composed of the following abstract components: the OS networking layer, a deployment component, a message distribution component, an application layer containing application components providing application and serialisation logic, and a service layer offering middleware and OS services. Application components are responsible for interpreting messages they receive and then calling the necessary services to enact the requested application functionality. The middleware services offer management of the message distribution system and application components. The OS services offer sensing, actuation, timing and other services to the components. A management component receives management messages and calls the management services to enact the requested management request.

The middleware must offer the following services to users:

- deploy component: deploys a new component, and sends back a node local component identifier.
- send management message: sends a management request to the node, which is delivered to the correct manager component depending on type of management message.
- add application subscription: example of management request, adding a subscription to some information in the message distribution component. Source and destination can either be local or remote. Such wiring policies allow the message distribution component to correctly deliver messages.
- set parameter: sets a parameter of a certain component to the required value.

Take for example an upgradeable CoAP or HTTP server which allows GET, PUT, POST and DELETE commands to allow node reconfiguration, such as for example the Californium CoAP Framework [57] or an Apache Tomcat server. It is composed of the networking component, providing an IPv6 layer. The message distribution component looks at the PATH of the CoAP request to delegate the request to the correct application component. This component is called the local endpoint in Californium. The PATH in combination with the command could be regarded as message type. The application component would look at the parameters of the request and the HTTP command, interpret them and perform the necessary actions either by retrieving data from other node services, or setting parameters of certain configurations. Californium calls such application components local resources. The server allows new components to be installed. Once installed, the new component must register itself, indicating that it can receive messages sent to a certain PATH and with a certain command, basically adding a subscription to itself for the message type (PATH + COMMAND), from all sources. Potentially the component could also send out certain requests to services, indicated by a PATH name and command, which the message distribution component can then delegate locally or remotely depending on installed subscriptions, allowing loose coupling of sensor network components.

5.1. Security architecture

This section identifies seven necessary additions to the non-secure system to secure the five data flows significantly impacted by sharing:

Secure network component: Secures the network initialisation data flow, enables secure joining, using and managing of wireless network.
Communication interception component: Authenticates and encrypts the application and management communication data flows based on installed policies.

Service interception layer: Provides authorisation for the management communication data flow and applications service usage data flow.

Security data store: Stores credentials, access policies and usage policies for the different users, applications and services.

Secure deployment component: Secures the application deployment data flow to allow multiple different parties to deploy application components.

Monitoring and enforcement component: Registers the usage of the different application components and node users, and potentially limits them.

Security management component: Allows users to add and remove users, parties, roles, and security policies.

The following sections describe the different components in more detail.

5.1.1. Secure network component

The secure network component secures the network initialisation data flow. The secure network component secures network level messages and ensures that no outsiders can read the messages that travel in the network. This paper assumes that nodes can communicate with each other using a standardised MAC and IP layer. This paper does not look at secure routing, since this topic is adequately handled by related work, and is not significantly changed by multi-party environments.

This component detects when the node enters a new network. When this happens, the Network Owner (NO) receives the node’s and Platform Owner’s (PO’s) identity, and some proof that the node is in fact in its network by retrieving a node token. The NO then contacts the PO securely using existing trust and certificate relationships. The NO sends the proof that the node is in his network. Once the PO verified this proof, the NO and PO negotiate a gateway key, allowing the gateway to manage the node. The PO encrypts this key into a token, together with some security metadata, and hands it to the NO. The NO delivers this token to the node using his network. The node validates this token, allowing the NO access to network management function. This component must provide the following services:

- get platform owner ip: retrieve the IP address of the PO.
- get node identity token: get a node token that proves the node is present in the network of the NO.
- deploy network token: deploy a network token, allowing the NO access to network management services.
- deploy network key: set the network key, allows the NO to manage the nodes network security information.

Once the NO has deployed the token, he can add the node to his Network Information Service. This service lists all nodes in the network, together with their available services. This allows the NO to keep an overview of the nodes that are currently in the network. Nodes leaving the network can either be detected by a heartbeat signal sent by the node, or a relocation message which is sent to the NO where the node was previously registered.
5.1.2. Communication interception component

The communication interceptor enables secure end-to-end communication. It comprises of two sub-components: one for management messages, and one for application messages. The difference between the two is that management messages can be attributed to users performing reconfiguration or introspection of the node's functional composition. Application messages are not directly attributable to users, and do not perform reconfiguration but rather only send functional information, such as information on sensor data.

To provide secure communication, the communication interceptor intercepts messages coming from or going to the secure network component. Depending on the type of message, message integrity, authenticity or confidentiality must be ensured. The communication interceptor provides this in an end to end fashion.

Management messages must be secured, i.e. authenticated and encrypted, at all times to ensure the security of the platform. They can be attributed to a user, either directly or through a system that acts on behalf of a user. The level of security for this kind of message is not evolvable, all messages must adhere to the same level of security in order to access management services. The users allowed to access services however is evolvable. The user IDs and key material required for authentication is stored in the user data store.

Application messages on the other hand may need no security, or high security depending on the content of the application. Application security policies specify the required level of security and key material for different message types. These policies are deployed on the sensor nodes whenever a new application is deployed. Each time the communication interceptor intercepts an application message, these policies are checked and the necessary operations are performed.

This paper proposes to separate encryption context from encryption policy. The encryption context specifies the security operations and consists of: (1) primitive: hash/MAC/encrypted, (2) key material, and (3) algorithm. The encryption policy declares which types of data are sent using which security context. This allows multiple different outgoing policies to use the same encryption context and a single incoming policy to declare the required level of security for a certain message type covering multiple channels. This reduces overhead caused by encryption policies and key material, and it allows for easier evolution of both security context (re-keying) and policy.

Incoming encryption policies state the needed security primitive for the incoming messages depending on message type, and message sources. This allows incoming message policies to express security policies more compact. By expressing the incoming policies potentially independent of security context, it is more flexible and allows such policies to be expressed in a single short expression.

This component must offer the following services:

- add channel: adds a channel with a given algorithm and security context. The security context contains key length, and key bytes.
- add outgoing policy: adds a policy which specifies which events need to be send over a given channel.
- add incoming policy: adds a policy which specifies the required security level for given incoming data.

5.1.3. Service interception layer

The service interception layer checks whether the entity accessing a service is authorised to access the service. A service interception component is inserted between application components and access restricted services. It intercepts service request and decides whether or not the request is allowed based on request meta-data and data stored in the security data store component. Then it contacts the Monitoring and Policy enforcement component to update usage information.

5.1.4. Security data store

The security data store contains information of the users currently installed on the system. This includes the identification of the users currently installed, the key material which is needed to authenticate them, and the roles and permissions these users have to access the node system. The security data store also contains the current usage statistics by the different users, to be retrieved by both the Application Owners and the Platform Owner.

The prototype assigns each service to a party. This party owns the service and controls access to that service. A party is a user or a group of users which provides the given service. For access control, this paper proposes a role based access control scheme, where users have roles at different parties. This allows compact encoding of access rights. A user can have a role with multiple parties at the same time, one for each party. Each service is owned by only one party. Each party dictates for each interface to the service which role the accessing user is minimally required to have. A user thus has access to a service if he has a role which is equal or higher to the required role. This paper initially proposes a simple hierarchical structure of roles: viewer, user, manager and administrator. These simple roles allow most requirements to be expressed. The architecture however is independent of the exact role structure, and can be adapted to use other role models, or other access control models.

This component must offer the following services:

- add token: add a pre-created token which adds a user with pre-established security credentials.
- add user: add a user. Can only be called if current user is authorised to add users.
- add party permission: give a user permission to resources of certain party. Can only be called if current user is authorised to add users to the given party.

5.1.5. Secure deployment component

To ensure only authorised user can deploy valid component, the middleware requires that a token is sent to the node before the actual component. The node validates the token, and ensures that the component that is deployed, has been approved by the PO. The AO receives the tokens
when the PO approves the components which the AO wants to deploy. This PO declares which applications are permitted to be deployed together with the allowed range of permissions and resource restrictions. The token also contains these resource restriction to ensure that the application of the AO only uses the agreed amount of resources.

If the token is valid, and the component code matches the token, the node installs the code and registers the resource limitations polities with the monitoring component. Once the application component is installed, the node returns some component identifiers which allows the AO to manage the component in future. This can be a numeric or literal identifier. In case the token or component validation fails, the AO returns a failure message.

This component must offer the following service:

- deploy token: deploy a security token. Token is cryptographically signed, and dated. If successfully decrypted and verified, the deploy component service will become available.
- deploy component: deploy a component. Component must match component hash contained in token, and will be started with limitations as described in token.

5.1.6. Monitoring and enforcement component

The monitoring and enforcement component monitors the behaviour of users and applications on the sensor node. It ensures that applications do not use excessive amounts of data, nor users perform excessive amounts of introspection or reconfiguration requests. This component is dependant on the Service Interceptor to monitor requests performed by users and application components.

The monitoring and enforcement framework performs two functions: data acquisition and policy limitation. Data acquisition is done by filtering raw monitoring data with monitoring profiles: a profile translates intercepted function calls to interception types and interception count, basically indicating which counter (identified by the interception type) has to be incremented by which value. This allows for aggregation and policy reasoning. Example interception profiles could be (1) all memory used per user, (2) memory used per component, (3) total processing power used per party, aggregated over all components, or (4) processing power used per component. Policy limitations are done by resource restriction policies: these policies dictate how much of the certain resource can be used. The resource is identified by the interception type generated by the monitoring profile. It is thus clear that policies can only be deployed if the profile generates the correct interception type.

The framework requires components to only use the instrumented interfaces. Since current embedded sensor nodes do not offer strong memory protection, this can cause issues. However since the PO has control over which components are deployed, he should verify that code does not perform any illegal actions before allowing deployment.

This component must offer the following services:

- deploy profile: add a monitoring profile, specifying which resources need to be monitored.
- add resource restriction policy: add a policy regarding a certain component, specifying the amount of resources that component can use.

5.2. Typical use case

This section presents a simple yet typical WSN use case as described in Section 2.1, showing how the different components are used across the lifecycle. The harbour customs wants to monitor the location of all containers in the harbour network, which is provided by the harbour owner. To do this, the harbour customs deploys a component that listens to authenticated beacons and calculates location based on these beacons. The component then broadcasts this location, encrypted and authenticated towards the harbour customs server.

Each party has a dedicated role in this example. The harbour customs wants to deploy a component, making him the Application Owner (AO). The harbour owner provides the network which allows all nodes from all parties to communicate with each other and with server systems. This makes him the Network Owner (NO). The logistics provider provides the actual node platform used for deployment. This makes him the Platform Owner (PO). Likely the logistics provider also has applications of his own, making him also an AO. However, this scenario documents the interactions when all three roles are enacted by different parties. The remainder of this section will go through all the steps in the lifecycle as shown on Fig. 2 and specify what all parties must do during each phase.

5.2.1. Policy declaration

All parties have to declare whom they trust. The logistics provider (PO) and harbour owner (NO) must trust each other so the logistics provider can use the network provided by the harbour. The harbour owner must trust the harbour customs, since the harbour owner must send a notification each time a container enters. The harbour customs must trust the harbour customs so the harbour customs can deploy an application on the node of the logistics provider. This is the policy declaration phase of the lifecycle.

5.2.2. Application creation

The harbour customs (AO) must create or purchase the localisation application so it can deploy a component on each node which enters the network of the harbour owner. Likely a general localisation application will be created for multiple harbours, and instantiated for each specific harbour with the specific beacon locations as parameter. A simplified application exists out of three different components, instantiated across the network: (1) the beacon component broadcasts beacon location type message, contain beacon-IId, timestamp, and other necessary metadata. (2) The node component listens to beacon location type messages, and sends out node location type messages. (3) The data gathering component runs at the harbour customs, which listens to all node location type messages and gathers the location of all nodes.
5.2.3. Network setup

When the container arrives at the network of the harbour, the node detects that it has entered a new network, and the node registration starts as shown on Fig. 4. The secure network component contacts the harbour owner (NO). The harbour owner retrieves the logistics provider’s (PO) identity and the token proving the nodes entry in the network. He then contacts the logistics provider and negotiates cost, network key and parameters. The logistics provider creates the node token and transmits it back. The harbour owner deploys this token, sets the network key and other potential network policies. Afterwards, he notifies the harbour customs that a node has entered the network. During this exchange the harbour owner can also inform the logistics provider what the new (IPv6) address of the node is, allowing the logistics provider to communicate with his nodes.

5.2.4. Application instantiation

The harbour customs is notified that a new node has entered, and instantiates his localisation application with the new node. He contacts the logistics provider to request to which services are available for the localisation application. The logistics provider and harbour customs mutually authenticate each other and negotiate policy and security requirements. For example the logistics provider requires that all location data of his nodes must be encrypted, while customs would satisfy for authentication only. This negotiation ends when the harbour customs and logistics provider agree on the component deployment and security context (location is encrypted, beacons authenticated). Once agreement is made, the harbour customs receives a token from the logistics provider which he can use to deploy the localisation component on the sensor node.

5.2.5. Application enactment

The harbour customs contacts the secure deployment module of the node, sends the security token, and deploys his code component as shown in Fig. 5. The node verifies that the deployment request is authorised by checking the deployment token. After the node received the component, the node verifies that the component is authentic and authorised. Once the component is verified, the node install the component in its runtime environment. Next the customs officer is installed as a users of the platform, so he can manage his application component. The harbour customs then sets up the necessary configurations, such as the current beacons, the subscription for the node location messages and security channel information: key material to authenticate beacon location messages and encrypt the node location messages as shown in Fig. 6. Once this is done, the harbour customs starts the application. To enact all these management operations, the harbour customs contacts the node using his credentials and manages the node using reconfiguration requests. All these requests are encrypted, authenticated and authorised by the communication interceptor on the node.

5.2.6. Application runtime

The final phase is the application runtime phase. During this phase messages must flow securely between components, and users must be monitored.

5.2.6.1. Application communication. Once the component is running, it will receive messages from beacons, and send messages to the harbour customs. Each beacon location message is authenticated by the application communication interceptor when coming from the secure network layer as shown in Fig. 7. The interceptor drops all non valid location messages. After interception the beacon message is dispatched to the application component. This component then calculates its location, and sends out the node’s location using a node location message. This message is then sent to the harbour customs. The application communication interceptor again intercepts this message and encrypts it, following the outgoing application security policy.

5.2.6.2. Application monitoring. The monitoring frameworks monitors the memory and processor usage of the component, based on deployed resource limit policies, as agreed upon in the contract. The monitor component tracks node service usage by using the service interception proxies as shown on Fig. 8 and ensures that the allowed limits are not surpassed.

5.2.7. Application removal

Once the container leaves the harbour, the logistics provider or customs removes the component and all associated policies. This frees up the used resources for other future applications. All removal and clean-up commands are again authenticated and authorised by the node.

Once all components have been cleaned up, and final node usage data has been acquired, the logistics provider can potentially charge the harbour customs for the node usage. However, since the harbour customs can offer the localisation service to the logistics provider, the logistics provider likely will not require nor receive any payment.
6. Prototype

This section presents the SecLooCI middleware that implements the components discussed in the previous section. This section goes over each of the five data flows, identifies which security components from the architecture are necessary to secure the data flow and details how they interact. For each data flow the overhead in terms of memory, processing and communication is listed. All components are implemented and integrated in a single prototype running the Contiki operating system [58] and the LooCl component middleware [18]. The implementation has been tested on AVR Raven sensor nodes [59], which have a 20 MHz MCU, 128 kB of ROM and 16 kB of RAM. The prototype shows that this class of devices is capable of supporting secure node sharing and implies that all more capable classes of devices are also able to support the necessary security middleware.

This section evaluates the framework protecting each data flow in terms of communication, memory and processing overhead. The test setup are single nodes in a one-hop network directly communicating with the

---

**Fig. 6.** Service management data flow.

**Fig. 7.** Application communication data flow.

**Fig. 8.** Service usage data flow.
gateway. All overhead is measured on a prototype implementation that is deployed onto AVR Raven sensor nodes. We chose this setup because: (1) many current industrial deployments only show single hop networks with very little mesh networking. (2) This evaluation setup allows for a clear, straight forward and easily reproducible test setup. (3) The security middleware enables sharing on a per node basis, with all messages managing a single node. There is currently no network-wide reconfiguration or management operatives. The overhead of the middleware thus needs to be evaluated on a per node basis instead of a network wide basis. (4) There are currently no standard mesh network test setup. Hence any networked results would only have very limited to no comparison value to other experiments. Additionally most networked system results are simulations, while we chose for an actually deployed prototype approach. (5) Recent work [60,61] has shown that single hop networking is under certain conditions more energy efficient compared to multi-hop routing, and is easier to set up and maintain.

6.1. Assumptions

The creation of the middleware is based on a few assumptions. Firstly the middleware assumes that nodes always start in the care of the PO. The PO must deploy the security and application middleware onto the node, together with initial key material that will be used in the different protocols. It is crucial that this deployment happens securely, since the following protocols depend on the security and secrecy of this initial key material [62]. The initial key material are one or more long term symmetric keys. The middleware uses a pure symmetric key approach since this significantly reduces the amount of memory and processing needed for protocols and key material. More capable nodes such as embedded PC’s or smart phones can support other cryptographic primitives, such as asymmetric cryptography. However, the goal of this prototype is to explore the minimal system requirements to support shared node operation.

The system uses TCP and UDP on top of an IPv6 6LoWPAN based wireless sensor network with 802.15.4 MAC layer. This allows server infrastructure and clients to easily and transparently communicate with sensor nodes across the globe, without having any notion of the exact WSN settings. We assume that nodes receive global IPv6 addresses from the Network Owner using IPv6 Stateless Address Autoconfiguration [63]. This then enables any party anywhere in the world to communicate with the nodes. While IPv6 and TCP are in theory not necessary for the system to operate, in practice it greatly simplifies node mobility and reliable end-to-end connectivity due to the standardisation and universal integration of the protocols. Naturally other protocols can be used, yet they would require adaptation layers to be able to integrate with the Internet.

Lastly the prototypes aims to provide secure node operation and management, however current sensor nodes offer little memory protection. Hence when code is deployed onto a sensor node, it has access to potentially all sensor node memory. This paper assumes that certain countermeasures are in place that prevent code components from performing such malicious actions either in hardware or software. The hardware countermeasures prevent over the air components from accessing prohibited memory (i.e. the rest of the middleware) [64], while software countermeasures ensure that only verified code can be deployed [65]. This code verification can be done during the planning phase: when the AO requests to use the PO’s node, the AO must submit the components he is planning to deploy. The PO can then verify the code of those components to ensure platform integrity.

6.2. Underlying platform

The proposed security middleware secures the LooCI middleware [18], which runs on top of the Contiki OS [58]. LooCI is a component-based middleware comprised of an execution environment, a component model and an event-based binding model. The LooCI middleware currently supports the following platforms: Contiki on AVR Ravens and Zigduino’s, SunSPOT sensor nodes, Android smart phones, and OSGi java environments. The LooCI middleware comprises of the following parts as shown on Fig. 9: (1) The LooCI networking layer, (2) the LooCI event manager, (3) the LooCI component runtime, (4) the LooCI management component, and (5) the LooCI code deployment module.

LooCI uses events as the sole mechanism to communicate between components. A LooCI event is comprised of: (1) sender information, (2) extension headers, (3) event type, and (4) event payload. LooCI components are code bundles, which are deployed over the air at runtime using the LooCI code deployment module. Components are only allowed to interact with each other through the usage of LooCI events flowing over the distributed LooCI event bus. Each component can subscribe to typed events or publish typed events himself. Components themselves are oblivious however to the exact senders or receivers of those events. This loose coupling promotes component reuse in different distributed contexts, since there are no hard wired connections between components.

The event manager implements the distributed event bus. It keeps wiring tables, dictating which events of which types from which components have to be sent where, either locally to another component or remote to another node. If it is sent across the network, the event manager transfers the event to the LooCI networking layer. The networking layer translates outgoing events to UDP messages, and incoming UDP messages to LooCI events. Reconfiguration and inspection of the event manager is done by the LooCI reconfiguration manager. All reconfiguration and introspection of the LooCI middleware is done using LooCI management events with the exception of deployment of new applications, which uses an optimised deployment protocol.

The overhead of Contiki OS and LooCI is small, yet significant for resource constrained environments. Contiki OS requires 42688 B of ROM and 9712 B of RAM memory to operate. This includes the uIP stack for networking and other libraries for memory and sensor management. The LooCI middleware requires another 24942 B of ROM and 2644 B of RAM on top of the Contiki OS.
For encryption support, the prototype uses the AVR crypto-lib [66], which offers software implementations of most popular encryption, authentication and integrity algorithms such as AES, SHA, and CMAC. Note that cryptographic algorithm overhead is listed with each protocol, but these algorithms are reused across multiple protocols. To get a full detailed overview of the overhead, please refer to Table 5 at the end of this chapter.

The prototypes operates on top of LooCI, yet the concepts of the architecture and the prototype can be applied to any system that matches the general non-secure system architecture as presented in Section 5.

### 6.3. Secure network initialisation

The secure network initialisation framework (MASY [24]) secures the network initialisation data flow on the node and requires the secure network component. Fig. 10 shows the secure initialisation data flow in detail. This section provides a limited overview of the protocol. For more information, please refer to the paper for more in depth information.

The protocol operates as follows: when a node detects that it has entered a new sensor network, it sends out a hello message containing the identity of its PO, its name and a timestamp. This is sent as one package to reduce network overhead. The hello message is authenticated using the long term node secret key. Currently the identity of the PO is encoded as an IPv4 address. Possible alternatives are DNS and IPv6, however this would require larger string or byte identifiers to be exchanged, increasing overhead. When a node in the network detects this hello message, it forwards this message towards the gateway. Using the token, the NO contacts the PO and negotiates network key and policy information. When the NO and PO reach agreement, the PO creates a small token containing the NO key and the timestamp to ensure freshness. The NO installs this token at the sensor node, enabling the node to communicate securely as discussed in Section 5.1.1.

This section evaluates the communication, memory, and processing overhead of the secure network component. This section evaluates the set-up of the secure key material, assuming other policy deployment actions can be done after the key material has been deployed. The results are compared to TinySec [23] as reference for an efficient WSN secure network layer.

**Communication overhead:** Two types of messages are transmitted across the network: Hello messages, which are 32 bytes and Reply messages, which are 44 bytes. The exact format of these messages is shown in Fig. 11. These figures ignore MAC/IP layer overhead. During communication, all messages can be encrypted, authenticated or integrity checked, depending on the network policy. The communication overhead for these cryptographic primitives is limited to the additional MAC. The current implementation uses an 8 B MAC to ensure message integrity. TinySec does not offer any clear registration functionality. TinySec also uses an 8 B MAC during normal communication, equalling the communication overhead during operations.

**Memory overhead:** The total ROM overhead of the component is 5690 B and consists of: (a) the component implementation (946 B), and (b) overhead for AES encryption algorithm (4746 B). The total RAM overhead of the secure network component is 218 B and consists of the encryption context and message buffers. TinySec has a ROM overhead of 7148 B and RAM overhead of 728 B, which is comparable to the secure network component.

**Processing overhead:** The secure network component operates at hello packet creation, reply packet reception, and when a message is sent to or received from the network. The creation of a hello packet requires the encryption of 16 B of data, requiring one AES operation. This takes a total time of approximately 2 ms using an AES software implementation. Receiving the network reply also only requires the decryption and authentication of 16 B of data, again taking approximately 2 ms. Encrypting and decrypting network packages also requires about 2 ms per 16 B block. To give a comparison, sending a plain
UDP packet also requires ca 2 ms on the AVR raven. TinySec uses two different block ciphers to process messages. For encrypting a 16 B message using RC5 requires 1.04 ms, while SkipJack requires 1.52 ms.

6.4. Secure application deployment

The secure deployment component implements the secure application deployment component as explained in Section 5.1.5. It listens to the network for user connections. When a user connects to it, it first must send a small token containing application hash, resource usage limits and owner information. If the token is valid, the secure deployment component listens for the component code, installs it, and deploys the usage limitation policies listed in the token. If the token is invalid, the connection is closed. This section evaluates the communication, memory, and processing overhead of the secure deployment component and compares it to the Sluice secure deployment protocol [46]. For more information on the SASHa secure deployment system, please refer to the paper [25].

Communication overhead: The only communication overhead of the secure deployment component is the additional token, which is sent before the actual deployment. The total size of this token is 56–78 B. Fig. 12 shows the exact format of the token. The deployment secret key is optional. The token is encrypted and authentication using AES128-CCM with 8 B MAC, using the nodes long term secret key. This is the only mandatory network overhead. Since a typical component has a size of 1000–2000 B, it entails a communication overhead of 2–5%. Sluice uses a similarly sized token of 44 B.

Memory overhead: The secure deployment component has a ROM overhead of 8.1 kB and a RAM overhead of 504 B. Table 1 contains a more detailed memory overhead analysis. To compare, Sluice has a ROM overhead of 9 kB and ROM overhead of 2000 B. The larger RAM overhead of Sluice is due to larger buffer size.

Processing overhead: The delay to deployment with added security consists of the transmission and validation of the application token and validation of the component. Since most application components are more than thousand bytes large, the additional delay for sending a token remains limited to a few percent. No tests were performed to test these delays.

The time overhead of the computation on the node is estimated in the hundreds of milliseconds. Decrypting of one 64 B block takes 8 ms, hashing 6.5 ms. A small

![Fig. 10. Secure network setup key exchange protocol.](image-url)

![Fig. 11. Packet format of the MASY Hello message. Signature is signed, and group key is encrypted with the node unique key, only known to the sensor node and Platform Owner.](image-url)
component update of 1 kB would for example take $\frac{1024}{64} \times (8 + 6.5) \text{ ms} = 232 \text{ ms}$. Sluice uses the Elliptic Curve Digital Cryptography Signature algorithm, an asymmetric key algorithm. Due to this, the verification of a token takes 30–35 s, due to ECC operations, which is significantly longer than the proposed system.

The PO’s server, which manages and grants the tokens and the AO’s management agent which requests the tokens and deploys the applications are both implemented in Java.

### 6.5. Secure application management

The secure application management framework allows for the secure management of the sensor node by authenticating and authorising management messages. The framework consists of: a communication interceptor management component, a service call interceptor, a user data store, and a user management service. For more information on the implementation of the framework, please refer to the paper [26].

The identify users, the system uses a 2 B numeric use-rID. This compact user identification can be node or network unique. Each authenticated message is encrypted using AES128-CCM and has a 8 B MAC in the secure payload header to verify message authenticity and integrity. At service call, the service interceptor intercepts the call and verifies the user is allowed to access the service based on Roles and Attributes of the calling user.

The current prototype implements a hierarchical role based access control scheme. 4 roles are defined: viewer, user, manager, and admin, each with increasing powers of viewing application state, and modifying and deleting applications. A viewer can only see data, users can subscribe to existing data producing components, managers can adapt the settings of these components, and admins can influence the lifecycle (pausing, removing and starting components). These roles allow for a simple, clear and compact description of required role for each functionality.

However, the framework allows parties to easily adapt the system and implement their own access control scheme.

This section evaluates the communication, memory, and processing overhead of the secure application management framework. This approach is compared to a symmetric key approach (TinySec [23]) and a asymmetric key approach (Authenticated Querying (AQ) by Beneson et al. [51]).

#### Communication overhead: The secure application management framework requires additional security information to be added to each management message. This overhead is 14 B and is detailed in Fig. 13. The payload of the messages is currently not padded to ensure the minimal size of reconfiguration messages. TinySec has a message overhead of 8 B, AQ has a message overhead of 20 B, both are comparable to the proposed system (see Fig. 14).

To register a user, it is possible to deploy a user token to the user management system. This requires the transmission of a token over the network containing the following fields: use-rId, partyId, nodeRole, partyRole, user-Key, time-stamp, timeOut, user-timeout, and MAC. This token has a total message size of 40 B and is encrypted with the node’s long term secret key. Installing the user by token is one option to allow new users to be added. Alternatively a service request can be called which has the same communication overhead, since the same data has to be transmitted. To compare, AQ has a user registration token size of 114 B due to ECC signature. TinySec does not mention any user registration token.

### Table 1

<table>
<thead>
<tr>
<th>ROM (kB)</th>
<th>RAM (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES crypto lib</td>
<td>4.2</td>
</tr>
<tr>
<td>SHA-2 crypto lib</td>
<td>1.8</td>
</tr>
<tr>
<td>Deployment component</td>
<td>2.3</td>
</tr>
<tr>
<td>Total ROM</td>
<td>8.1</td>
</tr>
<tr>
<td>Sluice</td>
<td>9</td>
</tr>
</tbody>
</table>

#### Memory overhead: The ROM overhead of the application management component is 12.15 kB, and the RAM overhead is 438 B. Table 2 contains a detailed overview of the overhead. To compare, TinySec has a ROM overhead of 7148 B and RAM overhead of 728 B, AQ has a significantly larger ROM and RAM overhead of respectively 45500 B and 2000 B.

#### Processing overhead: The secure application management framework operates at three point in the call chain: at message reception, at service call, and at message transmission. The overhead of message transmission and reception are almost identical and composed of retrieving the user’s key information from the user data store, and then either encrypting or decrypting the message. The authentication of a message comprises of (a) retrieving the userId from the request message, (b) retrieving the user information from the user data store, and (c) decrypting and verifying the authenticity of the message and takes ca 4 ms for a 32 B management request (mostly...
decryption). Encryption of reply messages takes a similar time, since the main overhead is the encryption of the request. The authorisation of a message is much faster. It comprises of (a) a proxy interception of a service message, (b) retrieving user access rights, and (c) matching access rights with the current request and requires only tens of nanoseconds. TinySec has a message processing overhead of only 1.04 ms or 1.52 ms, due to the usage of different encryption algorithms (RC5/SkipJack), and different hardware.

A user can register himself with the system using a token. This message is not intercepted by the Communication Interceptor to allow anyone to send such a token. However, the service itself verifies the integrity and validity of the token by decrypting and authenticating the token. Since the token size is about 40 B this takes about 6 ms to decrypt and verify the token. To compare, AQ requires 440000 ms to validate the user registration token due to the usage of different encryption algorithms (RC5/SkipJack), and different hardware.

6.6. Secure application communication

The secure application communication framework ensures that the application communication of the different application components is transmitted securely. In the LooCI Component middleware, communication happens using typed events, source component and destination node. As stated before, this component distinguishes between encryption contexts, outgoing policies and incoming policies. Encryption contexts identify a security context which entails algorithm type, algorithm identification, key material and security level. Outgoing policies dictate which events should be transmitted on which channel based on event type, source component and destination. Incoming policies dictate which level of security is required for certain events, and which events are allowed to pass, based on source node, source component, event type and context. An overview of the decision logic can be found in Fig. 16.

When a message is dispatched from the application middleware, it is checked against the outgoing policies. If the event matches an outgoing policy, the associated channel is retrieved and the event is processed according to the channel parameters. If an event matches multiple policies, only the first matching policy is considered, which is the policy which has been on the node the longest. Overlapping policies can be detected based on the source, destination and event type of the channel.

When an event comes up from the network stack, the encryption context is checked against relevant incoming
policies. If the context is present, the matching security primitive is performed. Once the verification is completed, the event is matched against the incoming policies. The incoming policies dictate the minimal level of security that is required for an event to pass. It is also possible to dictate a blocking policy that blocks the matching events, unless another policy explicitly allows it. If the context is not present, and the event is not encrypted, than the event is checked against the blocking policies.

This section evaluates the communication, memory, and processing overhead of the secure application communication component, and compares it with TinySec [23].

Communication overhead: The prototype implements three types of security policies: integrity, authenticity and confidentiality. Each message is identified by a (a) security information header, containing the channel ID (2 B), and (b) a security payload, containing the hash or mac payload, as shown in Fig. 15. The size of the security payload depends on the exact policy installed. The prototype allows for a flexible choice of security and supports multiple lengths of security messages. For integrity and authenticity a security payload of 4, 8, 16 and 32 B is supported, for confidentiality a MAC size of 0, 4, 8 and 16 B is supported.

The total message overhead is 4, 10, 14, 22 or 38 B depending on the security payload size, and thus the level of security. The AO can choose his level of security depending on the security requirements of the application, the available resources of the sensor node and the preferences of the POs involved in the deployment. TinySec has a fixed MAC overhead of 8 B.

Memory overhead: The prototype has total ROM overhead of 10900 B and a RAM overhead of 97 B assuming 2 channels, 1 incoming and 1 outgoing policy. Transmission buffers and encryption buffers are not included in this, and depend on specific transmissions and encryption protocols. Table 3 shows the overhead figures in more detail. Note that 7744 B of the implementation overhead is taken up by cryptographic algorithms, which are reused in other parts of the security middleware. TinySec has a ROM overhead of 7148 B and RAM overhead of 728 B.

Processing overhead: Processing a 32 B message takes: to encrypt/decrypt the message using AES: 4 ms, to authenticate or verify a message using AES-CMAC: 4 ms, to verify message integrity using SHA1: 13.6 ms. While this processing overhead is significant, it is still only a few milliseconds of processing overhead. In most use cases this is a tolerable overhead. TinySec has a message processing overhead of only 1.04 ms or 1.52 ms, due to the usage of different encryption algorithms (RC5/SkipJack).

6.7. Monitoring and enforcement

The monitoring and enforcement component FAMoS [27] monitors node usage by different components and users and allows policy enforcement. An overview of the monitoring and enforcement component is shown in Fig. 17.

The monitoring and enforcement framework operates by instrumenting the functions required to use the network, memory and sensors. It defines multiple different hooks in the network to be able to monitor performance on different levels of the network, including the MAC and IP layer. These hooks call the monitoring component when data requests or data passes through them.

The monitoring and enforcement component on the node is comprised of four blocks: the core, the profile, the policy and the flush component. The monitor core block maintains the buckets, which are basically resource counters that keep track of the resource usages. A bucket consists of a 32 bits bucket identifier and a 16 bits counter. The range of bucket identifies allows for encoding of different kinds of usage, and parts of the identifier can contain for example user id or component id. The profile block contains the logic which decides which buckets need to be incremented. The policy block checks whether the node usage is allows, and updates internal policy buckets, which operate in a leaky bucket fashion. Finally, the core adds all bucket updates to its internally saved reporting buckets. On a fixed schedule the monitor flush block flushes the acquired reporting buckets or when the 16 bits buckets are almost overflowing.

The monitoring and enforcement component can potentially operate independent from the other proposed pillars. However, by using the additional security pillars, the monitoring traffic benefits from the additional security features presented, such as secure deployment and secure end-to-end communication.

This section evaluates the communication, memory, and processing overhead of the monitoring and enforcement component implementation on an AVR raven and compares it to the Sympathy network debugger [54].

Communication overhead: The communication usage can be divided into two parts: (a) the overhead to send a monitoring flush packet and (b) the overhead to install a policy. The overhead to send a monitoring flush packet depends on the flush interval. A flush packet has a minimal size of 8 B. Each bucket that contains data extends this packet size with 6 B. On top of this is the network overhead, which can be 30–40 B of IP and MAC headers. Deploying a policy has a communication overhead of 8 B per policy. Sympathy does not explicitly mention monitoring packet size or overhead. Assuming 4 B per timestamp and 2 B per counter, Sympathy has a monitoring message size of 20 B, which is comparable to the proposed component.

Memory overhead: The basic framework without policy enforcement requires 3290 B of ROM and 216 B of RAM. The policy enforcement framework requires another 2140 B of ROM and 221 B of RAM. This is for a minimal monitoring policy. More complex policies will require

<table>
<thead>
<tr>
<th>Security Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity</td>
<td>Verifies the message integrity</td>
</tr>
<tr>
<td>Authenticity</td>
<td>Verifies the message authenticity</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Encrypts the message</td>
</tr>
</tbody>
</table>

Fig. 15. Packet format of a SecLoCoCI application communication message.
more ROM and RAM due to the larger amount of monitoring logic, and amount of buckets to monitor. Sympathy has an overhead of 47 B of RAM and 1558 B of ROM. This lower overhead is due to the fact that Sympathy has fixed monitoring which cannot be updated or changed, and cannot enforce policies, so the monitoring and enforcement component has a slightly higher overhead for some additional features.

**Processing overhead:** The processing overhead of the prototype is minimally 57 ns. This can run up to 200 ns or more depending on the complexity of the monitoring and enforcement policies. The default case of sending a packet over the network takes on average 1260 ns network processing, and 2228 ns to actually transmit the message, totalling at 3488 ns. 200 ns is only a 5.8% additional overhead. Sympathy does not mention any processing overhead, but due to the static and simple monitoring policy of Sympathy, this paper assumes it can be compared to a very simple monitoring policy, requiring 50 ns.

### 7. Discussion and evaluation

This section presents the integrated evaluation, and discusses these results. First it provides a general overview of
the overhead of the integrated prototype and performs an applied evaluation of this comprehensive middleware using the typical use case proposed in Section 5.2. Next it discusses how the architecture fulfils all non-functional requirements as listed in Section 3 and examines how the proposed security architecture and middleware meets the different proposed security requirements by analysing potential attacks. Finally this section looks at some interesting future work.

7.1. Prototype implementation

This section briefly presents the integrated security architecture and prototype of the components as discussed in the previous sections and evaluates on the integrated figures. The SecLooCI architecture has seven additional components over the standard LooCI middleware, in accordance with the general security architecture and as shown in Fig. 3:

- **The secure network component** secures the network initialisation, which is transparent to the upper and lower layers, and allows the creation of a sensor node registry at the gateway.
- **The communication interceptor component** performs the authentication and encryption for application and management communication.
- **The secure deployment component** secures the application deployment.
- **The security manager component** offers security management services such as user installation, channel and policy management for the application security, and policy management for the monitoring and enforcement component.
- **The authorisation and interception service proxy** provides the interception points necessary to enable authorisation of management communication and monitoring of application service usage.
- **The monitoring and enforcement component** monitors the network and the application service usage, and potentially also the different users of the network.
- **The security data store component** contains the necessary user and policy information needed for the application and management message security protocols to operate.

These seven components are implemented and integrated on the AVR Raven running the Contiki OS and the LooCI Component Middleware. Table 4 shows an overview of the overhead of the SecLooCI middleware compared to the unsecured LooCI and Contiki. Table 5 shows the ROM and RAM overhead of the different components of the middleware. The total additional overhead for the security middleware including cryptographic primitives is 27,372 B of ROM and 15,252 B of RAM memory. This is a significant amount of memory for these memory constrained devices. However, the security features which are offered by this middleware are necessary for the next generation of sensor network applications, and the implementation of the middleware shows that it is feasible to implement the required features on the targeted memory constrained devices, with a significant yet acceptable overhead.

Note that 28.3% of ROM and 9.3% of RAM is due to cryptographic primitives. A current trend is to implement these algorithms in hardware, to reduce processing cost and memory overhead. This would significantly reduce the overhead of the framework.

7.2. Test scenario and measurements

This section implements the end to end scenario presented in Section 5.2. This test is a prototypical test for a single hop network, which we use as our default test scenario as discussed at the start of Section 6. The time overhead and message size of each node interaction is stipulated in this scenario. Only node interactions are listed, excluding back-end to back-end communication, since only sensor node interactions influence sensor node lifetime. The memory overhead of the different components, which is mostly static, can be found in the previous section. For each interaction the total cost is listed. For an in depth view of all the different costs, please refer to Table 6. The test setup consists of two sensor nodes and a gateway in a one hop network. The scenario does not consider multi-hop networking. Network and MAC layer overhead is not considered.

**Network setup** The first step in network setup is the beacon that is sent across the network. This beacon currently has a size of 8 B containing the MAC address of the gateway. Once received, the node generates the node token of 32 B, and transmits it. This takes ca 4 ms, mostly encryption. The NO receives and processes the token, contacts the PO, receives a network token of ca 44 B, and transmits it back to the node which takes on average ca 100 ms. Most of this time the node is waiting for a reply to his key request. The total cost is 32 B sent, 52 B received, taking ca 100 ms, of which 12 ms is spent computing by the node.

**Application deployment** The next node interaction is the customs officer who wants to deploy a new application component. In this case, the customs officer deploys a localisation component, which listens to location events, does some very simple distance estimation in two dimensions based on the delay of the event and beacon location, and publishes its location every 60 s if it has received a beacon. A prototype component has a size of 2121 B. To deploy the component, the customs officer first sends the application token, with a size of 56 B, and then sends the

| Table 4 | SecLooCI middleware memory overhead. The overhead of the security suite is comparable to the overhead of the non-secure middleware, but clearly still within the possibilities of a low power sensor node. |
|——— |——— |——— |——— |
| | ROM (B) | % | RAM (B) | % |
| Contiki operating system | 42688 | 32.6 | 9712 | 59.3 |
| LooCI middleware | 24942 | 19.0 | 2644 | 16.1 |
| SecLooCI security suite | 27372 | 20.9 | 1525 | 9.3 |
| Total binary | 95002 | 72.5 | 13881 | 84.7 |
| Total available on AVR Raven | 131072 | 100.0 | 16384 | 100.0 |
component. The node only sends a success byte and the component identifier, totalling 2 B. This process takes about 12.3 s. Only about 8 ms were spent decrypting the token, and 215 ms performing the hashing. This totals at ca 2177 B received, 2 B sent, taking ca 12.3 s, of which 223 ms spent on crypto operations, and ca 10 s spent writing to flash memory. The remaining time is communication overhead.

Application management The following interaction is the management of the users and node applications. The necessary operations are: deployment of the user token by the Customs Officer, entering the five beacon locations in the application, entering the necessary subscriptions, adding the security policies (incoming channel and policy, outgoing channel and policy, and monitoring policy), and activating the component. The policies use an 8 B MAC for encryption and authentication. Each of these interactions is a service call by the customs officer to the management component. A total of 14 configuration requests are needed. The prototype performed this configuration in an automated way, taking about 2.2 s. Each request requires a minimal overhead of 14 B for security headers, and 6 of the 14 messages are security management messages. The total cost is 587 B sent, 573 B received, taking ca 2.2 s.

Table 5
Detailed SecLooCI middleware memory overhead. A significant amount of ROM and RAM is used for the encryption algorithms. Hardware implementations could reduce this overhead.

<table>
<thead>
<tr>
<th>Component</th>
<th>ROM (B)</th>
<th>%</th>
<th>RAM (B)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure network component</td>
<td>946</td>
<td>3.5</td>
<td>218</td>
<td>14.3</td>
</tr>
<tr>
<td>Communication interceptor component</td>
<td>2034</td>
<td>7.4</td>
<td>129</td>
<td>8.5</td>
</tr>
<tr>
<td>Secure deployment component</td>
<td>512</td>
<td>1.9</td>
<td>20</td>
<td>1.3</td>
</tr>
<tr>
<td>Security manager component</td>
<td>2780</td>
<td>10.2</td>
<td>295</td>
<td>19.3</td>
</tr>
<tr>
<td>Authorisation/interception proxies</td>
<td>3296</td>
<td>12.0</td>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td>Security data store component</td>
<td>1558</td>
<td>5.7</td>
<td>238</td>
<td>15.6</td>
</tr>
<tr>
<td>Monitoring component</td>
<td>3290</td>
<td>12.0</td>
<td>216</td>
<td>14.2</td>
</tr>
<tr>
<td>Policy enforcement component</td>
<td>2140</td>
<td>7.8</td>
<td>221</td>
<td>14.5</td>
</tr>
<tr>
<td>Middleware tools</td>
<td>3072</td>
<td>11.2</td>
<td>38</td>
<td>2.5</td>
</tr>
<tr>
<td>Encryption algorithms</td>
<td>7744</td>
<td>28.3</td>
<td>142</td>
<td>9.3</td>
</tr>
<tr>
<td>Total SecLooCI usage</td>
<td>27372</td>
<td>100.0</td>
<td>1525</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table 6
Overview of the SecLooCI middleware overhead. Daily overhead calculated assuming one incoming and outgoing message per minute. Table shows that processing overhead is comparable to one day of operation. Transmission overhead is equal to about 19 days of operation.

<table>
<thead>
<tr>
<th>Action</th>
<th>LooCI</th>
<th>%</th>
<th>SecLooCI</th>
<th>%</th>
<th>Crypto</th>
<th>%</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network setup sent(B)</td>
<td>0</td>
<td>0.0</td>
<td>16</td>
<td>50.0</td>
<td>16</td>
<td>50</td>
<td>32</td>
</tr>
<tr>
<td>Network setup received(B)</td>
<td>0</td>
<td>0.0</td>
<td>36</td>
<td>50.0</td>
<td>16</td>
<td>50</td>
<td>52</td>
</tr>
<tr>
<td>Network setup processing(ms)</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>100</td>
<td>12</td>
</tr>
<tr>
<td>Deploy sent(B)</td>
<td>2</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>2</td>
</tr>
<tr>
<td>Deploy received(B)</td>
<td>2121</td>
<td>97.4</td>
<td>16</td>
<td>0.7</td>
<td>40</td>
<td>1.8</td>
<td>2177</td>
</tr>
<tr>
<td>Deploy processing(ms)</td>
<td>10000</td>
<td>97.8</td>
<td>0</td>
<td>0</td>
<td>223</td>
<td>2.2</td>
<td>10223</td>
</tr>
<tr>
<td>Mgt sent(B)</td>
<td>172</td>
<td>29.3</td>
<td>303</td>
<td>51.6</td>
<td>112</td>
<td>19.1</td>
<td>587</td>
</tr>
<tr>
<td>Mgt received(B)</td>
<td>164</td>
<td>28.6</td>
<td>297</td>
<td>51.8</td>
<td>112</td>
<td>19.5</td>
<td>573</td>
</tr>
<tr>
<td>Mgt processing(ms)</td>
<td>0.42</td>
<td>0.7</td>
<td>0.28</td>
<td>0.5</td>
<td>56</td>
<td>98.8</td>
<td>56.7</td>
</tr>
<tr>
<td>Total sent(B)</td>
<td>174</td>
<td>28.0</td>
<td>319</td>
<td>51.4</td>
<td>128</td>
<td>20.6</td>
<td>621</td>
</tr>
<tr>
<td>Total received(B)</td>
<td>2285</td>
<td>81.5</td>
<td>349</td>
<td>12.5</td>
<td>168</td>
<td>6.0</td>
<td>2802</td>
</tr>
<tr>
<td>Total processing(ms)</td>
<td>10000</td>
<td>97.2</td>
<td>0</td>
<td>0</td>
<td>291</td>
<td>2.8</td>
<td>10291</td>
</tr>
<tr>
<td>App comm sent per message(B)</td>
<td>9</td>
<td>39.1</td>
<td>6</td>
<td>26.1</td>
<td>8</td>
<td>34.8</td>
<td>23</td>
</tr>
<tr>
<td>App comm rec per message(B)</td>
<td>9</td>
<td>39.1</td>
<td>6</td>
<td>26.1</td>
<td>8</td>
<td>34.8</td>
<td>23</td>
</tr>
<tr>
<td>Time to process a message(ms)</td>
<td>5</td>
<td>38.5</td>
<td>0</td>
<td>0.0</td>
<td>8</td>
<td>61.5</td>
<td>13</td>
</tr>
<tr>
<td>App comm sent per day(B)</td>
<td>12960</td>
<td>39.1</td>
<td>8460</td>
<td>26.1</td>
<td>11520</td>
<td>34.8</td>
<td>33120</td>
</tr>
<tr>
<td>App comm rec per day(B)</td>
<td>12960</td>
<td>39.1</td>
<td>8460</td>
<td>26.1</td>
<td>11520</td>
<td>34.8</td>
<td>33120</td>
</tr>
<tr>
<td>Daily message processing(ms)</td>
<td>7200</td>
<td>38.5</td>
<td>0</td>
<td>0.0</td>
<td>11520</td>
<td>61.5</td>
<td>18720</td>
</tr>
</tbody>
</table>

Application communication At this point, the application is running and communicating over secured channels. When a message enters the node, it is intercepted by the communication interceptor, the incoming channel is retrieved, the message decrypted and checked against incoming policies. This takes about 4 ms. Then the message is sent up to the LooCI Event Manager, which delivers it to the localisation component. When the timer expires, the localisation component calculates its current position based on available information, and broadcasts it. The broadcast is logged by the monitoring component, but only takes a few nanoseconds. The event manager receives the event and routes it to the network. The communication interceptor intercepts it, checks incoming policies and encrypts it, which takes again about 4 ms. The largest part of this overhead is encryption. The total process of sending only takes about 5 ms from event creation by the localisation component, until it is sent over the wire. The send overhead is the cost for encrypting and authenticating the event, requiring 14 B. The message itself is 9 B. Hence, the total cost is 23 B sent, 23 B received, taking ca 13 ms. Assuming one message per minute, this becomes a daily total cost of 33120 B sent, 33120 B received, and 18720 ms spent processing.
7.3. Architecture discussion

This section looks at how the proposed SecLoocI architecture handles the different non-functional concerns as listed in Section 3.3. It shows that the proposed architecture and reference implementation meet the requirements.

7.3.1. Evolvability

The middleware allows new sensor network applications to be added at runtime, and to securely manage these applications by for example adding new subscriptions, or adding or removing application security policies. The evolution of network applications is enabled by the secure deployment of software components, and secure management framework. The secure deployment framework ensures that only components that have been approved by the Platform Owner (PO) can be deployed, and the PO can specify additional resource constraints to ensure those components do not use too many node resources. The secure management framework ensures that users can only perform authorized evolution actions.

7.3.2. Performance

The integrated prototype shows that the overhead of the SecLoocI middleware is small enough to still fit on micro-controller level of devices. Section 7.1 shows that the overhead of the static system is significant, but comparable to other security solutions proposed by related work. Section 7.2 shows that the prototype communication overhead from security is fairly limited, and exists mostly of necessary security data. This shows that sharing nodes with multiple parties does not cause a large overhead compared to a secure single user single party environment.

7.3.3. Transparency

All security features are provided at a middleware level and are policy driven. The different owners can specify their security preferences in high level, expressive policies. These policies are then deployed in a compressed (binary) format on the nodes, where they are enforced wherever applicable. Additionally, because the security policies are enforced on a middleware level, application developers do not need to be concerned about security, nor do they have to implement their own protocols. This significantly increases security since usually application developers are not security experts. Security by policy costs some performance as shown in Section 7.1, but the evaluation has shown this cost is manageable.

7.3.4. Compatibility

The application deployment features, and the fine grained application security features ensure that the required communication patterns can be supported. By being able to deploy high level policies at both the sender and the receiver, one to one, many to one, one to many, or many to many communications can be secured. Additionally, these communication patterns can be secured in an end-to-end fashion, ensuring that only those parties which are authorized to receive the data, are able to encrypt/decrypt the messages. Additionally, by having this policy driven security, the system can differentiate between information which can be shared with all, most, some or no other parties.

7.3.5. Flexibility

The application communication security policies provide multiple different levels of security with multiple different algorithms. Security policies specify whether the communication needs to be secured using integrity, authenticity, or confidentiality ensuring protocols. Additionally the system can chose the length of authentication code, to be able to make a trade-off between communication size and security. The application user can chose which policy to apply depending on the application requirements and performance costs. Additionally by using policies and being able to support and deploy new and other security algorithms, the system or user can update and change algorithms in its network, to be able to meet changing security requirements.

7.3.6. Shareability

The network setup, deployment and management modules clearly offer support for multiple users from multiple different parties to securely share sensor node services and ensure secure data exchanges. The network setup data flow allows network owners to share their network with multiple Platform Owners. The secure deployment, secure management and application monitoring data flow allow Platform Owners to share their node services with multiple other application owners. By having end to end security policies, application owners can still be sure of their application’s communication security, even while using relaying functionality of untrusted nodes, as long as he trusts the Platform Owners of which he uses the services.

7.3.7. Fault tolerance

The architecture can continue to operate with reduced functionality depending on which party and which functionality becomes unavailable. When the network owner’s registration server goes offline, then no new nodes can join the network. However, all nodes that are in the network can continue to operate, and send data to their owners assuming the communication with the Internet is available. AOs can deploy an instantiate new applications, and manage existing application by contacting the relevant POs. When an Internet connection becomes unavailable, then no users can communicate with the nodes, nor can the nodes communicate with back-end infrastructure. However, any local functionality will continue to operate. Additionally, some nodes will be equipped with alternate Internet connections such as GSM and satellite. These nodes can then set up a new network, and can offer their networking service to other nodes and owners.

When a Platform Owner goes offline, then his nodes can no longer be used in new applications. However, existing applications will continue to run. Additionally it is possible for application owners to request usage tokens in advance, and deploy this token when he actually wants to deploy the application. In this case, the Platform Owner does not need to be available at the moment of deployment. Additionally in many networks, there will be nodes of many different Platform Owners available, so only a limited part of
the network will become unavailable for new deployments.

When an Application Owner becomes unavailable, then the applications owned by that application owner will not be able to send their data back to the owner, nor will the server monitor the existing applications. The autonomous behaviour of the nodes will however continue to operate.

7.3.8. Summary

This evaluation shows the SecLooCI architecture meets the proposed requirements. SecLooCI offers a comprehensive sensor node security middleware enable shared usage of sensor networks and is able to run on low power sensor nodes.

7.4. Security analysis

This section performs a security analysis by looking at the security guaranteed by the middleware while facing the three types of attackers as identified in Section 3.2: the network attacker (NA), the physical attacker (PA), and the insider attacker (IA). This section applies the STRIDE threat model \[67\] and assigns each threat to their main actor(s): Spoofing identity (NA + PA), Tampering with data (NA + PA), repudiation (IA), Information disclosure (NA + PA + IA), Denial of service (NA), and Elevation of privilege (IA). We chose STRIDE because it is an industry standard, proposed by Microsoft, and recommended by OWASP \[68\], the Open Web Access Security Project with members such as HP, IBM, and Oracle. OWASP recommends STRIDE because it works well for addressing the unique challenges facing web application security and is simple to learn and adopt by designers, developers, code reviewers, and the quality assurance team. Next this section reviews for each of the attackers the potential attacks they can perform as identified by STRIDE.

A network attacker is an attacker who is not part of the network. He can potentially (1) Spoof identity for managing nodes, (2) Tamper with application or management data, (3) Disclose information, and (4) Perform denial of service attacks. The system prevents the first three attacks, and offers potential mitigation against the forth attack. First, the secure application management framework prevents identity spoofing, by requiring all management actions to be encrypted and authenticated. Since we assume that attackers cannot break encryption algorithms, we must conclude that an NA cannot perform identity spoofing. Second, the network communication framework ensures that no information is disclosed and prevents tampering. All network communication is encrypted with integrity protection. This ensures that a network attacker cannot read message contents, only see that messages are passed, nor tamper with messages without being detected. Finally, the network attacker can perform denial of service (DoS) attacks. The current prototype offers no specific protection against such attacks. The proposed middleware does have tools that can detect DoS attacks. The monitoring and enforcement framework can monitor for DoS attacks by monitoring failed authorisation or excessive network activity, and when detected can try to notify the AO, and potentially temporarily disable the network stack to save energy.

A physical attacker on the other hand, can physically probe a node and retrieve all key material that is present on that node. When this happens, all keys used for network and application security will be available to the physical attacker. He can then perform the following attacks: (1) Spoofing identity, (2) tampering with data, and (3) disclose information. Spoofing identity is prevented by requiring all nodes to have separate key material for management operations. Since a physical attacker only gets the management keys from the nodes he broke, he does not have access to the other node keys, and cannot spoof identity.

A physical attacker can tamper with data and disclose information of all messages that are encrypted with keys known to the broken nodes. For application communication, only the end points know the used encryption keys. So if the physical attacker breaks a node that is not part of the confidential communication group, he cannot read the encrypted messages. While the current system does not currently have any detection for physical attackers, the framework has several ways to recover network security when it has been detected that a node is compromised. The system can recover from a node capture by performing a re-keying operation. Once a node breach is detected, the node owners can re-key all confidentiality keys, and leave the breached node out of the re-keying process. Since each node has unique management keys, the breached node cannot read the re-keying messages sent to other nodes, and prevents the physical attacker from continuing to read or tamper with the messages.

Finally, an insider attacker has key material to manage nodes, and is permitted to do certain management operations. This allows him to the following attacks: (1) Repudiation and (2) elevation of privilege. The secure application management framework ensures non-repudiation since the insider cannot deny performing an operation. Since keys are unique per user and only known to the user, the node and the node owner, the user cannot deny performing an operation on the node to the node owner. Secondly the secure application management framework prevents elevation of privilege, since privilege elevation operations are only permitted to the node owner.

To summarise, the security framework prevents many possible attacks from network, physical and internal attackers. Most of the possible attacks of network and insider attacks are prevented, and, while the framework cannot prevent physical attacks, it offers the tools to recover from such attacks.

7.5. General discussion

The previous section has shown that the SecLooCI architecture and implementation achieve the design goals. The architecture discussion verifies that the architecture meets the non-functional requirements. The security analysis shows that the middleware offers a wide range of countermeasures against many types of attacks. The prototype implementation overview confirms that the security middleware architecture can be implemented on resource constrained hardware. The integrated scenario demonstrates
that, while there is some significant overhead during management, most cost is still incurred by security metadata and cryptographic processes during application runtime, which are necessary even in a single party environment to ensure the secure operation of the WSN. This allows us to conclude that it is feasible to install a comprehensive security middleware on current resource constrained nodes, which allows multi-user node interactions. This enables the roll out of secure shared sensor networks, with dynamic trust setup, limiting of node usage, and charge-back of said node usage.

Next this section discusses the trust requirements for the security middleware, the energy consumption of the security middleware, and some limitations of the current prototype.

**Trust requirements** This system requires the different parties to trust each other, in order for the different data flows to succeed. By default, we assume that NOs and POs will only offer their network and node services to parties which they trust and only use trusted services. However, the architecture allows a trust by default approach, where unknown parties are trusted, so this trust does not need to be pre-established. There are also different levels of trust. A PO can trust an NO to use its networking service, but not trust the NO sufficiently to allow the NO to use its node services. Additionally a PO could trust some AOs to that extend to allow all sensors and actuators on its nodes, while the PO has lower trust in other AOs, only allowing them to use a limited amount of sensors, likely with stronger policies enforced on the node.

In secure environments, we do assume that there has to be an increased level of trust between the NOs, POs, and AOs. This does not mean that every PO needs to manually sign an agreement with every AO and PO. We foresee a system of trusted third parties (a certificate authority), with which the parties register. The different owners trust these authorities and all parties certified by them, providing certain services to all parties certified. Potentially there can even be multiple levels of certification, allowing multiple levels of trust. This system has proved to be quite scalable, and provides a fairly good level of security.

**Energy consumption** We estimate that the additional energy consumption of the prototype is relatively low compared to a non-secure system. De Meulenaer et al. [69] have shown that the cost of encryption, which is the main processing overhead of the security middleware, is relatively small compared to communication overhead when using symmetric encryption algorithms (less than 5%), and that the main cost of communication is from waiting and synchronisation. The actual cost of transmission and reception is only ca 10% of the total communication overhead.

The security middleware sends a limited amount of additional messages during the setup phase to set up key material and security policies. The test scenario sends 2459 bytes of functional data, compared to 964 bytes of security (and crypto) data, which is only 28%. However, this is a one-time cost when setting up the application. The setup data is only ca 5% of communication relative to a single day of normal operation.

After the set-up phase, the security middleware mainly encrypts messages or appends some security meta-information, causing some overhead. However, most of this overhead are message authentication codes and general security metadata, which are also necessary in secure single user environment. The additional overhead for multi-user interactions is estimated on ca 10%. The monitoring framework sends additional messages, however previous research has shown that the overhead of a sufficiently accurate view is generally low compared to the total amount of traffic as generated by the application [27], and can be parametrized to ensure minimal overhead. This allows us to conclude that the network lifetime will not be impacted significantly by the additional communication and processing required for the security middleware.

**Current limitations** Designing and building the prototype has shown us some interesting limitations. The most crucial implementation limit is the memory constraints. All elements, from userId to permissions, party information, component resource restrictions, consume some limited RAM memory. The current implementation only offers 2500 bytes of free RAM. This causes the current implementation to only support about 20 concurrent users and parties per node, with about a dozen components, each with a few resource consumption policies. With regards to static memory, only 36 kB of ROM remains, which translates to space for the code of nine components. Note that this code can be instantiated multiple times, so one can run the same code with different properties (e.g. filters with different high and low borders).

Due to these memory limitations, certain trade-offs have been made. The most severe one is that the current access control prototype limits the access control scheme to a hierarchical role based scheme. This allows for more efficient storage, but limits the access control policy options. In the current system, managers cannot assign per user and per service access control, but can assign users to roles with a party, and permit access to services of a party to certain roles.

Another limitation encountered while creating the prototype on this resource constrained hardware is that asymmetric cryptography is currently too resource consuming. This prohibits the usage of certificates to communicate with the sensor nodes. Certificates currently used in production systems are still to heavy weight to be of much usage, requiring too much time to verify and authenticate users. To mitigate this, the prototype uses tokens and industry grade symmetric key cryptography to maintain a high level of security.

None of these limitations are however fundamental to the architecture, but are trade-offs made to implement the architecture on currently available resource constrained hardware. When more resource rich embedded devices become ubiquitously available, the implementation of the architecture can be updated to make different trade-offs in favour of more expressive policies and easier user management, with no changes to the core architecture. However this implementation teaches us what is the minimal amount of resources required when building a secure, dynamical, networked sensor node system.
This paper proposes a comprehensive secure software architecture for wireless sensor nodes. We have implemented this architecture in a security middleware. These results could definitely benefit from complementary research: on sensor hardware and on management interfaces for WSN.

In order to further guarantee a secure and efficient shared platform some hardware provisions are needed that are not present in the current generation of sensor nodes. First is the need for secure, concurrent, and isolated execution of the application components on the sensor node. Future research should examine the possibility for creating hardware secured and isolated execution environments for sensor node applications, thus enabling the isolated execution of components of different users. Secondly, additional research should investigate how to efficiently manage the evolution of cryptographic algorithms on wireless sensor nodes. While current algorithms are considered safe and are implemented efficiently in hardware, long term deployments might need to receive upgrades to installed cryptographic algorithms during the lifetime of the WSN. It is very important to study how this will affect the sensor nodes.

The second avenue of future work is in the domain of human–computer interaction. In order to operate and manage large scale multi-party sensor networks, there is a clear need for a user interface that allows end-users to provide their applications and to express the functional and the security requirements at a high level. These high level declarations could then be processed by a planning infrastructure on the server side, possibly involving the negotiation of potential deployments between the Application Owners and Platform Owners. It is unclear how the non-expert end-users should interact with the whole ecosystem of sensing and processing capabilities. In this context, the security scope can be divided into two parts: (1) application security specification and (2) management security specification. Firstly, end users must be able to easily declare application security specifications. Similar to the case of personal computing devices, most end-users will not create their own applications, but rather install and parameterise existing applications to suit their own needs. However, this parameterisation and especially the specification of security requirements (confidentiality/integrity/authenticity of data) remains challenging. Secondly, end-users should be able to specify which parties have access to their systems; they should be able to set up a secure shared environment with minimal effort. Users need to be able to specify trust relationships and node, network and party wide resource policies in a simple yet expressive fashion. This requires research in how end-users are able to express their trusted parties and resource limitation requirements, which subsequently must be enforced in the WSN using the proposed WSN security middleware.

8. Conclusion

WSNs in real world applications have to support multiple users interacting with the sensor nodes in a dynamic, evolvable network that runs multiple applications simultaneously. However, current research in application development, management and security does not provide an adequate solution to allow wireless sensor networks to be used in such dynamic multi-user and multi-application ecosystems. This paper identifies five key data flows that need new additional support from the node middleware to ensure safe and secure multi-user node interactions: (1) network initialisation, (2) application deployment, (3) application management, (4) application service usage, and (5) application communication. Mainly, the paper presents SecLoocI, the comprehensive middleware that supports multi-party usage of resource-constrained infrastructures.

The paper presents the concepts, architecture, implementation and integration of the necessary middleware components in order to secure these data flows, and shows the interaction and dependencies between them. The implementations are extensively evaluated with regards to communication, memory and processing overhead. These evaluations show that, while the overhead of supporting multi-party interactions in WSNs is significant, it is acceptable to roll out even on current generation resource constrained devices. This shows the feasibility of multiple parties directly and securely interacting with shared, resource constrained, embedded sensor nodes.

Acknowledgements

Research funded by a Ph.D. Grant of the Agency for Innovation by Science and Technology (IWT). This research is partially funded by the Research Fund KU Leuven, the EU FP7 project NESSoS, and iMinds (a research institute founded by the Flemish government). With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). The research is conducted in the context of the COMACOD project.

References

Christophe Huygens is a part-time guest professor in the Department of Computer Science at Katholieke Universiteit Leuven in Belgium and member of its DistriNet research group. He’s also a private consultant for network and systems security and operational risk-control tools. His research focuses on policy and risk, and how to manage, monitor, enforce, and control policy from an infrastructure perspective. Huygens has a PhD in Computer Science from Katholieke Universiteit Leuven.

Wouter Joosen is full professor at the Department of Computer Science of the Katholieke Universiteit Leuven in Belgium, where he teaches courses on software architecture and component-based software engineering, distributed systems and the engineering of secure service platforms. His research interests are in aspect-oriented software development, focusing on software architecture and middleware, and in security aspects of software, including security in component frameworks and security architectures.