Protected pointers in wireless sensor networks

Gianluca Dini, Lanfranco Lopriore
Dipartimento di Ingegneria dell'Informazione, Università di Pisa, via G. Caruso 16, 56126 Pisa, Italy
E-mail: {g.dini | l.lopriore}@iet.unipi.it

Abstract — With reference to a distributed architecture consisting of sensor nodes connected by wireless links in an arbitrary network topology, we consider a segment-oriented implementation of the single address space paradigm of memory reference. In our approach, applications consist of active entities called components, which are distributed in the network nodes. A component accesses a given segment by presenting a handle for this segment. A handle is a form of pointer protected cryptographically. Handles allow an effective implementation of communications between components, and key replacement. The number of messages generated by the execution of the communication primitives is independent of the network size. The key replacement mechanism is well suited to reliable application rekeying over an unreliable network.

Keywords: cryptographic key, key replacement, sensor node, single address space, symmetric-key cryptography.

1. INTRODUCTION

We shall refer to a distributed architecture consisting of sensor nodes connected by a wireless network [1]. In an architecture of this type, stringent restrictions exist in terms of the hardware complexity, computational power and energy consumption of each node [13]. Memory is a scarce resource, and hardware limitations prevent utilization of an intrinsically complex device such as a memory management unit. Efficiency of the networking protocols in both terms of processing power and storage requirements is a significant parameter [6]. The number of messages transmitted across the network must be kept low, as a consequence of the high energy cost of wireless communications [18]. Consequently, the design of a wireless sensor network is largely different from that of a classical wireless or wired-line network.

We shall model a distributed application as the result of the joint activities of *components* distributed in the network nodes. A component is an active entity that generates memory accesses; thus, a component can be a scheduled computation [2], or, in an event-driven environment, the activity produced by a function activated by a hardware interrupt [8], [17]. Each network node can host a single component. A peculiar problem of wireless sensor networks is the distribution and management of the cryptographic keys, which are necessary for message transmission among the network nodes [22], [28]. Lack of physical protection, unattended positioning and limited resources complicate the incorporation of effective key management solutions [10]. We approach this problem from an application-based point of view: all the components

of a distributed application share a common *application key* that is used for communication among these components according to a symmetric-key encryption scheme [26].

We shall refer to the *single address space* paradigm of memory reference [4], [12], [19]. In this paradigm, the meaning of an address is unique in the whole system and is independent of the application that generates this address. In a distributed system, the main advantage of the single address space approach is simplicity in remote accesses. The components of a distributed application running on different nodes can refer a given information item using the address of this information item, which is unique system-wide.

More specifically, we divide the single address space into *partitions*, each partition being supported by the physical memory resources of a single sensor node. A component directly accesses the whole partition of its own node, for both read and write. The accesses to the partition of a different node occur on a segment basis. A *segment* is a contiguous memory area that is entirely contained within the boundaries of a single partition. The component in a given node can access the contents of a segment that is part of the partition of a different node, to read or modify these contents, by using a *handle* for that segment. A handle is a form of pointer that references a segment and is protected cryptographically [20], [21].

The rest of this paper is organized as follows. Section 2 illustrates our application model with special reference to partitions, segments and components. Section 3 analyses the concept of a segment handle, and introduces a set of system primitives, the *communication primitives*, which form the application interface of the distributed memory system. The problems connected with application key replacement are investigated in special depth. Section 4 discusses the proposed organization from a number of salient viewpoints including outdated key treatment, handle forging and stealing, storage requirements, and the network traffic generated by execution of the communication primitives. Section 5 gives concluding remarks.

2. THE APPLICATION MODEL

2.1. Partitions and segments

Let us consider a local network consisting of up to 2^d nodes connected by wireless links. The nodes share access to a single address space of size 2^t bytes. This virtual space is divided into 2^d partitions, and the i-th partition is associated with the i-th node (Figure 1). The size of a partition is 2^{t-d} bytes. An address in the virtual space consists of a d-bit node identifier and a ($t - d$)-bit offset in the partition of this node (Figure 2).

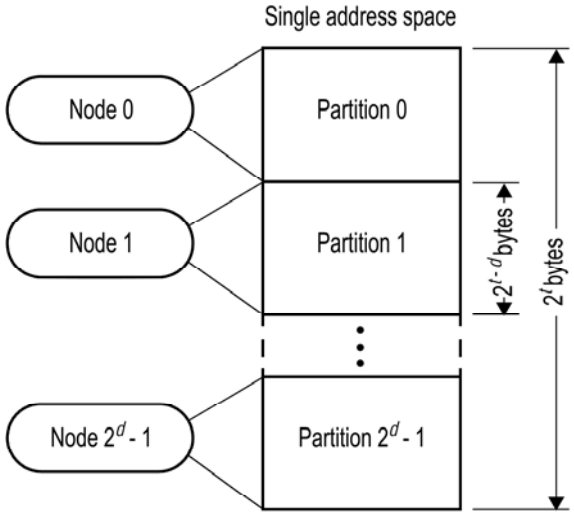

Figure 1. Configuration of the single address space featuring a partition for each node.

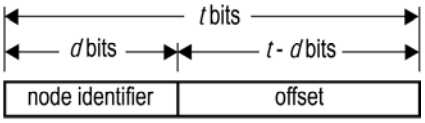

Figure 2. Configuration of an address in the single address space.

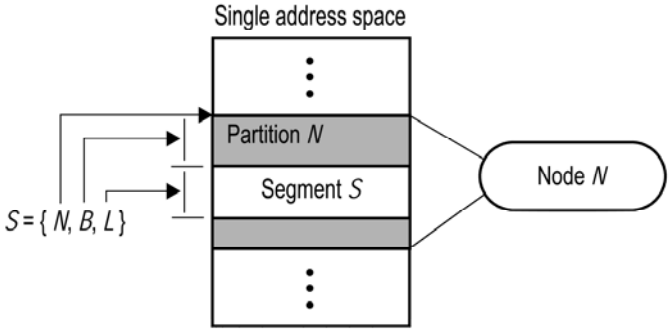

Figure 3. A segment *S* in the single address space. The segment is identified by the name *N* of the node, the segment base *B* and the segment length *L*.

.

A segment *S* is identified by triple $\{N, B, L\}$, where quantity *N* (*d* bits) specifies the node, and consequently the partition, of that segment; quantity *B* (*t* - *d* bits) specifies the segment *base*, i.e. the starting address of the segment in the partition; and quantity *L* expresses the segment *length* (Figure 3). Segments can overlap. This means that a memory address can be part of more than a single segment. As will be made clear shortly, segments are the basic units of data transmission between the nodes.

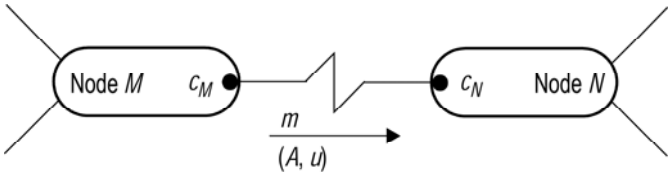

Figure 4. Sending a message m from node M to node N. c_M and c_N are components of application A. The control part of m includes application name A and the version u of the key used to encrypt the data part.

2.2. Components

As anticipated in Section 1, an application consists of a set of *components* distributed in the network nodes, and a given node may host a single component. At any given time, a cryptographic key, the *application key*, is associated with each application, and is used for communication among the components of this application. The application key may well be changed, as is required, for instance, when a component leaves the application, to prevent that component from taking advantage of the key any longer. Let k_0, k_1, … be the ordered sequence of the keys assigned to application A. The order number of a given key in the sequence is called the *version* of that key. Each given node holds an application key and the specification of the version of this key; the application key will be used by the component running in that node.

Components communicate via the exchange of messages. Let c_M and c_N be two components of application A being executed in nodes M and N, respectively. Suppose that c_M holds key k' of application A, and let u be the version of this key. Suppose also that c_M sends a message m to c_N (Figure 4). The message consists of a control part and a data part. The control part is in plaintext, the data part is encrypted by using a symmetric-key cipher and key k'. Besides the necessary routing information, the control part includes application name A and the version u of key k'. These information items will be used in the recipient node N to decrypt the data part. Throughout this paper we assume that the cipher is used in an authenticated encryption mode such as Counter with CBC MAC (CCM) mode, which has been designed to provide both confidentiality and authentication [9].[1]

In detail, suppose that the recipient component c_N holds key k'' of application A, and let v be the version of this key. When message m is received in node N, the version u of application key k' used to encrypt m is read in the control part of the message and is compared with the version v of k'':

[1] Intuitively, the same encryption key can be used for both confidentiality and authentication. At the sending side, the header and the payload are first authenticated. The resulting Message Identification Code (MIC) is then appended to the payload and the bundle is finally encrypted. At the receiving side, the ciphertext is decrypted into a payload and a MIC. Then, the MIC is verified against the received header and payload.

- If $u = v$ then the key held by the recipient component c_N matches the key that was used by the sender c_M to encrypt message m. c_N is in the position of using this key to decrypt the message.
- if $u < v$ then the key used by c_M to encrypt message m is outdated and the message should be discarded. A negative reply is sent to node M.
- if $u > v$ then c_N holds an outdated key that should be replaced; this issue will be considered in depth in the foregoing Section 3.3.

If the communication path from node M to node N includes other intermediate nodes, the control part of message m will be generally read by these intermediate nodes. This will be necessary for message routing, for instance. If an intermediate node includes a component of application A, this component will be in the position of decrypting the data part and accessing the message contents. This is not a protection violation, as we hypothesize that all the components of the same application are mutually trustworthy. On the other hand, an intermediate node that does not contain a component of application A will be precluded from accessing the data part, as it does not possess the key that was used to encrypt the message.

Our implementation of a distributed memory model is supposed to be layered on a routing service providing end-to-end connection between a sending component and a recipient component. A service of this type may be subject to many attacks (e.g. Sybil, blackhole, sinkhole, and HELLO flood [14]), which may endanger network integrity and availability, and possibily, confidentiality of the transported data. While countermeasures have been devised to implement forms of secure routing [14], the routing service is beyond the scope of this paper. Instead, our design concentrates on preventing possible attacks against the memory management layer from compromizing the integrity and the confidentiality of the single address space.

3. THE COMMUNICATION MODEL

3.1. Handles

Let $S = \{N, B, L\}$ be a segment, A be an application, and k be the key of this application. A handle H referencing S has the form $\{N, V\}$, where node name N is in plaintext, and quantity V is a *validation field* obtained by encrypting quadruple $\{N, B, L, E\}$ with a symmetric-key cipher and key k (Figure 5). Quantity E is a random number that is used as a "number used once" (*nonce*) [26]. As will be shown later, the nonce is used to prevent forms of replay attacks; it

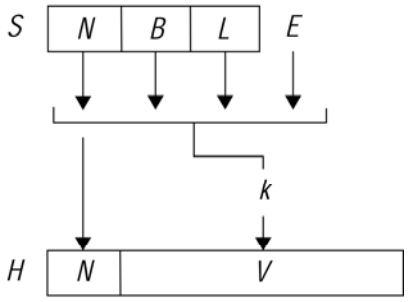

Figure 5. Generation of handle $H = \{N, V\}$ referencing segment $S = \{N, B, L\}$ with nonce E.

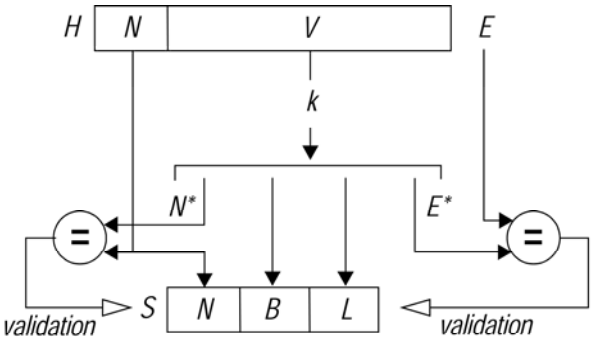

Figure 6. Validation of handle $H = \{N, V\}$ referencing segment $S = \{N, B, L\}$ with nonce E.

Table I. Communication primitives.

readSegment(*S*, *addr*)
Copies the contents of segment *S* from its present network position into a memory area at address *addr* of the partition of the current node.
writeSegment(*addr*, *S*)
Replaces the contents of segment *S* with quantities taken from a memory area at address *addr* of the partition of the current node.
replaceKey()
Reads a new application key and its version from the key segment reserved for the component running in the current node, and uses these quantities to update the application key in the current node.

allows us to distinguish a fresh request of segment access performed by using handle H from an illegitimate replay of a previous request performed by using the same handle.

Figure 6 shows the transformation of handle $H = \{N, V\}$ into plaintext, and the validation of the handle. Key k is used to decrypt validation field V and obtain quantities N^*, B, L and E^*. Quantity N^* is compared with partition name N and quantity E^* is compared with nonce E to validate H: if $N^* = N$ and $E^* = E$, H is valid and it references segment $S = \{N, B, L\}$.

3.2. Communication primitives

A set of three primitives, the communication primitives, forms the application interface of the distributed memory system (Table I). Execution of a communication primitive in a given node (the *current node*) implies interactions with a recipient node, and the components of both

nodes should be part of the same application. Let *A* be the application and *k* be the application key. Interactions consist of messages that are encrypted by using a symmetric-key cipher with key *k*.

In the rest of this section, we shall analyse the actions caused by execution of each communication primitive in detail. To simplify the presentation, we shall never mention the actions illustrated in Section 2.2, which are necessary when a message is received by a given node to validate the key used to encrypt the message against the key stored in that node. In the presentation, we shall take advantage of an informal notation to indicate the messages that are exchanged by the nodes involved in a communication. In this notation, $M \rightarrow N$: *string* is a message sent by node *M* to node *N*, and the string suggests a specific message implementation.

Accessing segments

Let $S = \{N, B, L\}$ be a segment in the partition of node *N*. A first example of a communication primitive is *readSegment*(*S*, *addr*). This primitive copies the contents of segment *S* from node *N* into a memory area of length *L* that starts at address *addr* of the partition of the invoking node, say node *M*. Let *A* be the application of the component issuing *readSegment*(), and let *k* be the key of this application. The actions caused by execution of this primitive are as follows (Figure 7):

1. Node *M* sends a message to node *N* asking for a random number that will be used as a nonce (message M1).
2. Node *N* generates a nonce *E* and sends it back to node *M* (message M2).
3. Node *M* generates a nonce *F*, and uses key *k* to assemble a handle $H = \{N, V\}$ referencing segment $S = \{N, B, L\}$ with nonce *E* (see Figure 1). Nonce *F* and handle *H* are sent to node *N* (message M3).
4. Node *N* uses key *k* to decrypt handle *H* into quadruple $\{N^*, B, L, E^*\}$ (see Figure 2). Then, quantity N^* is compared with node name *N*, and quantity E^* is compared with nonce *E* to validate *H*: if $N^* = N$ and $E^* = E$, *H* is valid and references segment $S = \{N, B, L\}$.
5. If handle validation fails, node *N* returns a negative reply to node *M* that raises an exception of violated protection and terminates execution of *readSegment*() unsuccessfully; otherwise
6. Node *N* assembles a message M4 including nonce *F* from message M3, the specification $\{N, B, L\}$ of segment *S*, and the contents of *S*. This message is encrypted by using application key *k* and is sent to node *M*.
7. Node *M* uses key *k* to decrypt message M4 into quantities F^*, N^*, B^*, L^*, and *contents*.

M1	$M \rightarrow N$:	*request*
M2	$N \rightarrow M$:	E
M3	$M \rightarrow N$:	$F, N, \{N, B, L, E\}_k$
M4	$N \rightarrow M$:	$\{F, N, B, L,$ *contents*$\}_k$

Figure 7. Messages exchanged in a successful execution of the *readSegment*() communication primitive.

Quantity F^* is compared with nonce F, and triple $\{N^*, B^*, L^*\}$ is compared with the specification $\{N, B, L\}$ of segment S; if matches are found, M4 is valid.

8. If validation of M4 fails, node M raises an exception of violated protection and terminates execution of *readSegment*() unsuccessfully; otherwise
9. Node M copies *contents* from message M4 into a memory area of length L starting at address *addr* of its own partition.

At points 3 and 6, in the transmission of messages M3 and M4, key k is only held by the components of A; it follows that a component of a different application, being executed, for instance, in an intermediate node in the path between node M and node N, will not be able to decrypt the messages, as it does not possess the key. At point 6, in message M4, nonce F is aimed at demonstrating freshness of this message, to avoid that an adversary can replay *contents*, and segment specification $\{N, B, L\}$ is compared with triple $\{N^*, B^*, L^*\}$ to allow the invoking node M to get certain that the returned contents correspond to the specified segment.

Let $S = \{N, B, L\}$ be a segment in the partition of node N. The *writeSegment*(*addr*, S) communication primitive copies the contents of a memory area of length L starting at address *addr* of the partition of the current node, say node M, into segment S. Let A be the application of the component issuing *writeSegment*(), and let k be the key of this application. Execution of this primitive is as follows (Figure 8):

1. Node M sends a message to node N asking for a random number that will be used as a nonce (message M1).
2. Node N generates a nonce E and sends it back to node M (message M2).
3. Node M generates a nonce F and a handle $H = \{N, V\}$ referencing segment $S = \{N, B, L\}$ with nonce E. Message M3 is assembled including nonce F, handle H, and the *contents* of a memory area of length L starting at address *addr* of the partition of M. This message is encrypted by using key k, and is sent to node N.
4. Node N uses key k to decrypt message M3 into handle $H = \{N^*, B, L, E^*\}$ and *contents* Then, quantity N^* is compared with node name N, and quantity E^* is compared with nonce E to validate H: if $N^* = N$ and $E^* = E$, H is valid and references segment $S = \{N, B, L\}$.

M1	$M \rightarrow N:$	*request*
M2	$N \rightarrow M:$	E
M3	$M \rightarrow N:$	$F, N, \{N, B, L, E, contents\}_k$
M4	$N \rightarrow M:$	$\{F, N, B, L, ACK\}_k$

Figure 8. Messages exchanged in a successful execution of the *writeSegment*() communication primitive.

5. If handle validation fails, node *N* returns a negative reply to node *M* that raises an exception of violated protection and terminates execution of *writeSegment*() unsuccessfully; otherwise
6. Node *N* replaces the contents of segment *S* with the new *contents* from message M3. Finally, node *N* returns a positive reply to node *M* in the form of a message M4 encrypted by using key *k* and containing nonce *F* and the specification $\{N, B, L\}$ of segment *S*.

At point 3, the encryption of message M3 indissolubly links the segment contents to the handle, and nonce *E* is aimed at proving that the message is not a replay. Similiarly, at point 6, in message M4, nonce *F* and segment specification $\{N, B, L\}$ prove that the reply is actually relevant to the current execution of the primitive and is not a replay.

3.3. Key replacement

In each application, a component, called the *application controller,* is responsible for the distribution of a new version of the application key to all the other components of that application. To this aim, the application controller associates a cryptographic key, the *base key*, and a segment, the *key segment*, with each given component. Key segments are all stored in the partition of the node where the application controller is running. Each component receives its own base key. It follows that while the components of a given application share the same application key, each component holds its own base key. Base keys are used to replace the application key as follows. The application controller generates a new key at random, and copies this new key and its version into the key segment of each component. Both these items are stored in ciphertext, and the encryption key is the base key of the given component. Afterwards, the controller sends a *key replacement message* to all the components. Consequently, each component assembles a handle referencing its own key segment, and uses this handle to ask the controller for the contents of this segment. On receipt of the reply from the controller, the component deciphers these contents by using its own base key, and uses the results to update the application key.[2]

[2] We wish to remark that we have a base key for each component (i.e. sensor node). Thus, different components of the same application use different base keys for communication with the application controller. In contrast, the

In more detail, let *ac* be the controller of application *A*, and let *C* be the node where *ac* is running. Let us refer to component c_M of application *A* being executed in node *M*. Furthermore, let bk_M be the base key of c_M (held by both the controller and c_M), and let $KS_M = \{C, B_M, L_M\}$ be the key segment that the controller has reserved for c_M in the partition of node *C*, where B_M and L_M are the base and the length of KS_M. Now suppose that *k'* is the current key of application *A*, and the version of *k'* is *u*. Suppose also this key should be replaced by a new key *k"* whose version is *v*, where $v > u$. To replace the key, the controller generates *k"* at random, and inserts this key and its version *v* into key segment KS_M; both these quantities will be encrypted by using base key bk_M. Then, the controller issues a key replacement message. On receipt of this message, component c_M executes communication primitive *replaceKey*() producing the actions that follow (Figure 9):

1. Node *M* sends a message to node *C* asking for a random number that will be used as a nonce (message M1).
2. Node *C* generates a nonce *E* and sends it back to node *M* (message M2).
3. Node *M* generates a nonce *F*, and uses base key bk_M to assemble a handle $H = \{C, V\}$ referencing segment $KS_M = \{C, B_M, L_M\}$ with nonce *E*. Nonce *F* and handle *H* are sent to node *C* (message M3).
4. Node *C* uses base key bk_M to decrypt handle *H* into quadruple $\{N^*, B_M, L_M, E^*\}$. Then, quantity N^* is compared with node name *C*, and quantity E^* is compared with nonce *E* to validate *H*: if $N^* = C$ and $E^* = E$, *H* is valid and it references key segment $KS_M = \{C, B_M, L_M\}$.
5. If handle validation fails, node *C* returns a negative reply to node *M* that raises an exception of violated protection and terminates execution of *replaceKey*() unsuccessfully; otherwise
6. Node *C* assembles message M4 including nonce *F* from message M3, the specification $\{C, B_{KS}, L_{KS}\}$ of key segment KS_M, the new application key *k"* and its version *v* (quantities *k"* and *v* are taken from KS_M). This message is encrypted by using base key bk_M and is sent to node *M*.
7. Node *M* uses base key bk_M to decrypt message M4 into quantities F^*, C^*, B^*, L^*, *k"*, and *v*. Quantity F^* is compared with nonce *F*, and triple $\{C^*, B^*, L^*\}$ is compared with the specification $\{C, B_M, L_M\}$ of K_{SM}. If matches are found, M4 is valid.
8. If validation of M4 fails, node *M* raises an exception of violated protection and terminates execution of *replaceKey*() unsuccessfully; otherwise

application key is shared by all the components of the same given application; this key make interactions possible between the components.

M1	$M \rightarrow C:$	*request*
M2	$C \rightarrow M:$	E
M3	$M \rightarrow C:$	$F, C, \{C, B_M, L_M, E\}_{bk_M}$
M4	$C \rightarrow M:$	$\{F, C, B_{KS}, L_{KS}, k", v\}_{bk_M}$

Figure 9. Messages exchanged in a successful execution of the *replaceKey*() communication primitive.

9. Node M uses the new key $k"$ and its version v to replace the current key and the corresponding version.

We wish to remark that each component executes the *replaceKey*() primitive as a consequence of receipt of a key replacement message from the application controller. The controller sends this message after inserting a new key (generated at random) and its new version (generated by incrementing the previous version) into every key segment; these actions are not part of *replaceKey*().

Base key bk_M is only held by application controller *ac* and component c_M, and is never transmitted across the network, so it cannot be captured. It follows that any other component will not be able to execute *replaceKey*() successfully to read the new key, as it does not possess the base key. This is true even for the other components of application *A*. In this way, we prevent a deviated component from taking advantage of an access to key segment KS_M in a form of an identity stealing.

4. DISCUSSION

4.1. Outdated keys

The mechanism for key replacement, introduced in Section 3.3, is able to deal with situations in which a component omitted to comply with one or more requests of key replacement. Let us consider application *A*, let *ac* be the controller of this application, and let *C* be the node where *ac* is running. Suppose that the key of application *A* has been changed from k' (version u) to $k"$ (version $v > u$), however component c_M of application *A* in node *M* has not updated the key. Suppose also that a new key replacement takes place, from $k"$ (version v) to k^* (version $w > v$). In a situation of this type, c_M is using the previous key k' instead of the more recent $k"$, and key segment KS_M reserved for c_M in node *C* contains the forthcoming key, k^*. When c_M issues *replaceKey*(), a handle *H* referencing KS_M will be sent to node *C* encrypted by using the base key bk_M of c_M. Execution of *replaceKey*() accesses KS_M in node *C* to read key k^*, and this key will be sent to node *M* encrypted by using bk_M. Thus, c_M will be in the position to decrypt k^* and update the key.

As seen in Section 2.2, if a component sends a message encrypted by using an outdated key, the message is discarded by the recipient component, which generates a negative reply. On receipt of the negative reply, the sender updates its own key and sends the message again. This means that components are able to recover from losses of key replacement messages. This feature is especially important for reliable group rekeying over an unreliable network [15], [16]. Furthermore, consider a system featuring a form of periodic rekeying [23], [24]. In a system of this type, the cryptographic keys are renewed at regular intervals to safeguard secrecy and maintain resilience to attacks and failures. In our system, if a component ignores one or more periodic key replacement messages, and then obeys a subsequent key replacement message, no negative effect follows on the communication ability of that component.

Suppose that component c_M of application A in node M executes *replaceKey*() twice. The second execution causes a new access to key segment KS_M that controller ac has reserved for that component; if the contents of the key segment have not been changed, the same application key is read again from the key segment, and *replaceKey*() has no other effect. In this respect, *replaceKey*() is idempotent; it can be executed multiple times without changing the result beyond the first execution.

When a component c_M leaves its own application A, it is necessary to change the application key to prevent that component from taking advantage of the old key any longer [5], [7]. The new key must be distributed to all the components of A except c_M. We shall obtain a result of this type as follows. The controller ac of application A will insert the new key in the key segments of all the components of A except key segment KS_M of c_M. Then, ac will send a key replacement message, thereby causing the components of A to execute *replaceKey*() and update the key. It should be noted that, if component c_M executes *replaceKey*(), this action produces no other effect, as key segment KS_M still contains the old, discarded key.

4.2. Handle forging and stealing

Let us suppose that component c_M of application A being executed in node M forges handle $H = \{N, V\}$ for a segment in the partition of node N that hosts a component of a different application A'. c_M will have to use an arbitrary value for validation field V, as it does not possess the application key of A'. Let us now suppose that c_M performs an attempt to use H, for instance, to read the contents of the corresponding, unknown segment. To this aim, c_M executes the *readSegment*() communication primitive. In the execution of this primitive, node N uses the application key of A' to decrypt the V field of handle H into quadruple $\{N^*, B, L, E^*\}$. Then, N validates H by verifying that $N^* = N$ and E^* is a fresh nonce. Of course, if we assume that the

cipher is in an authenticated encryption mode and that the size of the nonce is sufficiently large (e.g. 64 bits), the probability of casual matches is vanishingly low, and *readSegment*() is destined to terminate unsuccessfully.

Let us now consider the case that a component of a given application steals a handle from the legitimate owner, which is a component of a different application. In our system, an action of this type can be carried out at little effort, for instance, in the transmission of a handle between nodes: any intermediate node in the path from the sender node to the recipient node may well keep a copy of the handle. Let *A'* and *A"* be two applications, let *k'* and *k"* be the corresponding cryptographic keys, and let *c'* and *c"* be components of *A'* and *A"*, respectively. Suppose that *c'* sends handle $H = \{N, V\}$ referencing segment $S = \{N, B, L\}$ with nonce *E*, and *c"* steals a copy of this handle. In order to take advantage of *H* and read the contents of *S*, *c"* will issue communication primitive *readSegment*(). Execution of this primitive sends handle *H* to node *N*. However, node *N* is part of application *A'* (it hosts a segment of this application). Consequently, it will return the contents of segment *S* encrypted by using application key *k'*, and component *c"* will not be able to decrypt these contents.

Let us now assume that a component c_M of application *A* is captured. This means that both the application key and the base key of c_M are compromised. As soon as the intrusion is detected, the controller ac_A of application *A* generates a new application key and inserts this key into the key segments of all components except c_M. The controller issues a key replacement message causing all genuine components to execute the *replaceKey*() primitive to get the new key; the compromised component c_M keeps the old key and consequently is logically evicted from the system.

4.3. Considerations concerning performance

Storage costs

In sensor nodes, memory is a scarce resource. Related issues are the key distribution scheme and the memory requirements for key storage. If a single master key is shared by all nodes, the memory requirements are kept to a minimum [27]. In this approach, we have a form of perfect key connectivity, but a node that discloses the master key compromises the whole network; revocation of the master key is hard if not impossible, owing to the need to rekey all remaining nodes without using the compromised key.

An alternative approach is the *full pairwise scheme*, whereby each node receives a cryptographic key for each other node [25]. This means that, in a network consisting of *n* nodes, each node stores *n* - 1 keys (and many of them will never be used). The resulting high memory

cost makes this approach only suitable for small networks featuring a predictably low number of nodes.

In the wide class of the probabilistic key sharing schemes, each node receives a number of keys that is much smaller than the total number of nodes that form the network [3]. In the so-called *basic scheme* [11], a large set of keys K is initialized with random keys and their identifiers. Each node is loaded with k keys, which are chosen at random from K. Two adjacent nodes (connected by a direct network link) are in the position of communicating if they share at least one key. The probability that this is indeed true is a function of both the cardinality of K and quantity k, e.g. if $k = 75$ and K contains 10,000 keys the probability is 0.5 [11]. A node will be disconnected from the network if it has no key in common with every adjacent node; for adequate levels of network density, the probability that a node be actually disconnected is negligible.

In our approach, the given node stores the key of the application of the component being executed in that node. Further memory costs are connected with storage of the base keys and the key segments (see Section 3.3). The controller of a given application stores a base key and a key segment for each component of that application. This means that more memory space is required in the controller of a complex application featuring a large number of components distributed across the network. Even in a situation of this type, for each component that is not an application controller the memory cost is equal to a single base key, and is negligible. We may conclude that the total memory requirements are independent of the network size, and are much lower than those necessary to guarantee a suitable degree of connectivity in a probabilistic key sharing scheme.

Network traffic

Execution of the *readSegment*() communication primitive causes the transmission of four messages across the network (two messages for the request and delivery of the first nonce, one message to send the second nonce and the handle for the segment being accessed, and one message to transmit the segment contents). This is similar to the communication cost of the *writeSegment*() primitive. Thus, for these primitives the network cost is kept to a minimum.

As far as key replacement is concerned, one message is necessary from the application controller to each controlled component to trigger the key replacement activity. Each component will then issue the *replaceKey*() primitive. The cost of this primitive in terms of memory traffic is four messages (two messages for the first nonce, one message to send the second nonce and the handle for the key segment from the controlled component to the

application controller, and one message for transmission of the new key from the application controller back to the component). Thus, the total number of messages generated by a key replacement activity is a function of the complexity of the given application in terms of the number of its components, and is independent of the network size.

5. CONCLUDING REMARKS

With reference to a distributed architecture consisting of sensor nodes connected by wireless links in an arbitrary network topology, we have considered a single address space paradigm of memory reference. In a segment-oriented, distributed implementation of this paradigm, a salient problem is the mapping of segment names into physical addresses to identify the network node that gives physical support to the given segment. In our solution, the address space is divided into partitions. Each partition is physically supported by the memory resources of a single sensor node. An application component accesses a given segment by presenting a handle for this segment, which includes the name of the corresponding node. The following is a brief summary of the main results we have obtained:

- Handles are protected cryptographically. The meaning of a handle is confined within the boundaries of the application that created this handle, and this nullifies any action of handle stealing. Any attempt to forge a handle from scratch and use this handle for memory access is destined to fail if this handle references a segment in a node of a different application.
- The replacement of the key of a given application is initiated by the application controller that sends a key replacement message to all the application components. Consequently, each component executes the *replaceKey*() communication primitive and updates its own key. This key replacement mechanism results to possess a number of interesting properties. If a key replacement message is lost, the key replacement activity is initiated by the first message that is received encrypted with the new key. If a key replacement message is obeyed twice, e.g. as a consequence of a transmission error leading to repeated message delivery, the second key replacement activity produces no effect. If a component ignores one or more key replacement messages and then obeys a subsequent message, no negative consequence follows on the communication ability of that component. These features are especially important for reliable group rekeying over an unreliable network.
- The memory requirements of the activities of application key replacement are a function the number of the components that form the application whose key is replaced, and are independent of the network size.

- The number of messages generated by the execution of the communication primitives is independent of the network size.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their insightful comments and constructive suggestions.

This work has been partially supported by the TENACE PRIN Project (Grant no. 20103P34XC_008) funded by the Italian Ministry of Education, University and Research, and by the PLANET Integrated Project (Grant no. FP7-2009-5-257649) funded by the European Commission under the 7th Framework Programme.

REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, "A survey on sensor networks," *IEEE Communications Magazine*, vol. 40, no. 8 (August 2002), pp. 102–114.

[2] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, C. Yoon, "RETOS: resilient, expandable, and threaded operating system for wireless sensor networks," *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, Cambridge, Massachusetts, USA, April 2007, pp. 148–157.

[3] H. Chan, A. Perrig, D. Song, "Random key predistribution schemes for sensor networks," *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2003, pp. 197–213.

[4] J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 4 (November 1994), pp. 271–307.

[5] O. Cheikhrouhou, A. Koubâa, G. Dini, M. Abid, "RiSeG: a ring based secure group communication protocol for resource-constrained wireless sensor networks," *Personal and Ubiquitous Computing*, vol. 15, no. 8 (December 2011), pp. 783–797.

[6] L. H. A. Correia, D. F. Macedo, A. L. dos Santos, A. A. F. Loureiro, J. M. S. Nogueira, "Transmission power control techniques for wireless sensor networks," *Computer Networks*, vol. 51, no. 17 (December 2007), pp. 4765–4779.

[7] G. Dini, I. M. Savino, "LARK: a lightweight authenticated rekeying scheme for clustered wireless sensor networks," *ACM Transactions on Embedded Computing Systems*, vol. 10, no. 4 (November 2011).

[8] A. Dunkels, B. Grönvall, T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004, pp. 455–462.

[9] M. J. Dworkin, *Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality*, Technical Report, National Institute of Standards and Technology, Special Publication 800-38C, May 2004, Gaithersburg, MD, USA.

[10] M. Eltoweissy, M. Moharrum, R. Mukkamala, “Dynamic key management in sensor networks,” *IEEE Communications Magazine*, vol. 44, no. 4 (April 2006), pp. 122–130.

[11] L. Eschenauer, V. D. Gligor, “A key-management scheme for distributed sensor networks,” *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, USA, November 2002, pp. 41–47.

[12] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, J. Liedtke, “The Mungi single-address-space operating system,” *Software — Practice and Experience*, vol. 28, no. 9 (July 1998), pp. 901–928.

[13] F. Hu, N. K. Sharma, “Security considerations in ad hoc sensor networks,” *Ad Hoc Networks*, vol. 3, no. 1 (January 2005), pp. 69–89.

[14] C. Karlof, D. Wagner, “Secure routing in wireless sensor networks: attacks and countermeasures,” *Ad Hoc Networks*, vol. 1, no. 2–3 (September 2003), pp. 293–315.

[15] F. Kausar, S. Hussain, J. H. Park, A. Masood, “Secure group communication with self-healing and rekeying in wireless sensor networks,” *Proceedings of the 3rd International Conference on Mobile Ad-Hoc and Sensor Networks*, Beijing, China, December 2007, pp. 737–748.

[16] H. Kurnio, R. Safavi-Naini, H. Wang, “A secure re-keying scheme with key recovery property,” *Information Security and Privacy*; in *Lecture Notes in Computer Science*, vol. 2384/2002, pp. 1–51.

[17] P. Levis *et al.*, “TinyOS: an operating system for wireless sensor networks,” in: W. Weber, J. Rabaey and E. H. L. Aarts, editors, *Ambient Intelligence*. New York: Springer-Verlag, 2005, pp. 115–148.

[18] Y. Liang, W. Peng, “Minimizing energy consumptions in wireless sensor networks via two-modal transmission,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1 (January 2010), pp. 12–18.

[19] L. Lopriore, “Access control mechanisms in a distributed, persistent memory system,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 10 (October 2002), pp. 1066–1083.

[20] L. Lopriore, “Encrypted pointers in protection system design,” *The Computer Journal*, vol. 55, no. 4 (April 2012), pp. 497–507.

[21] L. Lopriore, “Object protection in distributed systems,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 5 (May 2013), pp. 570–579.

[22] A. Perrig, J. Stankovic, D. Wagner, “Security in wireless sensor networks,” *Communications of the ACM*, vol. 47, no. 6 (June 2004), pp. 53–57.

[23] T. Pham, P. A. Watters, “The efficiency of periodic rekeying in dynamic group key management,” *Proceedings of the Fourth European Conference on Universal Multiservice Networks*, Toulouse, France, February 2007, pp. 425–432.

[24] S. Rafaeli, D. Hutchison, “A survey of key management for secure group communication,” *ACM Computing Surveys*, vol. 35, no. 3 (September 2003), pp. 309–329.

[25] M. A. Simplício, P. S. L. M. Barreto, C. B. Margi, T. C. M. B. Carvalho, “A survey on key management mechanisms for distributed wireless sensor networks,” *Computer Networks*, vol. 54, no. 15 (October 2010), pp. 2591–2612.

[26] M. Stamp, *Information Security: Principles and Practice*, 2nd Edition. Hoboken, New Jersey: Wiley, 2011.

[27] Y. Xiao *et al.*, "A survey of key management schemes in wireless sensor networks," *Computer Communications*, vol. 30, no. 11–12 (September 2007), pp. 2314–2341.

[28] J. Zhang, V. Varadharajan, "Wireless sensor network key management survey and taxonomy," *Journal of Network and Computer Applications*, vol. 33, no. 2 (March 2010), pp. 63–75.