A

SpartanRPC: Remote Procedure Call Authorization in Wireless Sensor Networks

PETER CHAPIN and CHRISTIAN SKALKA, University of Vermont

We describe SpartanRPC, a secure middleware technology that supports cooperation between distinct security domains in wireless sensor networks. SpartanRPC extends nesC to provide a link-layer remote procedure call (RPC) mechanism, along with an enhancement of configuration wirings that allow specification of remote, dynamic endpoints. RPC invocation is secured via an authorization logic that enables servers to specify access policies, and requires clients to prove authorization. This mechanism is implemented using a combination of symmetric and public key cryptography. We report on benchmark testing of a prototype implementation, and on an application of the framework that supports secure collaborative use and administration of an existing WSN data gathering system.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Run-time environments*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server*

General Terms: Security, Languages

Additional Key Words and Phrases: remote procedure call, sensor networks, trust management

ACM Reference Format:
Peter Chapin and Christian Skalka. 2014. SpartanRPC: Remote Procedure Call Authorization in Wireless Sensor Networks *ACM Trans. Info. Syst. Sec.* V, N, Article A (January YYYY), 30 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

SpartanRPC is an extension of the nesC programming language [Gay et al. 2003] that supports application development in a decentralized, open world security model for wireless sensor networks (WSNs). Traditional networks have long enjoyed support for an open world security model via public key based security architectures such as the secure sockets layer (SSL). The goal of our work is to introduce an open world security model to the TinyOS programming environment for embedded device programming.

Currently, TinyOS security models are very simple and support only a closed world paradigm. TinySec [Karlof et al. 2004] and MiniSec [Luk et al. 2007] are based on shared secrets and generally assume that an entire network comprises a single security domain. Furthermore, these systems support confidentiality and integrity properties, but not access control, a.k.a. authorization. In contrast SpartanRPC includes primitive features for specifying and enforcing authorization policies and allows multiple security domains to interact within a single network. SpartanRPC security mech-

Christian Skalkas work was supported by a grant from the Air Force Office of Scientific Research Young Investigator Program (AFOSR YIP).
Author's addresses: P. Chapin, (Current address) Computer Information Systems Department, Vermont Technical College; C. Skalka, Computer Science Department, University of Vermont.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1094-9224/YYYY/01-ARTA $15.00
DOI:http://dx.doi.org/10.1145/0000000.0000000

anisms leverage public key cryptography and an authorization logic to support an open world model where shared secrets are not required a priori.

1.1. Overview and Applications

The SpartanRPC system provides an applications programming interface for managing resource access control in a WSN. It allows network administrators to define security policies that mediate access to specified resources on network nodes, and allows subnetworks from different security domains to interact. A *resource* in SpartanRPC is user-defined functionality programmed in an extension of nesC, and accessible via RPC by client code programmed in the same extension of nesC. Thus, while previous systems have explored the problem of establishing multiple security domains in a WSN [Claycomb and Shin 2011], and other have considered RPC in WSNs [May et al. 2007], SpartanRPC combines these features for secure WSN application development in TinyOS. Furthermore, SpartanRPC's expressive authorization logic allows specification of fine-grained and decentralized security policies, better supporting collaborations between multiple security domains.

Since the SpartanRPC API is flexible and easily accessible to TinyOS programmers, these features can be readily used in a variety of application spaces. Consider a first-responder situation, in which multiple social entities must interact and cooperate on an ad-hoc basis. Recent work has shown the effectiveness of WSNs in such scenarios [Gao et al. 2008; Lorincz et al. 2004], in their ability to coordinate multiple data collection and communication devices in an easily deployable manner. However, data collection and communication in this scenario (and other similar ones) must be a secured resource, due to e.g. requirements of the Health Insurance Portability and Accountability Act. Furthermore, security must be coordinated on-site in a WSN comprising subnetworks administered separately (police, medical units from different hospitals, etc.), and no prior coordination between domains can generally be assumed. The SpartanRPC system is designed to address these types of scenarios.

For example, if an emergency medical technician (EMT) emplaces a WSN to monitor patient locations and vital signs, a security policy can be imposed whereby responding police departments can emplace their own WSN, and through it access patient identity and location data but *not* medical data directly from the EMT's network. This direct data access will often be necessary due to real-time constraints and lack of Internet connectivity in emergency situations.

Time synchronization is another important WSN function that is security sensitive, since many higher-level protocols rely on it. A number of previous authors have considered secure time synchronization in the presence of "insider" attacks [Manzo et al. 2005; Ganeriwal et al. 2008], whereby nodes within the network may be compromised and function as malicious actors capable of corrupting the protocol. In particular, the FTSP protocol can be attacked by a single compromised "root" node injecting false timing information into the network [Manzo et al. 2005], even when symmetric keys are used for secure information exchange. However, the threat model in this work treats all nodes in a network as equally compromisable. In cases where a connected subnetwork is more resistant to compromise, due to e.g. differences in hardware, a SpartanRPC policy can be established whereby only nodes in the most tamper-resistant sub-component of the network may function as roots. FTSP time sync updates on any given node can be defined to require a root authorization level. This implies that nodes requiring secure time synchronization must be at most a single radio hop from a root node, but nodes willing to accept possibly corrupted time sync data can extend the network indefinitely. Note that in this scenario, SpartanRPC policies adapt to heterogeneity in network device hardware, vs. network administration as in the previous example.

Other potential applications of our system include secure routing protocols in heterogeneous trust environments [Karlof and Wagner 2003], transport and network layer protocols [Perillo and Heinzelman 2005], tracking protocols [Brooks et al. 2003], and even mote-based web servers supporting secure channels [Gupta et al. 2005].

As a concrete application example, in Sec. 8 we describe our implementation of a WSN application for environmental data sampling and collection. The node software provides remote resources for data collection or node control. Access to these resources, in particular via a distinguished sink node that is under control of some user, is mediated by a security policy. Depending on the user's credentials, only data collection, or collection and control, or neither, will be allowed. Thus, we can specify that data end users can be provided with sink nodes that only allow collection, whereas system administrators can also control the network from theirs. And since user populations may be part of different social entities than system administration, our policy language allows authorization of a social entity itself to administer its own user base for data collection. This supports collaboration between different institutions which is a hallmark of this particular application.

1.2. Security Goals

The main security-related goal of our work is to provide fine-grained *authorization* a.k.a. *access control* in a WSN. We support an *open world* setting, where principals do not share secrets a priori, and employ an authorization logic capable of expressing rich security policies [Li et al. 2002]. As in various authorization frameworks for more traditional distributed systems [Ellison et al. 1999], principals are identified as asymmetric key pairs, so that possession of a private key is necessary and sufficient to establish a principal's identity. With this understanding of identity, our underlying security protocols support authentication of resource users through the use of a Diffie-Hellman protocol. We assume that WSN nodes are provided with private keys a priori by a trusted certification authority. Thus, we are not concerned with communication of private keys.

Because WSN nodes are assumed to possess private keys, our system is vulnerable to hardware tampering, which we consider out-of-scope for this work. Modulo this, our system ensures that only authenticated principals capable of proving compliance with a specified authorization policy may access a protected resource. Authorization is proved by communication of credentials to the authorizer in the form of certificates, as discussed in Sec. 4. We note that the onus of credential communication is on the requester, not the authorizer, who only passively receives credentials. This implies that access may not be granted in case incomplete credential sets are provided. Although this incompleteness has been addressed in traditional distributed systems via e.g. distributed credential chain discovery [Chapin et al. 2008], these approaches are relatively heavyweight and difficult to implement in a WSN. The validity and integrity of the certificates themselves are established via signature verification by the authorizer.

In order to authenticate requesters, and to improve the efficiency of normal message transfers between a requester and an authorizer, symmetric session keys are computed using a simple Diffie-Hellman key agreement protocol [Diffie and Hellman 2006] as described in Sec. 6.1.2. The version of Diffie-Hellman we use is not susceptible to man-in-the-middle attacks because of the way principles are directly represented by their keys. Thanks to this, and our certificate representation of credentials, the SpartanRPC authorization scheme is not susceptible to man-in-the-middle attacks, where an unauthorized principal (i.e. a device not holding the private key denoting an authorized principal) can obtain access to resource. This claim is supported by relevant technical discussion in Sec. 6. We consider other security properties of our system, relevant to

DoS, replay, and confidentiality, in Sec. 6.1.4 following a more detailed discussion of the language model implementation.

1.3. Technical Foundations

Technical foundations underlying our secure RPC design include programming language-based techniques, asynchronous RPC semantics, and public key (PK) based authorization logics.

Language-Based Security. SpartanRPC provides language-level abstractions for defining remote services and associated security policies. Programmers are presented with an extension of nesC, with new features for defining remote access controlled services, and for invoking those services at specific authorization levels. This hides the implementation details of underlying security protocols and only requires mastery of a simple authorization logic. SpartanRPC programs are compiled in the same manner as nesC programs, in fact the SpartanRPC compiler rewrites SpartanRPC programs to nesC code and compiles the latter.

Asynchronous Remote Procedure Call. As other authors have observed [May et al. 2007], RPC is an appropriate abstraction for node services on the network and supports whole-network (vs. node-specific) programming. Secure RPC is well-studied in a traditional networking environment, and is a natural means of layering security over a distributed communication abstraction.

It is necessary for RPC invocation in a WSN to be asynchronous, since synchronous call-and-return to a remote node would significantly impede performance in the best case and cause deadlock in the worst. In order to minimally impact the nesC programming model, we define RPC invocation as a form of *remote task*. Local tasks are units of programmer-defined asynchronous computation in nesC, so treating remote computational services as remote tasks fits well in this paradigm. Remote tasks can be invoked on one-hop neighbors, providing a link layer service on which network layer services can be built. For example in Sec. 5 we illustrate how a secure multi-hop data collection protocol can be built using our link layer service.

PK-Based Authorization Policies. SpartanRPC provides language-level abstractions for specifying RPC authorization policies. The policy language we support is RT [Li et al. 2002], which allows network entities to communicate composable credentials for authorizing service invocations. Credentials are typically signed by certificate authorities and do not require shared secrets to validate. SpartanRPC certificates use elliptic curve cryptography (ECC) [Bertoni et al. 2006] signatures which are validated during an initial authorization phase. ECC is significantly more tractable than RSA in a WSN setting. Following the initial authorization phase our protocol establishes a shared AES key for computing message authentication codes (MACs) during subsequent invocations of a given service by the same node. Since hardware AES is available on common WSN radio chipsets, we obtain highly efficient performance for secure invocations following authorization. This is demonstrated with empirical results reported in Sec. 7.

1.4. Outline of Paper

In Sec. 2 we describe the fundamental language abstraction we provide for defining remote services, called duties. In Sec. 3 we describe a modification of remote services that accommodates dynamically changing communication neighborhoods. In Sec. 4 we define our authorization logic, and show how to specify duty posting policies and how they are enforced in the implementation. In Sec. 5 we provide the extended example of secure directed diffusion. In Sec. 6 we summarize novel and important features of

our implementation. In Sec. 7 we discuss empirical results of system performance. In Sec. 8 we describe a realistic application of our system. We conclude with a discussion of related work in Sec. 10.

2. DUTIES AND REMOTABILITY

The wireless communication in a WSN is typically unreliable due to frequent node failures and interference effects. Furthermore the data transfer rate in a WSN based on the IEEE 802.15.4 link layer protocol is much less than in other common networked environments. For example the CC2420 radio used in our experiments, a typical WSN device, supports a data rate of only 250 Kbps [Chipcon 2004]. Consequently we believe it is unrealistic for RPC services in WSNs to be synchronous. Instead we believe the semantics of nesC tasks are a more appropriate abstraction. They are not quite right however, since RPC services, unlike nesC tasks, will typically require arguments to be passed. Also a nesC task can only be posted in the module where it is defined. In contrast an RPC service invokes remotely defined functionality. We therefore define a new RPC abstraction called a *duty*.

2.1. Syntax and Semantics

Duties are declared in interfaces and syntactically resemble command declarations. Instead of using the reserved word `command` the new reserved word `duty` is used. Duties are allowed to take parameters, with restrictions as discussed below, but must return the type `void`. For example the following interface describes an RPC service for remotely controlling a collection of LEDs:

```
interface LEDControl { duty void setLeds(uint8_t ctl); }
```

Duties are defined in modules in a manner similar to the way tasks, commands, or events are defined. The reserved word `duty` is again used on the definition. Like commands and events the name of the duty is qualified by the name of the interface in which it is declared. Including a duty in an interface definition automatically implies that the interface can be remotely invoked, or is *remotable* in the sense formalized in Sec. 2.2. Any remotable interface provided by a component must be specified as `remote` in the component's list of provided interfaces. The first code sample in Fig. 1 shows an `LEDControllerC` component that provides the `LEDControl` interface remotely– i.e. that allows remote nodes to control LED status lights on a board. A more extended example of duty implementation and usage is provided in Sec. 5.

A module on the client node that wishes to use a remote interface simply posts the duty in the same manner as tasks are posted. The use of `post` emphasizes the asynchronous nature of the invocation. An example duty posting is illustrated in Fig. 1. The standard component semantics of nesC provide a natural abstraction of "where" the RPC call goes, just as e.g. a normal command invocation will go through a component interface that is disconnected from its implementation. Like a normal command invocation, configuration wirings determine where duty control flows. However, in SpartanRPC duty invocation control flows to a component residing on a different network node. The invoking module must be connected to the remote modules by way of a dynamic wire as described in Sec. 3.

When a duty is posted by a client it may run at some time in the future on the server node. The client node continues at once without waiting for the duty to start. Once posted the client has no direct way to determine the status of the duty. Also, due to the unreliability of the network a posted duty may not run at all. SpartanRPC does not attempt to retransmit or even detect failed duty invocations; postings occur at most once. Any error semantics for duty postings must be implemented by the application developer.

```
module LEDControllerC { provides remote interface LEDControl; }
implementation {
  duty void LEDControl.setLeds(uint8_t ctl) { ... }
}

module LoggerC { uses interface LEDControl; }
implementation {
  void f() { ... post LEDControl.setLeds(42); }
}
```

Fig. 1. Duty Implementation and Invocation Examples

2.2. Remotable Interfaces

We impose certain requirements on RPC service definitions for ease of implementation. First, since WSN nodes do not share state we disallow passing references to duties—such a reference would be meaningless on the receiving node. Thus we define remotable types:

Definition 2.1. A type is *remotable* if and only if it satisfies the following inductive definition: The nesC built-in arithmetic types, including enumeration types, are remotable, and structures containing remotable types are remotable.

Since a remotable interface describes RPC services, we require that they specify duties taking only arguments of remotable type; also, remotable interfaces can only contain duties, to ensure meaningful remote usage.

Definition 2.2. An interface is *remotable* if and only if it only provides duties whose argument types are remotable.

3. DYNAMIC WIRES

In an ordinary nesC program the "wiring" between components as defined by configurations is entirely static. The nesC compiler arranges for all connections, and at run time the code invoked by each called command or signaled event is predetermined.

In a remote procedure call system for wireless networks this static arrangement is insufficient. A node can not, in general, know its neighbors at compilation time but rather must discover this information after deployment. In addition, the volatility of wireless links and of the nodes themselves means that a given node's set of neighbors will change over time. In this section we discuss the facility in SpartanRPC to allow *dynamic wirings* for specifying control flow from duty invocation to duty implementation.

3.1. Component IDs, Component Managers

We begin by discussing how remote components are identified for wiring. In order to uniquely identify components on the network, remotable components are specified via a two-element structure called a `component_id` defined on the left side of Fig. 2. The `node_id` member is the same node ID used by TinyOS and is set when the node is programmed during deployment. The local ID member is an arbitrary value defined by the programmer of the server node. Only components that are visible remotely need to have ID values assigned, however, the ID values must be unique *on the node*. The `component_set` structure defined on the right side of Fig. 2 wraps an arbitrary array of `component_id` values.

A *component manager* is a component that provides the `ComponentManager` interface defined at the bottom of Fig. 2. It dynamically specifies a set of component IDs that

```
typedef struct {                typedef struct {
    uint16_t node_id;               int count;
    uint8_t  local_id;              component_id *ids;
} component_id;                 } component_set;

interface ComponentManager { command component_set elements(); }
```

Fig. 2. Component Manager Interface and Type Definitions

```
module RemoteSelectorC { provides interface ComponentManager; }
implementation {
    component_id  broadcast  = { 0xFFFF, 1 };
    component_set broadcast_set = { 1, &broadcast };

    command component_set ComponentManager.elements() {
        return broadcast_set;
    }
}
```

Fig. 3. Example Component Manager

ultimately serve as dynamic wiring endpoints. An example component manager is discussed in detail Sec. 5.

As a simple example, consider the component manager `RemoteSelectorC` as shown in Fig. 3. This component manager always returns a component set containing a single component. The special SpartanRPC broadcast node ID is used (`0xFFFF`) indicating that all neighbors should be the target of the dynamic wire. The component ID on the neighbors is specified as `1` in this example. In a more complex example the component manager would compute the component set each time the dynamic wire is used, filling in an array of component IDs based on information gathered earlier.

3.2. Syntax and Semantics

In SpartanRPC we extend the syntax and semantics of nesC to allow the target of a connection to be dynamically specified by a component manager. The syntax of wirings, or connections, is extended as follows:

```
      connection ::= endpoint '->' dynamic_endpoint
dynamic_endpoint ::= '[' IDENTIFIER ']' ('.' IDENTIFIER)?
```

Given a dynamic wiring of the form `C.I -> [M].I`, we informally summarize its semantics as follows. First, we statically require that `M` be a component providing the `ComponentManager` interface, and that `I` be a remotable interface. At run time, if control flows across this wire via posting of some duty `I.d` within component `C`, the command `elements` in `M` is called to obtain a set of component IDs. The duties `I.d` provided by the specified remote components will then be posted on their respective nodes via an underlying link layer communication, the details of which are hidden from the programmer. Thus, duties can only be posted on neighbors. Note that since this call to `elements` may return more than one component ID, this is a sort of fan-out wiring.

For example, the programmer could wire the `LoggerC` component mentioned in Fig. 1 to LED controller components on a dynamically changing subset of neighbors using a configuration such as:

```
LoggerC.LEDControl -> [RemoteSelectorC];
```

The server's configuration does not need to wire anything to the remote interface explicitly.

3.3. Callbacks and First-Class IDs

We assume that the component IDs for well known services will be agreed upon ahead of time by a social process outside of our system. By broadcasting to a well known component ID, a node can use services on neighboring nodes without knowing their node IDs.

If a node expects a reply from a service that it invokes, the invoking node must set up a component with a suitable remote interface to receive the service's result. In SpartanRPC remote invocations can only transmit information in one direction. Bidirectional data flow requires separate dynamic wires. This design provides a natural "split-phase" semantic in which the invoker of a service can continue executing while waiting for the result of that service. For example, a service might require the client to provide the node ID and component ID of the component that will receive the service result as arguments to the service invocation. The server could store those values for use by a server-side component manager. It is permitted for a component to be its own component manager making it easy for a service to return a result by posting the appropriate duty.

4. SECURITY POLICY SPECIFICATION AND PROGRAM LOGIC

In this section we discuss how to extend the language setting described previously with security features. The goal is a language framework where RPC services require authorization for use, and where authorization policies support collaboration between multiple security domains. To this end we adapt a distributed trust management system [Chapin et al. 2008] for policy specification. This system secures WSN application programming by way of the SpartanRPC API.

4.1. Security Policy Language

Authorization in trust management systems is more expressive than in traditional access control schemes such as access control lists or role based access control (RBAC) [Sandhu et al. 1996]. In these simpler models, access is based on identities of principals. But in the distributed scenarios we are considering here, creating a single local database of all potential requesters is untenable. Where there are multiple domains of administrative control, no single authorizer can have direct knowledge of all users of the system. Furthermore, in highly dynamic and volatile environments, no single entity in-network can be expected to keep pace with changes in an authoritative manner. Finally, basing authorization purely on identity is not a sufficiently expressive or flexible approach, since security in modern distributed systems utilizes more sophisticated features (e.g. delegation). These problems are addressed by the use of trust management systems such as the RT framework [Li et al. 2002]. We use the system RT_0 in this foundational presentation due to its simplicity, but other RT variants [Li et al. 2003; Li and Mitchell 2003b] could be adapted.

Like other trust management systems such as SPKI/SDSI [Ellison et al. 1999], RT represents principals as public keys and does not attempt to formalize the connection between a key and an individual. The RT literature usually refers to principals as *entities*. RT allows each entity to define *roles* in a name space that is local to that entity. An authorizer associates permissions with a particular role; to access a resource a requester must prove membership in the role. In this way RT provides a form of role based access control.

To define a role, an entity issues credentials that specify the role's membership. Some of these credentials may be a part of private policy, others may be signed by the issuer and made publicly available as certificates. The overall membership of a role is taken as the union of the memberships specified by all known defining credentials.

Let $A, B, C, \ldots$ range over entities and let $r, s, t, \ldots$ range over role names. A role r local to an entity A is denoted by $A.r$. RT credentials are of the form $A.r \leftarrow f$, where f can take on one of four forms to obtain one of four credential types:

(1) $A.r \leftarrow E$. This form asserts that entity E is a member of role $A.r$.

(2) $A.r \leftarrow B.s$. This form asserts that all members of role $B.s$ are members of role $A.r$. Credentials of this form can be used to delegate authority over the membership of a role to another entity.

(3) $A.r \leftarrow B.s.t$. This form asserts that for each member E of $B.s$, all members of role $E.t$ are members of role $A.r$. Credentials of this form can be used to delegate authority over the membership of a role to all entities that have the attribute represented by $B.s$. The expression $B.s.t$ is called a *linked role*.

(4) $A.r \leftarrow q_1 \cap \cdots \cap q_n$. Where the q_i are qualified role names such as $B.s$. This form asserts that each entity that is a member of all roles $q_1, \ldots, q_n$ is also a member of role $A.r$. The expression $q_1 \cap \cdots \cap q_n$ is called an *intersection role*. In our implementation only two constituent roles q_1 and q_2 are allowed in an intersection role. This does not limit expressivity since intermediate roles can be introduced as necessary to handle larger intersections.

For all credential forms $A.r \leftarrow f$, the principal A is called the *issuer* of the credential. An example credential set is presented and discussed in Sec. 8.

The formal semantics of RT can be expressed in terms of Datalog [Li et al. 2002]. The translation of RT credentials to Datalog requires only a single relation *isMember* to assert when a particular entity is a member of a particular role. A type (1) credential, called a *membership credential*, is translated into Datalog simply as a fact. For example the credential $A.r \leftarrow E$ becomes the fact *isMember*(E, A, r). The other three credential types are translated into Datalog rules. For example, the type (3) credential $A.r \leftarrow B.s.t$ becomes the following Datalog rule:[1]

$$\textit{isMember}(?x, A, r) \leftarrow \textit{isMember}(?y, B, s), \textit{isMember}(?x, ?y, t).$$

The meaning of an RT credential $[\![C]\!]$ is the Datalog fact or rule to which it translates. Let $\mathcal{C}$ be a set of RT credentials split into two disjoint subsets $\mathcal{C} = \mathcal{C}_f \amalg \mathcal{C}_r$ where $\mathcal{C}_f$ is the set of all membership credentials. The meaning of $\mathcal{C}$, which we denote as $[\![\mathcal{C}]\!]$, is the minimum model of the Datalog program $[\![\mathcal{C}_r]\!]$ using $[\![\mathcal{C}_f]\!]$ as input [Abiteboul et al. 1995]. The authorizer associates an access permission with a particular role, say $A.g$, that we call the *governing role*. Hence we formally define authorization in a given credential environment $\mathcal{C}$ as follows:

Definition 4.1. Given a credential set $\mathcal{C}$, entities A and E, and role g, *E is authorized for $A.g$ in $\mathcal{C}$*, denoted $\mathcal{C} \vdash E \in A.g$, if and only if *isMember*(E, A, g) is in $[\![\mathcal{C}]\!]$.

One appealing characteristic of the RT_0 trust management system is monotonicity. Negative credentials that explicitly remove entities from roles are not supported. Consequently if an authorizer has incomplete information she might deny access that would otherwise be granted, but she will never grant access that should have been denied. This property is essential in a WSN context where the unreliability of wireless communication together with the limited memory resources of sensor nodes make it impossible to *guarantee* complete information about all roles.

[1] Logical variables are shown prefixed with '?'

Our system requires the requester to provide all necessary credentials using some external means to obtain them. Methods for doing, for example, distributed credential chain discovery have been described [Li et al. 2003] but we feel they would be prohibitively expensive in a WSN context. The best approach for collecting a complete set of credentials may be application specific, and thus we regard it as outside the scope of this work.

The use of RT_0 means that principles cannot switch to roles of lesser privilege, as is often desirable in accordance with the principle of least privilege. However, variants of RT are available that support switching to less privileged roles [Li and Mitchell 2003b], and that could be used in SpartanRPC instead of RT_0. Also techniques for supporting certificate revocation in an RT-style trust management framework have been explored [Li and Feigenbaum 2002].

4.2. Program Logic

Our authorization model can be viewed as a client-server interaction; respective sides of the interaction protocol are summarized separately as follows.

4.2.1. RPC Server Side Logic. RPC service providers establish policy by assigning governing roles $A.g$ to remote interface implementations. Service providers also possess a set of assumed credentials $\mathcal{C}$ which establish an authorization environment by providing initial server side authorization policy. As we will describe in detail, the set $\mathcal{C}$ may grow as additional credentials are communicated to servers. Finally, in the presence of security, client invocations of any RPC service are not anonymous, but are performed on behalf of some entity B, which must be a member of the governing role $A.g$ to use the protected service.

In summary, access to an RPC level is allowed if and only if the property $\mathcal{C} \vdash B \in A.g$ holds, where:

— B is the identity of the RPC client
— $A.g$ is the governing role of the RPC service
— $\mathcal{C}$ are the credentials known to the RPC host server

RPC service programmers specify governing roles as part of module definitions—specifically at remote interface `provides` clauses. Hence, governing roles are associated with interface *implementations*, not interfaces themselves. This allows application flexibility, in that the same interface can be implemented with various authorization levels within the same network. Syntax is as follows:

```
provides remote interface I requires A.g
```

Note the minor modification to previously introduced syntax for remote module definitions via the `requires` keyword.

4.2.2. RPC Client Side Logic. In order to use a secure remote module, RPC clients wire to it as for unsecured modules (see Sec. 3.2), but with two additional capabilities: (1) the client specifies under what RT entity the invocations will be performed, and (2) the client may also specify credentials in their possession which are to be activated for use in the invocation. Syntax is as follows:

```
enable "C1, ..., Cn" as "B" for C.I -> [M].J
```

For any invocation made through this wiring the credentials $C_1, \ldots, C_n$ will be remotely added to the RPC server's database for the authorization decision, via a process detailed in Sec. 6. Note that these credentials *need not establish authorization entirely by themselves*, rather they will be *added* to the server's existing credentials, all of which will be used in the authorization decision. A special form of the `enable` clause using

```
interface InterestManagement {
    duty void set_interest(uint16_t sender_node, int temp_threshold,
                           int interval, int duration);
}

interface DataManagement {
    duty void set_data(uint16_t sender_node, uint16_t originator_node,
                       int temp_value);
}
```

Fig. 4. Directed Diffusion Interest and Data Management Interfaces

`"*"` for the list of credentials is also supported. This form indicates that all credentials known to the client should be communicated to the server.

Each node is deployed with a collection of ECC key pairs, one for each entity the node represents. When an invocation is made the entity B mentioned in the `as` clause of the dynamic wire is used in the request. The `as` clause is optional; if it is omitted a distinguished *default entity* is used for the invocation.

5. EXAMPLE: SECURE DIRECTED DIFFUSION

To illustrate our language design we have implemented a secured version of the well-known directed diffusion protocol [Intanagonwiwat et al. 2003] for ad-hoc routing of data in a sensor network. It is one example of the publish/subscribe paradigm for data gathering. In our secure version facilities for subscribing to a data stream are defined as secure RPC services by data stream providers. Directed diffusion supports multi-hop data collection, so this example illustrates how our link layer RPC service supports network layer communication. It also serves as a good benchmark application for empirical observations reported in Sec. 7. We provide another extended example in a real prototype application of SpartanRPC in Sec. 8.

The directed diffusion algorithm [Intanagonwiwat et al. 2003] allows a node to subscribe to a data stream by expressing an *interest* in it. In our example, an interest is expressed as temperature data above a given threshold. A certain data rate, expressed as a time interval between samples, is associated with each interest. Initially a node seeking temperature data floods the network using an interest with a low data rate. As data events find their way back to the interested node, that node selectively *reinforces* certain immediate neighbors by retransmitting the interest with a higher associated data rate to just those neighbors. Also, in our version of directed diffusion we imagine that it is to be implemented in a network comprising multiple security domains, and specify that subscription to data streams requires certain authorization levels as defined by policy. We omit the policy specification in this example to focus on the language API. An example policy specification is presented and discussed in Sec. 8.

5.1. Interfaces

Interest and data propagation are handled by separate interfaces, as shown in Fig. 4, each containing a single duty.

A node expresses interest in temperature data above a certain threshold and at a certain data rate by posting the `set_interest` duty on its neighboring nodes. The duration parameter of the `set_interest` duty specifies a lifetime of the interest. Once an interest expires it is removed from the node's interest cache. Temperature values are expressed as integers presumably corresponding to the output of an analog to digi-

tal converter. Similarly time intervals are expressed as integer multiples of some unit time, the exact value of which is arbitrary.

A node passes data to its interested neighbors by posting the `set_data` duty on those neighbors. The `originator_node` is the ID of the node where the data was originally observed; the node soliciting this data will typically want to know its provenance.

Both of these interfaces include the sender's node ID as an explicit parameter. The nodes use that information to track paths through the network. Although the node ids are also part of the low level radio packets sent between the nodes, we chose for demonstration purposes to manage the interest and data information strictly at a higher level in the protocol stack. As usual greater efficiency may be possible by mixing protocol layers.

5.2. Configuration

The interest and data caches, which we call "managers," are the two central components of our application. The interest manager provides the `InterestManagement` interface remotely and uses the same interface on other components. The data manager provides and uses the `DataManagement` interface in a similar way. Both components serve as their own component managers, using internal information to specify the destination nodes of each outgoing post operation.

The main configuration contains, in part, the following wiring for the interest manager:

```
enable "*" for InterestManagerC.NeighborSensors ->
                  [InterestManagerC].InterestManagement;
```

In this example, we assume the nodes are deployed with a single default entity as their identity. As discussed in Sec. 4.2.2, because the `as` clause is missing this wiring makes the invocation on behalf of that entity. We anticipate that single entity nodes will be common.

Because the interest manager provides and uses the same interface, it defines `NeighborSensors` as an alias for the `InterestManagement` interface that it uses remotely. When the interest manager posts the `set_interest` duty, that duty is invoked in all neighbors *currently* selected by its own, internal component manager. These post operations are authorized using credentials available to the invoking node; neighbors can be in multiple security domains.

5.3. Authorized Interest Management

The interest manager has a partial specification as follows, which we assume resides in a domain controlled by some `Admin` entity. Observe that its `InterestManagement` interface requires authorization for the `Admin.OK` role:

```
module InterestManagerC {
    provides interface ComponentManager;
    provides remote interface InterestManagement requires "Admin.OK";
    uses interface InterestManagement as NeighborSensors;
}
```

Because the interest manager is its own component manager, setting up target node addresses entails updating an internal `component_set`. In the case when a new interest is received the interest manager propagates that interest to all neighbors. This is done inside the interest manager's `set_interest` duty as shown in Fig. 5.

The "well known" local component ID of the interest manager is used to specify which component on neighbor nodes is to process the duty. The implementation of the `elements` command in the `ComponentManager` interface merely returns `remote_set` com-

```
remote_set.ids   = &remote_components;
remote_set.count = 1;
remote_components[0].node_id  = 0xFFFF;
remote_components[0].local_id = INTEREST_ID;
post NeighborSensors.set_interest( ... );
```

Fig. 5. Propagation of New Interests

puted above. Before the posting of `set_interest` returns, `remote_set` is used to prepare the outgoing packet. After the post is complete `remote_set` and `remote_components` can be reused without affecting any pending radio transmissions.

In the more complicated case where an interest is being reinforced, the interest manager must use information in the data cache to compute which neighbors need reinforcing. Although SpartanRPC allows a component manager to dynamically select neighbor nodes, the component used as a component manager is statically bound. Thus in this example the interest manager cannot switch its component manager to, for example, the data manager. To work around this, the interest manager communicates with the data manager using connections not shown here. With the data manager's help the interest manager computes appropriate neighbors dynamically before posting `set_interest` on those neighbors. The data manager implementation has a similar structure and authorization mechanism.

6. THE SPARTANRPC IMPLEMENTATION

In this section we describe the implementation of the SpartanRPC system using RT_0 trust management for authorization. We call our implementation Sprocket$_{RT}$[Chapin 2014]. Sprocket$_{RT}$ rewrites a SpartanRPC program into a pure nesC program and provides a supporting runtime system. Program rewriting converts remote duty postings into a nesC messaging protocol. The main task of the runtime system is to implement the encapsulated, underlying security protocols for authorization of remote duty postings.

6.1. Authorization and Security Protocols

Sprocket$_{RT}$ implements SpartanRPC authorization using a combination of public and symmetric key cryptography. We use the TinyECC library [Liu and Ning 2008] for public key functionality, and AES encryption for symmetric key functionality. TinyECC uses elliptic curve cryptography for more efficient public key operations in WSNs. Using AES has the benefit of hardware support on many current embedded platforms, e.g. those employing the Chipcon CC2420 radio.

There are three security protocols for authorized duty postings, illustrated in Fig. 6, that operate asynchronously. First, a credential exchange protocol in which RT credentials are communicated between nodes and authorization levels for various entities are computed, i.e. the *minimum model* as described in Sec. 4. Second, a session key protocol in which symmetric keys for multiple authorized service invocations are computed between a client and server. And third, an authorized service invocation protocol in which duty postings are generated and checked. This decomposition of authorized service invocation into three protocols supports efficiency especially through the use of symmetric keys for multiple service invocations. Its asynchronous nature is also appropriate in an asynchronous TinyOS setting.

6.1.1. Credential Exchange. SpartanRPC credentials are implemented as signed certificates. All SpartanRPC-enabled nodes contain a certificate sender component and

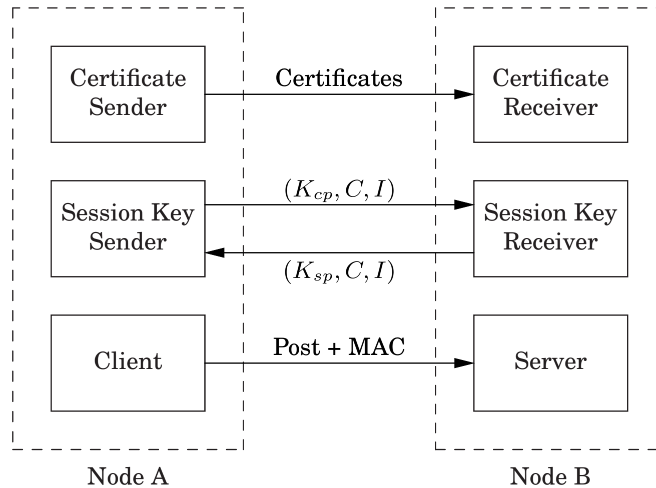

Fig. 6. SpartanRPC Security Protocol Elements

a certificate receiver component to transfer certificates between nodes and to verify them and interpret the credentials they represent. Both components run as background daemons, performing their function asynchronously. A SpartanRPC-enabled node is deployed with a collection of certificates in read-only storage representing that node's credentials, which are determined by some external means. Once the node is booted, the certificate sender starts a periodic timer. When the timer fires, the node link-layer broadcasts (i.e. only to neighbors) all certificates in its certificate storage that are mentioned in the `enable` clauses of the dynamic wires used during program execution. To prevent adjacent node certificate broadcasts from colliding, the certificate broadcast interval is modulated randomly by $\pm 10\%$. For example if the nominal broadcast interval is one minute, the actual time varies randomly between 54 and 66 seconds.

The certificate distribution strategy is robust in the face of new nodes being added to the network or intermittent radio connectivity. If a node fails to receive certain certificates from its neighbors it will have another opportunity to do so when those neighbors rebroadcast their certificates. There is a trade off between the broadcast interval, responsiveness, and network energy consumption. A short broadcast interval allows authorizations to succeed "quickly" since neighbors become aware of the necessary credentials early, but at the cost of increased radio traffic and power consumption.

Once a newly received certificate has been verified, the credential it represents is extracted and stored. The credential storage also contains the RT_0 minimum model implied by the currently known collection of credentials. Each time a new credential is added to storage, the minimum model is updated. This is done by repeatedly applying authorization logic inference rules implied by the credentials to the current model until a fixed point is reached, i.e. a logical closure [Li and Mitchell 2003a]. Thus, each node maintains a local view of authorization levels for network entities based on received credentials.

Our certificate representation of an RT credential contains the public keys denoting entities mentioned in the credential. Roles are identified by one byte numeric codes and are scoped by the entity defining the role. Credential forms are distinguished by numbers $\{1, 2, 3, 4\}$. Certificates are also signed by their issuing authority. Conveniently, the issuing authority is always mentioned in a credential, e.g. the issuing authority

$A.r \leftarrow B.s \cap C.t$

4	A (40)	r	B (40)	s	C (40)	t	sig (42)	chk (2)

Fig. 7. Intersection Certificate Format (parenthesized numbers indicate byte counts)

of $A.r \leftarrow B$ is A, so the means to verify the certificate (i.e. the relevant public key) is always included with it for free. This does not introduce a security problem. Since entities are identified directly by their keys, an attacker who creates a new key is simply creating a new entity.

The over-the-air format for the intersection certificate is illustrated in Fig. 7. The other certificate forms are organized in a similar way. The certificate starts with a certificate type identifier byte, and then follows the written form of the RT credential with a signature appended to the end. Role names are mapped to single byte role numbers specified by the entity defining the role.

Certificates range in size from 124 bytes for the membership credential to 166 bytes for the intersection credential. This is larger than the maximum payload size limit of TinyOS T-Frame Active Message packets as transported by IEEE 802.15.4 [IEEE 2003; Hui et al. 2008]. It is much larger than the default maximum payload size of 28 bytes used by TinyOS [Levis]. Consequently the certificates are fragmented into four messages requiring a maximum payload size of 46 bytes. The choice of using four fragments is a trade off between an excessive number of fragments on one hand and excessive memory use for packet buffers on the other. Also the session key negotiation messages described in Sec. 6.1.2 require 44 bytes in any case.

Verification of RT certificates is the most computationally expensive component of our system as we discuss in Sec. 7. Thus, we want to minimize the amount of effort spent on verification. To this end, we append a 16-bit Fletcher checksum to each certificate. Nodes maintain a database of certificate Fletcher checksums to quickly check whether a certificate has already been received and verified. It is necessary to include this checksum rather than rely on the checksum provided by the underlying network stack because the later value covers frame headers that are not part of the certificate. Fletcher checksums are commonly used in WSNs since their error detection properties are almost as good as CRCs with significantly reduced computational cost [Fletcher 1982].

6.1.2. Session Key Negotiation. Public key cryptography is much too computationally expensive to use for routine duty posting authorizations. Sprocket's run time system addresses this by negotiating session keys between the sender (client) and receiver (server) nodes.

The client maintains a session key storage that is indexed by the triple (N, C, I) where N is the remote node ID, C is the remote component ID, and I is the remote interface ID. A session key is thus created for each combination of these IDs. The server also maintains a session key storage indexed by (N, C, I). In this case N is the node ID of the client and C, I are the component and interface IDs on the server to which that client is communicating. Since any given node can be both server and client, each session key storage entry has a flag to indicate the nature (client-side or server-side) of the session key.

The first time a client attempts to access a service on a particular server, it will send a session key negotiation request. When a server receives a session key negotiation request message from a client node N containing the public key K_{cp} of the requesting entity (as mentioned in the as clause of the dynamic wire) and the (C, I) address of the desired service, the following steps are taken:

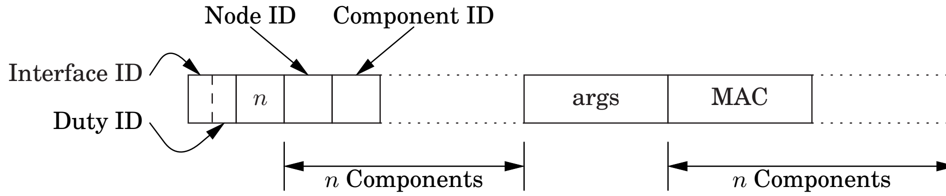

Fig. 8. Duty Post Message

(1) Authorization of K_{cp} for service (C, I) is checked using the RT minimum model computed by the certificate receiver. If authorization fails nothing more is done.
(2) A session key is computed using elliptic curve Diffie-Hellman and added to the session key storage under the proper (N, C, I) value. The key is stored as a remote client key.
(3) A message is returned to the client containing the server's public key K_{sp} and the original (N, C, I) values used by the client. This is so the client is able to compute the same session key and associate it with the proper endpoint from its perspective.

The session key negotiation protocol is a simple Diffie-Hellman key agreement protocol that combines the public key of the peer entity with the private key of the local entity. The implementation does not include any nonces as would be done, for example, with the ECMQV protocol [ISO 2008]. As a result any renegotiated session keys between the same entities would be identical. However, this is not a serious problem because the implementation does not currently renegotiate session keys. Furthermore the ECMQV protocol entails three exchanged messages and additional computations and so would further increase the burden on nodes.

A potentially more serious concern is that the simple protocol described here might be vulnerable to a man in the middle attack, whereby an active attacker negotiates independent session keys with each peer and is able to modify messages sent between those peers. However, in RT private keys are identities directly so a man in the middle without access to either the client or server's private key would appear as a new entity, presumably without authorized access.

6.1.3. Authorized RPC Invocations. Authorized RPC invocations are made with MACs on invocation messages using AES session keys. Verification of a MAC by the receiver constitutes authorization since a session key for a particular client and service is negotiated only after client credentials have been verified as providing proof of compliance to authorization policy. Fig. 8 shows the format of authorized invocation request messages. The interface and duty IDs, both four bit values, are packed into the initial byte of the message. Following is the number of receivers, the (N, C) addresses of the receiving components, and the duty arguments.

Since invocation of an RPC service on multiple hosts can be made at once in a fan-out wiring (see Sec. 3), a single invocation request message may apply to multiple servers in the neighborhood of the client. To conserve bandwidth, fan-out invocation messages include multiple MACs, since separate session keys are negotiated with each of n servers, allowing a single message to invoke the same service on the servers. If the duty arguments consume d bytes of data, then invocation messages consume $2 + 6n + d$ bytes. In the common case where $n = 1$ invocation messages contain 8 bytes of overhead. As we describe above our implementation uses a 46 byte message buffer size as required for sending certificate fragments. Our experience suggests that using the same buffer size for invocation messages allows for reasonable values of both d and n.

To conserve space in the invocation messages we only use a 32 bit MAC. Such a small MAC would not normally be considered secure. However, wireless sensor networks generate data so slowly that attacking even such a short MAC is not considered feasible [Karlof et al. 2004; Luk et al. 2007].

6.1.4. More on Security Properties. As in Sec. 1.2, we stress that our scheme is intended to enforce authorization, which we achieve via the protocols described above. Integrity is a side effect of this, since we use MACs to enforce authorization, which are computed over complete message payloads and are verified by the receiver. Although confidentiality is not directly supported by our current protocols, it could be easily added. In particular payloads could be encrypted using negotiated session keys (payloads are currently sent as plaintext).

Our system does not provide any form of replay protection out of the box, but this can be added at the application level. For example an application could pass a counter or time stamp as an additional duty argument. The server could verify that the argument increases monotonically as a simple form of replay protection. The space required for counter or time stamp information would increase message sizes, but this is unavoidable for any solution to replay protection. Delegating replay protection to the application is appropriate since SpartanRPC is intended to be a low level infrastructure on which more complex systems can be built. Furthermore the need for replay protection is likely to be application specific.

Perhaps the most problematic vulnerability of our system is to denial of service (DoS) attacks– unlike e.g. replay, for which countermeasures can be readily implemented at the application level, it is not clear how DoS attacks could be mitigated without significant changes to the underlying security protocols. For example, a constant flood of certificates over the correct channel would place receiving nodes in a constant state of signature verification, potentially consuming large amounts of CPU time and energy. Mitigation of such attacks is outside the scope of this work but has been discussed in the literature [Raymond and Midkiff 2008].

A note on multicast security. Fan-out wirings are a common idiom, and provide a form of multicast communication. However, the use of MACs for security in a multicast setting presents well-known challenges. In particular, while n-way Diffie-Hellman can be used to negotiate secret keys between n actors, such a scheme cannot be used in light of the possibly heterogeneous authorization requirements we anticipate. For example, suppose a node A fan-out wires to service s on distinct nodes B and C, and suppose also that A is authorized for s on both nodes but that B is not authorized for s on C and vice-versa. If a single session key were negotiated between A, B, and C in this case, then B could make unauthorized use of C's version of s and vice-versa. While a variety of techniques have been proposed to mitigate this problem [Canetti et al. 1999], most typically rely on very large multicast groups and are not applicable in our setting. Thus, we handle fan-out wirings using multiple MACs as described above.

6.2. Identifying Services Over the Air

RPC service endpoints are identified by the 4-tuple (N, C, I, D) where N is the TinyOS ID of the node on which a duty is implemented. C is the local component ID assigned to each component that provides a remotable interface. I is an interface ID, required since a component may provide more than one remotable interface. Interface IDs are component-level unique. Finally D is a duty ID, which must be interface-level unique.

In the current version of Sprocket$_{RT}$, (C, I) values are assigned statically by an arbitrary (automated or social) process. Sprocket$_{RT}$ accepts configuration files that define the association between (C, I) values and the entities to which they refer. Duties are numbered in the order in which they appear in their enclosing interface definitions.

Some RPC systems, such as ONC RPC, allow each node to provide a registry of RPC services available on that node [Srinivasan 1995]. When a large number of RPC services are provided by a node it becomes unreasonable to expect clients to have hard coded knowledge of the endpoint identifiers for all those services. Instead clients communicate with the single well known registry to obtain endpoint identifiers that were dynamically assigned. In contrast we assume this configuration information is known a priori to all interacting actors. It is unclear if sensor networks could benefit from a more sophisticated technique for defining and communicating endpoint identifiers, but it would be an interesting topic for future work.

6.3. Rewriting SpartanRPC to nesC

There are five major features requiring SpartanRPC-to-nesC rewriting by Sprocket$_{RT}$: interface definitions, call sites where remote services are invoked, duty definitions, dynamic wires, and server components providing remote interfaces. In addition Sprocket$_{RT}$ generates a stub component for each dynamic wire, and a skeleton component for each remote interface. Finally Sprocket$_{RT}$ generates configurations that wrap server components. Here we summarize rewriting strategies for these features.

6.3.1. Interfaces, Call Sites, and Duty Definitions. Duty declarations in interfaces are rewritten to command declarations by substituting `command` for `duty`. Since nesC commands are allowed to have arbitrary parameters, duties with parameters present no complications. Sprocket$_{RT}$ verifies that if an interface contains a duty, then the only declarations in that interface are duties. Sprocket$_{RT}$ further verifies that the parameters of each duty, if any, conform to the restrictions described in Sec. 2.2.

Call sites where duties are posted are rewritten to command invocations by substituting `call` for `post`. Only post operations applied to duties are rewritten in this way. Finally, duty definitions are rewritten to command definitions by also substituting `command` for `duty`.

6.3.2. Authorization Interfaces. The rewriting process makes use of two interfaces that mediate the interaction between the Sprocket$_{RT}$ generated code and the security processing components of the run-time system. Fig. 9 shows how a message, entering from the left, is extended with authorization information by the client and then passed to the server where the authorization information is checked.

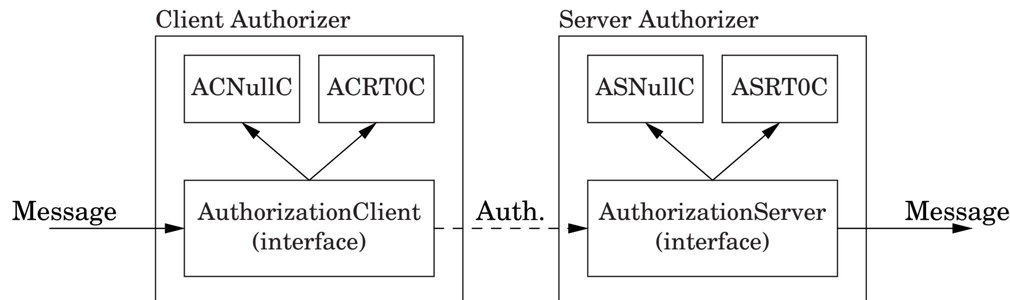

Fig. 9. Client/Server Authorization Architecture

The `AuthorizationClient` interface abstracts the details of how an authorized message is prepared before being sent. The `AuthorizationServer` interface abstracts the details of how authorized messages are processed after they are received. This design allows for pluggable authorization mechanisms. Future versions of Sprocket$_{RT}$ could

```
Dynamic Wire
    ClientC.I -> [SelectorC].I;

Rewritten as...
    components Spkt_n;
    ClientC.I -> Spkt_n;
    Spkt_n.ComponentManager -> SelectorC;
    Spkt_n.AuthorizationClient -> AuthorizerC;
    Spkt_n.Packet -> AMSenderC;
    Spkt_n.AMSend -> AMSenderC;
```

Fig. 10. Dynamic Wire Rewriting

potentially support other authorization schemes than those described here, in a modular fashion.

The authorization interfaces provide their services in a split-phase manner so that potentially long-running authorization computations can be performed while allowing the node to continue other functions. In the current implementation, two kinds of authorization are supported. On the client side the precise method used depends on the dynamic wire over which a particular communication takes place. On the server side it depends on the presence of a `requires` clause on the remotely provided interface.

The full RT_0 mechanism is supported by client and server components `ACRT0C` and `ASRT0C` respectively (details about this mechanism are discussed in Sec. 4 and Sec. 6.1). In addition a "null" authorization is supported by client and server components `ACNullC` and `ASNullC` respectively. The null authorization components perform no operation. They are used for dynamic wires that do not require authorization and remote interfaces provided publicly by servers.

6.3.3. Dynamic Wires. In the following, we use italics for metavariables that range over arbitrary identifiers. The reader is referred to the rewriting schema defined in Fig. 10. Configurations containing dynamic wires are rewritten to configurations that statically wire the using component *`ClientC`* to a stub `Spkt_`*n* that interacts with the appropriate component manager *`SelectorC`* and that handles radio communication. Every stub generated by Sprocket$_{RT}$ is uniquely identified over the scope of the entire program by an arbitrary integer n. The *`AuthorizerC`* component is `ACNullC` in the case where no authorization is requested.

In contrast a dynamic wire using either an `enable` or `as` clause is rewritten the same way except that the *`AuthorizerC`* component is `ACRT0C`. Furthermore, the list of enabled credentials is added to local certificate storage by Sprocket$_{RT}$. Certificates in storage are periodically beaconed at run-time as described above. Finally, the entity on whose behalf the RPC invocation is performed is specified in the session key negotiation message sent to the server, also as described above.

The `Spkt_`*n* stub provides the same interface provided by *`ClientC`*. Wherever a duty is posted by *`ClientC`* in source code, the rewritten call invokes code in the stub that was specialized to handle that duty. The stub calls into the component manager at run time to obtain a list of the dynamic wire's endpoints and then prepares a data packet containing remote endpoint addresses and marshaled duty arguments. Finally the stub calls through the `AuthorizationClient` interface to perform whatever authorization is needed.

6.3.4. Remote Services. For nodes supporting RPC services, Sprocket$_{RT}$ generates a skeleton component for each remote interface provided. This skeleton contains a task

corresponding to each duty provided in the interface, and every generated skeleton is distinguished by a unique integer n taken from the same numbering space as the generated stubs.

When messages are received on a node that provides RPC services, they are examined to see if they are duty postings and thus to be handled by a skeleton. If so, the `AuthorizationServer` interface is used to authorize the message. If authorization succeeds, the task corresponding to the specified duty is posted. That task simply calls into *ServerC* through the original interface *I*. Thus the task-like behavior of duties is ultimately implemented using actual nesC tasks inside the server skeletons. Duty parameters are conveyed via module-level variables accessed by the duty tasks (since nesC tasks do not take formal arguments).

For each component that provides at least one remote interface, Sprocket$_{RT}$ creates a configuration that wires the corresponding skeleton(s) to that component. This new configuration wraps the original component and replaces uses of the original component in other configurations in the program.

7. EMPIRICAL ANALYSIS OF OVERHEAD

The practicality of our system depends on its cost in terms of memory and processor overhead. In this section we report on performance measurements made on our implementation. In summary, we show that our combined use of public and private key cryptography in the underlying security protocol imposes a low amortized cost over time, despite high costs for initial authorizations.

Test Environment and Programs.. Since many communication chips now support hardware AES encryption, we were primarily interested in demonstrating performance using that feature. In particular, the popular MEMSIC TelosB wireless sensor mote [MEMSIC] uses a Chipcon CC2420 transceiver with hardware encryption. Unfortunately, the standard TOSSIM simulation environment does not model hardware encryption for TinyOS 2.1, so all of our tests were performed on real hardware. We used TelosB motes, with 10 KB of RAM, 48 KB of ROM and an 8 MHz MSP430 microcontroller running TinyOS 2.1.2 [TinyOS].

We exercised our system using both small test programs and using our implementation of the directed diffusion example described in Sec. 5. The small test programs consisted of a simple client/server pair where the client repeatedly sent a message containing a single 16 bit value to the server. The purpose of these tests was to explore the overhead induced by our system with minimal obscuring effects from application logic. The percentage overhead observed with the small programs is thus a worst case overhead. In contrast, the directed diffusion example allowed us to test the behavior of the system in a more realistic, long-running setting. It serves as a demonstration that our system is feasible in practice, and allowed us to exercise our system in a multi-mote, multi-hop network environment.

7.1. Memory Overhead for Security Features

The Sprocket$_{RT}$ run time system uses several memory caches to hold key material, credential information, and the minimum model implied by the set of known credentials. These caches are statically allocated but must be stored in RAM since their contents are dynamic. Table I summarizes the RAM consumption of the various storage areas used by the current implementation.

The number of items in each cache are tunable parameters. The optimum settings depend on the intended application. The values in Table I attempt to strike a balance between usability and flexibility on one hand and excessive memory consumption on

Table I. RAM consumed by various storage areas

Storage Area	*# Items*	*Bytes/Item*	*Total Bytes*
Session Keys (n_k)	10	22	220
Public Keys (n_p)	12	40	480
Credentials (n_c)	12	16	192
Model (n_m)	16	6	96
Total			**988**

Table II. Memory consumption of test programs

Test Program	*RAM Bytes*	*ROM Bytes*
Baseline Client	349	10982
Baseline Server	283	10490
SpartanRPC Client	2222	23108
SpartanRPC Server	2126	23394

the other. In applications where these needs are more clearly known a priori, the sizes of the caches can be adjusted to potentially result in lower memory consumption.

Table II shows the overall memory consumption of two small client/server pairs. The baseline pair handle all communication through normal Active Message packets that are explicitly programmed by the user. The SpartanRPC pair uses our system which includes support for certificate distribution and verification, session key management, authorization logic, and MAC computations.

Although the overhead incurred by the Sprocket$_{RT}$ runtime system is significant on our test platform, nearly 80% of RAM and 50% of ROM resources are still available. Furthermore, these memory usage numbers scale to denser neighborhoods and extended RPC services.

7.2. Transient and Steady State Processor Overhead

The execution performance of our system displays two distinct behaviors. The first is a transient behavior that occurs after a node boots when certificates are exchanged and session keys are negotiated between the new node and its neighbors. The second is a steady-state behavior that occurs during normal operation. The transient overhead of our system is large but the steady state overhead is not. In a quasi-static environment where new nodes enter the network infrequently the transient costs are amortized and it is the small, steady state overhead that dominates.

To explore the steady state overhead three tests were conducted.

(1) A baseline test where the message handling was done explicitly using traditional Active Message interfaces.
(2) A duties test where the Sprocket$_{RT}$ system was used but no authorization was requested. This is equivalent to using the authorization components `ACNullC` and `ASNullC` in Fig. 9.
(3) A MAC test where authorization was requested but where the session key storage areas were preloaded with appropriate session keys.

Table III shows the maximum rate at which messages could be sent and received by the test programs mentioned above. Note that the MAC test made use of the hardware assisted AES support provided by the CC2420 radio chip. These results show that maximum message send rates decrease by a factor of 7% due to the addition of our duties program logic (our security API), and further decreases by a factor of 25% due to MAC calculations. We note that the latter overhead would be incurred in any system using CC2420 MAC calculations.

Table III. Maximum message transfer rate

Test	*messages/s*	*% Reduction*
Baseline	128	–
Duties	119	7.0
MAC	87	32.0

The transient runtime overhead of our system can be subdivided into three primitive operations: the time required to transmit and verify a certificate, the time required to build the minimum model, and the time required to negotiate a session key. Two of these operations require lengthy public key computations and dominate the transient behavior of our system. Thus the performance of our system in this regard is closely tied to the performance provided by TinyECC, which we used with default settings (no optimizations). Table IV shows the times required for each of the primitive transient operations in our implementation.

Table IV. Transient processing time

Operation	*Time*
Certificate Verification	82s
Minimum Model Construction	370μs
Session Key Negotiation	80s

The time required to build the minimum model is directly related to the number and nature of the credentials involved. In our test we used a collection of five representative credentials, one of each type. In any case this time is entirely negligible compared to the other transient operations.

The time quoted for session key negotiation represents the time required for both negotiating partners to compute the session key. In the current implementation the two negotiating nodes do this sequentially with the server node computing the session key before responding to the client node. This was done in case the session key computation failed on the server to ensure that the client does not falsely believe a session key was successfully negotiated.

7.3. Transient State Times for Directed Diffusion

As argued above, the overhead imposed by our system is primarily the time the network spends in a initial transient state when credentials are verified and session keys are negotiated. Subsequently, the network enters a steady state during which the main cost is a 32% reduction in *maximal* message send rates due to hardware MAC computation. In order to evaluate the performance of our system in a realistic application, we therefore quantified the transient state times of the secure directed diffusion application described in Sec. 5. In our experiments we elected a single node to repeatedly express an interest and we observed how long was required for that interest to flood the network. This time depends on three major factors:

(1) The number of certificates transferred.
(2) The number of neighbors for each node.
(3) The number of hops to the “far” edge of the network.

We conducted two experiments, one on a single hop (star) network and another on a multi-hop (mesh) network.

In the single hop case, transient time T can be described by the following equation:

$$T = n_c B + V + n_n K$$

where B is the certificate broadcast interval, V is the certificate verification time, K is the session key negotiation time, n_c is the number of certificates and n_n is the number of neighbors. Since B was set to 90 seconds, which is greater than V, certificate verification for n_c certificates takes time n_cB+V given a 90 second system initialization period. And since session keys need to be negotiated with n_k neighbors in turn, T also comprises a n_nK delay. Table V shows the transient time required to flood a network where all nodes are one-hop neighbors of the root node. Values are given for three different policies with different numbers of certificates transferred from the root to the neighbors.

Table V. Single hop transient time

# neighbors	*1 Cert*	*2 Certs*	*3 Certs*
1	4m03s	5m27s	6m52s
2	5m16s	6m50s	8m24s
3	6m32s	7m57s	9m30s
4	7m50s	9m22s	10m51s

Table VI. Multi-hop transient time

Run	*1 hop*	*2 hops*	*3 hops*
1	4m05s	7m24s	9m10s
2	3m12s	5m12s	6m30s
3	3m57s	7m37s	9m15s
4	4m09s	7m15s	8m49s
Average	3m51s	6m52s	8m23s

We explored the behavior of our system in a multi-hop environment by creating a linear mesh network. Each node (except the root) had a single downstream neighbor. All nodes were booted simultaneously and the time required for interest information to reach each node was observed. The policy used required only a single certificate to be transferred between nodes. Table VI shows the results of several runs.

The reason for variations in transient times over each run was due to a randomized element in the protocol, specifically a randomized $\pm 10\%$ interval in certificate broadcast times to avoid collisions. In these results it is essential to note that for hops > 2, extra transient time is comprised solely of session key negotiation times (80s per session key, see Table IV) that are forced by duty postings as interests propagate through the network. Certificates are broadcast and verified in parallel throughout the network upon system boot up, during the same time period required for the root's interest to propagate through the first and second hops.

8. A PROTOTYPE APPLICATION

To evaluate the performance of SpartanRPC in a real application setting, we have used the system to implement secure versions of data collection and sampling control protocols in an environmental monitoring system. The Snowcloud system [Frolik and Skalka 2013; Moeser et al. 2011] is a WSN developed at the University of Vermont for snow hydrology research applications. It is based on the MEMSIC TelosB mote platform running TinyOS, and has seen multiple field deployments. Typical deployed systems comprise 4-8 sensor nodes but the technology is currently scalable to arbitrary numbers of nodes. For data collection and sampling rate control, the system also includes a handheld "Harvester" device. This device incorporates a TelosB mote to establish a network connection when in radio communication with the deployment. Users transport the device to and from deployment sites, and interact with the sensor node network by issuing commands from a simple push-button interface. A Harvester device and a deployed Snowcloud sensor tower are pictured in Fig. 11. The scheme described here has been implemented and tested in our test network at UVM, which uses the same software and hardware platforms as in our active deployments.

In our secured version of the Snowcloud system, the goal is to treat data collection and sampling rate control as protected resources requiring authorization. Furthermore, sampling rate modifications should require a higher, "administrator" level of

Fig. 11. A Snowcloud Sensor Node (L,C) and Harvester Device (R).

authorization than data collection. That is, only system engineers should be able to perform control operations, whereas data end-users making field visits should be able to collect data. Snowcloud sensor node code in particular makes use of nearly every resource available on the mote– including timing, sensor I/O, radio messaging, and flash memory, not to mention CPU and main memory. Thus, it is a robust example of the interaction of SpartanRPC with mote resources in real applications.

The system described here is also informative since it can be easily ported to other similar application settings. That is, WSN application settings wherein multiple users of various authorization levels need to interact with the same network in control or collection capacities, as mediated by security policy. The SpartanRPC API allows straightforward retasking of authorized service implementations to these various settings. Furthermore, the RT authorization logic supports collaboration between multiple social domains, by allowing security policy to be managed in a decentralized manner as we illustrate below.

8.1. Security Policies

To specify and implement the security policies informally described above, we consider the sensor network and the Harvester single node "network" as separate security domains, each with their own set of credentials. The sensor network is always endowed with administrator-level credentials. If a Harvester is to be used by a system engineer, its mote is also endowed with administrator-level credentials, whereas a Harvester to be used by a data end-user is only endowed with user-level credentials. When a Harvester is introduced to the sensor network, its resource accesses are mediated by its authorization level. Since credentials are unforgeable, a user-level Harvester can never be used for sensor network control even if it is reprogrammed.

Sensor nodes within the network possess four credentials, as follows. In these credentials the Snowcloud domain is abbreviated SC. Authority to collect data and control sensors in the network are governed by the roles $SC.Col$ and $SC.Con$, respectively. Credential (1) says that control authority contains collection authority. (2) says that nodes in the Snowcloud domain have control authority. (3) says that any entity in a Snowcloud collaborator's Usr role has collection authority. (4) says that the node identified

by NId is in the Snowcloud domain.

$$(1)\quad SC.Col \leftarrow SC.Con \qquad (2)\quad SC.Con \leftarrow SC.Node \qquad (3)\quad SC.Col \leftarrow SC.Collab.Usr$$

$$(4)\quad SC.Node \leftarrow NId$$

When invoking remote services, the node will do so on behalf of the entity NId. It will also be imaged with the NId private key for session key negotiation.

Any Harvester within the Snowcloud domain is then provided with the credential $SC.Node \leftarrow HId$ and the HId private key issued by Snowcloud domain administration. This will provide that Harvester with collection and control authority in the domain. For Harvesters to be provided to collaborators, the Snowcloud administrators issue a credential establishing the institution as a collaborator, while the institution itself may define and manage policy for their Usr role membership. For example, the University of New Hampshire (UNH) can be established as a collaborator with credential (5) issued by Snowcloud domain administration, and may specify role membership with the credential (6) issued by UNH domain administration:

$$(5)\quad SC.Collab \leftarrow UNH \qquad (6)\quad UNH.Usr \leftarrow UsrID$$

These two credentials, along with the $UsrID$ private key, are imaged on Harvesters issued to UNH collaborators for data collection, but which remain unauthorized for control.

8.2. Implementation

Resources themselves are accessed through a secure command dissemination protocol, that is modeled upon the TinyOS Dissemination protocol (as described in TEP 118). In short, protected RPC services establish network level broadcast channels requiring authorization for use. Commands are communicated to the network over these channels, and different channels are used for different sorts of commands.

In more detail, command broadcast services can be specified as a duty in a remotable interface:

```
interface SpDisseminationUpdate { duty void change(command_t new_value); }
```

To implement e.g. the control command channel, the following module can be defined and included on sensor nodes in the Snowcloud domain:

```
module ControlDissemC {
    provides remote interface SpDisseminationUpdate requires "SC.Con";
    uses            interface SpDisseminationUpdate as NeighborUpdate;
    provides        interface ComponentManager;
}
implementation { ... }
```

In the implementation, the provided `SpDisseminationUpdate` interface accepts command invocations from neighbors, but requires them to be authorized for the $SC.Con$ role. Commands are relayed to all other neighbors (i.e. disseminated) via the used `NeighborUpdate` interface; those neighbors are identified by the provided `ComponentManager`.

To use this component, both sensor and Harvester nodes can configure it through the following component instantiation and wiring, where the component's `NeighborUpdate` interface is wired remotely to neighbors:

```
components ControlDissemC as ControlChan;
activate "*" for
    ControlChan.NeighborUpdate -> [ControlChan].SpDisseminationUpdate;
```

Note that a node must be endowed with the appropriate credentials for this wiring to be useful.

This same code pattern can be used to implement a data collection request channel, protected by the $SC.Col$ role instead of $SC.Con$. In response to an authorized control command invocation, sensor nodes will modify their behavior appropriately, whereas in response to authorized data collection requests sensor nodes will report their data using collection tree protocol (TEP 123) to the Harvester.

Note that since the only "border" between security domains in this scenario is between the Harvester and its neighbor(s), Snowcloud scalability is not affected. Only authorization between the Harvester and its one-hop neighbors needs to be established no matter what the network size, and since areal coverage is the goal of a deployment, network densities remain fairly constant where neighborhoods are on the order of 1-5 nodes in conceivable deployments.

8.3. Results

Results can be characterized according to both the application end-user experience and to quantitative aspects. As detailed in Sec. 7, a one-time transient overhead is imposed for initial credential exchange and session key negotiation when a Harvester is first introduced to the network. However, since data collection for a network after several months of deployment can take up to an hour, this overhead is relatively insignificant. And steady-state overhead is small, and does not affect data collection rates. Further, subsequent field visits will not impose transient overhead since negotiated keys can be cached in non-volatile memory. Thus, authorized user experience is not significantly impacted by the addition of security.

From a quantitative perspective, the most important measurements to consider for this application, beyond the general ones already considered in Sec. 7, are RAM and ROM consumption of the insecure and secured versions of the Harvester collection protocol. We have to consider whether layering SpartanRPC security over a realistic application will overrun the resources available to a mote platform. Relevant measurements are as follows.

Table VII. RAM and ROM use for Snowcloud versions

Program	*RAM Bytes*	*ROM Bytes*
Insecure Harvester	2274	24316
Secure Harvester	4771	35834
Insecure Sensor Node	2868	36254
Secure Sensor Node	5417	48616

Both RAM and ROM consumption are significantly increased by the addition of SpartanRPC security to this application. However, these numbers are within operating parameters, and the Sprocket implementation of SpartanRPC described in Sec. 6 has not yet been optimized for space efficiency in any way; improvements in this respect can be made but are out of scope for this work.

9. RELATED WORK

Extending wireless sensor network software platforms with support for secure interactions between domains has been studied in previous research on SSL for WSNs [Jung et al. 2009]. However, this work was focused on extending the Internet to WSNs (a.k.a. "IP for WSNs"), whereas SpartanRPC is a more general system for enhancing secure communications *within* a WSN. Research on WSN security has also addressed secure

routing [Karlof and Wagner 2003], link layer security [Karlof et al. 2004], cryptography [Bertoni et al. 2006], key distribution [Çamtepe and Yener 2005], and hardware issues [Perrig et al. 2004]. In contrast to these low-level systems, SpartanRPC provides language-level abstractions for secure RPC services. Perhaps even more closely related in this same vein is a system for establishing fine-grained, "node-level" policies in WSNs [Claycomb and Shin 2011]. However, this work is more focused on group-based key negotiation and distribution, and while it does offer a policy language, it is rooted in implementation details and not a separable specification as in our system. Also, they do not provide a language API for integrating their system into secure applications as in SpartanRPC.

Previous related work also illustrates interest in and useful applications of RPC in embedded networks. For example, the Marionette system uses network layer RPC for remote (PC-based) analysis and debugging of WSNs [Whitehouse et al. 2006]. The Fleck operating system provides a small pre-defined set of RPC services for WSN applications, while the trustedFleck system extends this with a form a secure RPC [Hu et al. 2009; Hu et al. 2010]. S-RPC provides an RPC facility for sensor networks that allows remote services to be added to the system dynamically [Reinhardt et al. 2011]. SpartanRPC differs from these systems in that it extends the nesC programming language (unlike trustedFleck) to allow programmer definition of secure RPC services (unlike S-RPC) that can be accessed by nodes within the network itself (unlike Marionette). Our system is similar to and inspired by TinyRPC [May et al. 2007], except the latter does not provide security and has a different semantics that are not as expressive as our approach.

TeenyLIME allows application programs to access an abstract "tuple space" that is the union of tuple spaces on the local node and the immediately neighboring nodes [Costa et al. 2007]. This provides an alternative to RPC for uniformly accessing remote and local data. However, interaction with the middleware is by way of a dedicated API; there is no attempt to provide a true RPC mechanism. Also TeenyLIME does not address issues of access control.

Secure Middleware for Embedded Peer to Peer systems (SMEPP) is a general framework for creating security sensitive applications from a distributed network of embedded peers [Brogi et al. 2008]. SMEPP Light [Vairo et al. 2008] is a reduced version of SMEPP to address the resource constraints of wireless sensor networks. SMEPP Light provides a publish/subscribe communication model using directed diffusion to distribute "events" to all subscribers and symmetric key cryptography to provide confidentiality and data integrity within a group of nodes. However, SMEPP Light is not integrated into a programming language and does not provide a remote procedure call mechanism. Furthermore SMEPP Light only supports a simple model of access control based on group membership.

High level macroprogramming languages such as Kairos [Gummadi et al. 2005], Regiment [Newton et al. 2007], and even Flask [Mainland et al. 2008] provide a way to program the entire network as a single entity. These systems attempt to hide not only the inter-node communication from the programmer, but also the entire node level programs. SpartanRPC operates at a much lower level and also, unlike these macroprogramming systems, addresses access control issues in networks containing multiple security domains.

Whole network programming of wireless sensor networks has also been investigated using mobile agents in systems such as Agilla [Fok et al. 2009] and Wiseman [González-Valenzuela et al. 2010]. However, like the macroprogramming systems mentioned previously neither of these systems address issues related to access control in the presence of multiple security domains.

10. CONCLUSION

We have designed and implemented SpartanRPC, a dialect of nesC with a light weight, link-layer, secure RPC API. SpartanRPC is a middleware technology supporting secure WSN applications comprising multiple security domains. It is ideal for settings in which multiple networks administered by distinct social entities cooperate to obtain a holistic behavior. As discussed in Sec. 9, currently no other WSN security architectures support multiple security domains or principled techniques for communication between them. Thus, SpartanRPC provides crucial tools for next-generation WSN applications wherein multiple, distinct security domains interact.

RPC communication in our system is implemented using a modification of existing nesC abstractions, specifically module wirings. In SpartanRPC, module wirings connect to remote services dynamically. Furthermore, these connections are mediated by authorization levels specified by an access control policy defined in the trust management language RT [Li et al. 2002], and authorization is proved by presentation of RT credentials by requesters. Our implementation is based on public keys, supporting an open-world security model where shared secrets need not be known a priori. Underlying security protocols defend against man-in-the middle attacks through the use of a Diffie-Hellman protocol, ensuring that only authorized principals may access resources. We have reported on testing and performance evaluations, providing evidence of the practicality of SpartanRPC in its intended application space. We have also used SpartanRPC to implement a secure real-world WSN application for environmental data collection, demonstrating the effectiveness of the API and of the authorization logic.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

Guido Bertoni, Luca Breveglieri, and Matteo Venturi. 2006. ECC hardware coprocessors for 8-bit systems and power consumption considerations. In *Proceedings of the Third IEEE International Conference on Information Technology: New Generations*. IEEE Computer Society, 573–574.

Antonio Brogi, Răzvan Popescu, Francisco Gutiérrez, Pablo López, and Ernesto Pimentel. 2008. A Service-Oriented Model for Embedded Peer-to-Peer Systems. *Electron. Notes Theor. Comput. Sci.* 194 (April 2008), 5–22. Issue 4.

Richard R. Brooks, Parameswaran Ramanathan, and Akbar M. Sayeed. 2003. Distributed Target Classification and Tracking in Sensor Networks. *Proc. IEEE* 91, 8 (August 2003), 1163–1171.

R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. 1999. Multicast security: a taxonomy and some efficient constructions. In *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99)*, Vol. 2. 708–716.

Seyit A. Çamtepe and Bülent Yener. 2005. *Key Distribution Mechanisms for Wireless Sensor Networks: a Survey*. Technical Report TR-05-07. Rensselaer Polytechnic Institute.

Peter Chapin. 2014. Sprocket Home Page. (December 2014). https://github.com/pchapin/sprocket. Accessed July 2014.

Peter C. Chapin, Christian Skalka, and X. Sean Wang. 2008. Authorization in trust management: Features and foundations. *Comput. Surveys* 40, Article 9 (August 2008), 48 pages. Issue 3.

Chipcon. 2004. CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. Preliminary datasheet rev 1.2. (June 2004).

William R. Claycomb and Dongwan Shin. 2011. A novel node level security policy framework for wireless sensor networks. *J. Netw. Comput. Appl.* 34, 1 (Jan. 2011), 418–428.

Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. 2007. Programming wireless sensor networks with the TeenyLime middleware. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware (Middleware '07)*. Springer, New York, NY, USA, 429–449.

W. Diffie and M. Hellman. 2006. New Directions in Cryptography. *IEEE Trans. Inf. Theor.* 22, 6 (Sept. 2006), 644–654.

C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. 1999. *RFC-2693: SPKI Certificate Theory*. Internet Engineering Task Force.

John G. Fletcher. 1982. An Arithmetic Checksum for Serial Transmissions. *IEEE Transactions on Communications* 30, 1 (jan 1982), 247 – 252.

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. 2009. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Trans. Auton. Adapt. Syst.* 4, Article 16 (July 2009), 26 pages. Issue 3.

Jeffrey Frolik and Christian Skalka. 2013. *Snowcloud*. Technical Report. University of Vermont. Submitted. http://www.cs.uvm.edu/~skalka/skalka-pubs/frolik-skalka-snowcloudtr.pdf.

Saurabh Ganeriwal, Christina Pöpper, Srdjan Čapkun, and Mani B. Srivastava. 2008. Secure Time Synchronization in Sensor Networks. *ACM Trans. Inf. Syst. Secur.* 11, 4, Article 23 (July 2008), 35 pages.

Tia Gao, C. Pesto, L. Selavo, Yin Chen, Jeong G. Ko, Jong H. Lim, A. Terzis, A. Watt, J. Jeng, Bor-Rong Chen, K. Lorincz, and M. Welsh. 2008. Wireless medical sensor networks in emergency response: implementation and pilot results. In *IEEE Conference on Technologies for Homeland Security*. 187–192.

David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 1–11.

Sergio González-Valenzuela, Min Chen, and Victor C. Leung. 2010. Programmable Middleware for Wireless Sensor Networks Applications Using Mobile Agents. *Mob. Netw. Appl.* 15 (December 2010), 853–865. Issue 6.

Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming Wireless Sensor Networks Using Kairos. In *Distributed Computing in Sensor Systems*, Viktor Prasanna, Sitharama Iyengar, Paul Spirakis, and Matt Welsh (Eds.). Lecture Notes in Computer Science, Vol. 3560. Springer, Berlin/Heidelberg, 466–466.

Vipul Gupta, Matthew Millard, Stephen Fung, Yu Zhu, Nils Gura, Hans Eberle, and Sheueling Chang Shantz. 2005. Sizzle: A standards-based end-to-end security architecture for the embedded internet. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PERCOM '05)*. IEEE Computer Society, Washington, DC, USA, 247–256.

Wen Hu, Peter Corke, Wen Chan Shih, and Leslie Overs. 2009. secFleck: A public key technology platform for wireless sensor networks. In *Proceedings of the Sixth European Conference on Wireless Sensor Networks (EWSN '09)*. Springer, Berlin/Heidelberg, 296–311.

Wen Hu, Hailun Tan, Peter Corke, Wen Chan Shih, and Sanjay Jha. 2010. Toward trusted wireless sensor networks. *ACM Trans. Sen. Netw.* 7, Article 5 (August 2010), 25 pages. Issue 1.

Jonathan Hui, Philip Levis, and David Moss. 2008. TinyOS 802.15.4 Frames. (June 2008). http://www.tinyos.net/tinyos-2.x/doc/html/tep125.html. Accessed December 2011.

IEEE. 2003. IEEE std. 802.15.4 - 2003: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). Standard. (October 2003).

C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. 2003. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking* 11, 1 (feb 2003), 2–16.

ISO. 2008. ISO/IEC 1170-3:2008 Information technology – Security techniques – Key management – Part 3: Mechanisms using asymmetric techniques. (2008).

Wooyoung Jung, Sungmin Hong, Minkeun Ha, Young-Joo Kim, and Daeyoung Kim. 2009. SSL-Based Lightweight Security of IP-Based Wireless Sensor Networks. In *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops*, Vol. 0. IEEE Computer Society, 1112–1117.

Chris Karlof, Naveen Sastry, and David Wagner. 2004. TinySec: a link layer security architecture for wireless sensor networks. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys '04)*. ACM, New York, NY, USA, 162–175.

Chris Karlof and David Wagner. 2003. Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures. *Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols* 1, 2–3 (September 2003), 293–315.

Philip Levis. TEP-111: message_t. (????). http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html. Accessed August 2011.

Ninghui Li and Joan Feigenbaum. 2002. Nonmonotonicity, User Interfaces, and Risk Assessment in Certificate Revocation. In *Proceedings of the Fifth International Conference on Financial Cryptography*. Springer-Verlag, London, UK, 166–177.

Ninghui Li and John C. Mitchell. 2003a. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages*. http://www.cs.purdue.edu/homes/ninghui/abstracts/cdatalog_padl03.html

Ninghui Li and John C. Mitchell. 2003b. RT: A Role-based Trust-management Framework. In *Proceedings of the Third DARPA Information Survivability Conference and Exposition*. IEEE Computer Society Press, 201–212. http://www.cs.purdue.edu/homes/ninghui/abstracts/rt_discex03.html

Ninghui Li, John C. Mitchell, and William H. Winsborough. 2002. Design of a Role-based Trust-management Framework. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 114–130. http://www.cs.purdue.edu/homes/ninghui/abstracts/rt_oakland02.html

Ninghui Li, William H. Winsborough, and John C. Mitchell. 2003. Distributed Chain Discovery in Trust Management. *Journal of Computer Security* 11, 1 (Feb 2003), 35–86. http://www.cs.purdue.edu/homes/ninghui/abstracts/discovery_jcs03.html

An Liu and Peng Ning. 2008. TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Proceedings of the Seventh International Conference on Information Processing in Sensor Networks (ISPN '08)*. IEEE Computer Society, Washington, DC, USA, 245–256.

Konrad Lorincz, David J. Malan, Thaddeus R. F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoffrey Mainland, Matt Welsh, and Steve Moulton. 2004. Sensor Networks for Emergency Response: Challenges and Opportunities. *IEEE Pervasive Computing* 3, 4 (2004), 16–23.

Mark Luk, Ghita Mezzour, Adrian Perrig, and Virgil Gligor. 2007. MiniSec: a secure sensor network communication architecture. In *Proceedings of the Sixth International Conference on Information Processing in Sensor Networks (IPSN '07)*. ACM, New York, NY, USA, 479–488.

Geoffrey Mainland, Greg Morrisett, and Matt Welsh. 2008. Flask: staged functional programming for sensor networks. In *Proceeding of the Thirteenth ACM International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 335–346.

Michael Manzo, Tanya Roosta, and Shankar Sastry. 2005. Time synchronization attacks in sensor networks. In *Proceedings of the Third ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN '05)*. ACM, New York, NY, USA, 107–116.

Terry D. May, Shaun H. Dunning, George A. Dowding, and Jason O. Hallstrom. 2007. An RPC Design for Wireless Sensor Networks. *International Journal of Pervasive Computing and Communications* 2, 4 (March 2007), 384–397.

MEMSIC. TelosB Mote Platform. Datasheet. (????). http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf. Accessed January 2014.

C. David Moeser, Mark Walker, Christian Skalka, and Jeff Frolik. 2011. Application of a wireless sensor network for distributed snow water equivalence estimation. In *Proceedings of the Western Snow Conference*.

Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The regiment macroprogramming system. In *Proceedings of the Sixth International Conference on Information Processing in Sensor Networks (IPSN '07)*. ACM, New York, NY, USA, 489–498.

M. Perillo and W. Heinzelman. 2005. *Fundamental Algorithms and Protocols for Wireless and Mobile Networks*. CRC Hall, Chapter Wireless Sensor Network Protocols, 813–842.

Adrian Perrig, John Stankovic, and David Wagner. 2004. Security in Wireless Sensor Networks. *Commun. ACM* 47, 6 (2004), 53–57.

D.R. Raymond and S.F. Midkiff. 2008. Denial-of-Service in Wireless Sensor Networks: Attacks and Defenses. *Pervasive Computing* 7, 1 (jan–march 2008), 74–81.

A. Reinhardt, P.S. Mogre, and R. Steinmetz. 2011. Lightweight remote procedure calls for wireless sensor and actuator networks. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM '11)*. 172–177.

Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *Computer* 29, 2 (1996), 38–47.

R. Srinivasan. 1995. *RFC-1833: Binding Protocols for ONC RPC Version 2*. Internet Engineering Task Force.

TinyOS. TinyOS Community Forum. (????). http://www.tinyos.net/. Accessed February 2012.

Claudio Vairo, Michele Albano, and Stefano Chessa. 2008. A secure middleware for wireless sensor networks. In *Proceedings of the Fifth International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services (Mobiquitous '08)*. ICST, Brussels, Belgium, Article 59, 6 pages.

Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. 2006. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN '06)*. ACM, New York, NY, USA, 416–423.

Received Month Year; revised Month Year; accepted Month Year