Technical Report 06-31, Dept. of Computer Science, Iowa State University, First Version: Oct 12, 2006, Revised: Nov 14, 2006

Lightweight Specification Language and Verification Framework for Sensor Network Security Protocols

Youssef Hanna
Iowa State University
ywhanna@iastate.edu

Hridesh Rajan
Iowa State University
hridesh@iastate.edu

Wensheng Zhang
Iowa State University
wzhang@iastate.edu

Abstract

The contribution of this work is an approach for lightweight specification and verification of nesC implementations of sensor networks security protocols. Our approach provides annotations to specify objectives, network topology, intruder models, and channel fault models. The objectives of the protocols can be specified in terms of user-defined events, which is significantly more expressive compared to earlier approaches such as CAPSL that provide a fixed set of objectives. Moreover, our approach is extensible in that it allows new intruder and channel fault models to be added to the verification process. These models are themselves written in nesC. To show the feasibility of our approach, we describe the implementation of our verification framework. Our verification framework uses the model checker SPIN as the underlying technology. Our approach was able to detect earlier known bugs in protocols and an assumption violation in the protocol implementation.

1. Introduction

Flaws in security protocols are subtle and hard to find [46, 9]. Finding flaws in the security protocols for sensor networks is even harder because they operate under fundamentally different system design assumptions such as event-driven [14] vs. imperative or message passing, resource and bandwidth constraints, hostile deployment scenarios, trivial physical capturing due to the lack of tamper resistance, group-oriented behavior, ad hoc and dynamic topologies, open-ended nature, etc. These assumptions lead to complex protocols, which in turn makes them much harder to verify.

To address this issue, in this work we present a lightweight specification language and a verification framework tailored for sensor network security protocols implemented in the nesC language [14], the dominant language for this paradigm. Our verification framework automatically extracts the PROMELA model [15, 16] of the subject protocol from the source code annotated with specifications. This model is verified against the properties specified in an extended form of Linear Temporal Logic (LTL) formula [41] using the SPIN model checker [15, 16]. Finally, the verification results are mapped back to the implementation domain.

Our approach has two features that allow the user to guide the model generation thereby limiting the state space generated for verification. It allows a user to add a new channel fault model and an intruder model to the verification process. These user-defined intruder models may more closely reflect the attack scenarios specific to the subject protocol thereby reducing the search space significantly. Our approach also allows properties to be specified in terms of *commands* and *events* declared in the protocol, using which a developer can easily model specific properties such as SECRET, PRECEDES, etc, in CAPSL-like specification languages [31].

Outline: Section 2 motivates this work. Section 3 briefly describes the nesC language. A basic familiarity with an imperative programming language such as C is assumed. Section 4 describes the key constructs of our specification language in detail. Section 5 shows example extensions of our verification framework to include more intruders and channel models. Section 6 describes specification language constructs to add non-deterministic behavior to intruder and channel models. Section 7 describes our model extraction methodology and our verification framework. Section 8 describes the verification process. Section 9 discusses the related works. Section 10 describes some limitations of our approach and section 11 concludes.

2. Motivation

Every significant innovation in computer and network systems often introduces new cryptographic failure modes. These failure modes are frequently the result of changed assumptions about the system, its environment, and the client organization. Discovery of the new failure modes fuel the design of new kinds of cryptographic protocols, which in turn leads to the invention and refinement of new verification mechanisms.

For standalone computers, perhaps the only cryptographic failure mode was malicious access to un-authorized information by supplying forged authentication information or physical access to a system. The cryptographic protocols for this era dealt with issues like password encryption, protection of password files during storage and retrieval, etc.

The advent of networking introduced another failure mode, access to un-authorized information on a computer by another computer or principal remotely accessing the computer. As a result, cryptographic protocols based on a central authentication server began to emerge [34]. Massively networked distributed systems and the Internet made the server a bottleneck, resulting in the emergence of decentralized mechanisms such as Needham-Schroeder protocol [35].

2.1 New Failure Modes in Sensor Networks

A *sensor network* is a collection of small size, low power, low-cost sensor nodes that has some computational, communication and storage capacity. These nodes can operate unattended, sensing and recording detailed information about their surroundings. The innovation in wireless networking coupled with the effect of Moore's law is making these networks attractive for many civil and military applications [4] such as target tracking, remote surveillance, and habitat monitoring.

Like their precursors, sensor network systems also introduce new cryptographic failure modes to system design. For example, their lack of tamper resistance makes physical capturing trivial. The low power, communication and storage capacity render intensive cryptographic protocols unusable. The operating environments for sensor networks are often hostile, requiring mechanisms for secure communication. In particular, messages containing missions or queries disseminated by administrators [17], control or data messages for decentralized collaborations, etc, need to be secure.

2.2 New Security Protocols Emerging

The security research community has risen to the challenge to address the new cryptographic failure modes in the sensor networks. A number of security protocols for sensor networks have been proposed in the past decade. Perrig *et al.* [40], Eschenauer and Gligor [13], Liu and Ning [24], and Zhu *et al.* [52] proposed schemes for pairwise key establishment; Perrig *et al.* [40], Ye *et al.* [51], and Yang *et al.* [50] proposed schemes for message authentication; Przydatek *et al.* [42] proposed secure data aggregation scheme.

2.3 Security Flaws Hard to Find

Flaws in security protocols are subtle and very hard to find. In the past, even widely-studied cryptographic protocols are shown to have faults that are detected much later. For example, Meadows [29] showed flaws in selective broadcast protocol by Simmons [46]. Burrows *et al.* [9] showed faults in the directory authentication framework [1]. Simmons [46] showed that replay of messages can be used to trick an authentication protocol.

Finding flaws in the security protocols for sensor networks is even harder. The security protocols for sensor networks protect against more cryptographic failure modes compared to their counterparts. For example, traditional cryptographic protocols assume that the nodes cannot be physically captured; however, capturing sensor nodes is trivial due to the lack of tamper resistance and hostile deployment scenarios. Traditional cryptographic protocols do not have to worry about nodes running out of power, low computational and bandwidth overhead, dynamic and ad hoc topologies, etc. In order to be usable for sensor networks, security protocols do have to consider these constraints. In addition, security objectives are often assigned to groups instead of individual nodes. As a result, these protocols are often more complex making their verification challenging.

2.4 Current Specification Languages Inadequate

Existing specification and verification techniques are inadequate to address these new verification challenges. A major issue is the gap between specification and implementation languages. Most security protocols in sensor networks are designed to work in an event-driven paradigm, whereas existing specification languages are either imperative or use a message passing style. This impedance mismatch between specification and implementation often leads to missing assumptions, which often translates to bugs in protocol implementations. In this work, instead of requiring a separate specification, we extract a model of the protocol from its implementation for verification purposes.

3. The NesC programming Language

NesC [14] is an extension of the C language designed to develop the sensor network applications. NesC applications have three major building blocks: modules, interfaces and configurations. NesC modules are similar to early Ada and ML modules in that they cannot be instantiated, but they serve as containers. A module can contain state declaration (shared by other elements of the modules), command declaration (methods) and event handlers. An event handler is similar to a method; yet, it is executed only when the event is triggered. An interface is a collection of related commands/events. A module that provides an interface has to implement its commands, while a module that uses an interface has to implement its events.

```
module CompM {
  provides interface StdControl;
  uses interface Timer;
}
implementation {
  command result_t StdControl.init() {...}
  event result_t Timer.fired() {...}
}

configuration Comp {
}
Implementation {
  components Main, CompM, SingleTimer;
  Main.StdControl -> CompM.StdControl;
  CompM.Timer -> SingleTimer.Timer;
  ...
}
```

Figure 1. A NesC Example

An example module in NesC is shown in Figure 1. Module `CompM` provides interface `StdControl`, so it has to implement the interface commands (i.e. `StdControl.init()`). `CompM` also uses the interface `Timer`, so it has to implement its events (i.e. `Timer.fired`). A configuration component is responsible for connecting (wiring) the components that are using interfaces to the components that provide their implementation. For example, component `Main` is using interface `StdControl` and is wired to component `CompM` that provides the implementation for `StdControl` comands. Every application has a single top-level configuration.

4. Lightweight Specification Language

Our specification language adds a small set of annotations to be able to guide the verification process. In this section, we will describe these annotations in detail. To describe our specification language, we show the annotated specification of Needham-Schroeder public key protocol [35] written in nesC. The example selected is simple enough to be able to demonstrate the language features without bogging down the reader with details of the protocol. Needham-Schroeder protocol is described in the next subsection.

4.1 Example Protocol: Needham-Schroeder

The objective of this protocol is to exchange two secret numbers between two principals. Figure 2 shows the sequence of messages in this protocol. Principal *A* encrypts its randomly generated secret number called Nonce *Na* and its name using the public key of *B*, so only *B* can decrypt it using its own private key. After receiving and decrypting the message, principal *B* encrypts its Nonce *Nb* and its partner nonce *Na* using *A*'s public key and sends it back to *A*. Principal *A* decrypts the message, receives the partner's nonce, encrypts it and sends it back to *B*. The two principals now share the nonces *Na* and *Nb*.

```
Msg1. A -> B : {Na,A}pubkB
Msg2. B -> A : {Na,Nb}pubkA
Msg3. A -> B : {Nb} pubkB
```

Figure 2. Needham-Schroeder Protocol

4.2 Topology

The first task is to decide upon the network topology for protocol verification. Our language provides annotations for this task. The network topology could also be specified as a command line option for the framework, or as a configuration file; however, neither of these options make this knowledge explicit as part of the protocol implementation itself.

```
/*@
* __nodes__: 2;
*
* __reachable__:0 <-> 1;
*
@*/
configuration Needham {
...
}
```

Figure 3. Test Topology for Needham-Schroeder Protocol

Figure 3 shows an example topology specification. The annotation comments start and end with an at-sign (@) similar to the annotation comments for the JML specification [22]. The number of nodes involved in the protocol is specified by the special word `__nodes__` (Line 2). These special words are not keywords; they only have special meanings in this context. In this example, the network is composed of 2 nodes. Next, the network connectivity in the topology is specified by the special word `__reachable__` (Line 4). The network topology in this example specifies the bi-directional connection between node 0 and node 1. An alternative uni-directional topology could also be 0 -> 1, which would mean that 1 is reachable from 0 but not the other way around.

For ease of topology specification, we provide some syntactic sugars, allow use of wildcards, and provide facilities to group a number of nodes. For example, 0 –> * would mean that any node in the network is reachable from the node 0 and 0 –> {2,5,8} would mean that nodes 2, 5 and 8 are reachable from the node 0. Such topologies are commonly known as the *star*. To show a communication path in the network, a group of nodes can also be chained. For example, 0 –> 1 –> 2 would mean that node 1 is reachable from node 0, node 2 is reachable from node 1. The ring (or circular) topologies such as 0 –> 1 –> 2 –> 0 are also allowed. The concrete syntax of topologies will be discussed later in this section.

To start the verification process, topology specification is sufficient. Figure 4 shows the resulting trace of the verification process, where the input to the verification framework was the implementation of the Needham-Schroeder protocol annotated with the topology information shown in Figure 3. The figure shows the successful execution of the protocol implementation and the resulting message exchange between the principals. To keep the verification state space minimal, we recommend starting with a small number of nodes and gradually scaling up the verification process.

4.3 Channel Specification in Topologies

The topology specification in Figure 3 although instructive from a pedagogical perspective is not very interesting from the verification perspective until faults are introduced in the topology. Simplest types of faults in a network configuration are due to unreliable communication mediums (channels).

Figure 5 introduces a simple faulty channel `Sink` between node 0 and node 1. This topology can be read as node 1 is reachable from node 0 via channel `Sink` and vice-versa. As the name implies, this channel simply drops all messages from the recipients. Other channel models are also defined. Moreover, as we will discuss later users may themselves define a new channel type for use in their specifications. The compiler throws a syntactic error, if an

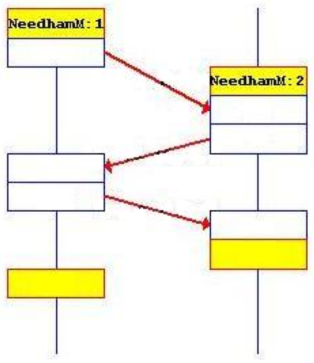

Figure 4. Trace Generated after Adding Topology. (Details elided)

```
/*@
* __nodes__: 2;
*
* __reachable__:0 <-> 1/Sink;
*
@*/
configuration Needham {
...
}
```

Figure 5. Introducing Faulty Channels in a Test Topology

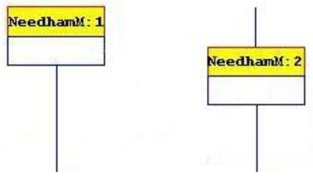

Figure 6. Test Trace Generated after Topology Modification: shows that protocol cannot make progress in this case.(Details elided)

undefined channel model is used. After modifying the topology, the user can run the verification process again to get the resulting trace as shown in Figure 6, which shows that the protocol cannot make progress in this case.

4.4 Intruder Specification in Topologies

Another component of the topology specification is the description of intruder models, sometimes also called attack models. Three types of intruder models can be defined: promiscuous intruders, impersonators, and insiders. A promiscuous intruder is not visible to the legitimate nodes in the network. They silently gather private data by listening to communication between other nodes. In a wireless network such as sensor networks, promiscuous intruder-based attacks are prevalent. An impersonator, also called man-in-the-middle attack, pretends to be a legitimate node in the network to communicate with other nodes with the overall objective to gather secret. An insider, as the name implies, is a legitimate participant of the network that is compromised to be hostile. The small form factor and the deployment scenarios for sensor networks make such attacks easier in comparison to traditional systems.

The presence of a promiscuous intruder in the topology is specified in the same way as the faulty channel. For example, a simple predefined promiscuous intruder type is `Store`. As the name suggests, this intruder keeps a log of the messages transmitted between principals for offline or online analysis to derive attack patterns and for replay attacks. The channel model `Sink` in Figure 5 can be replaced to insert this intruder into the topology in just a simple edit to change line 4 to `__reachable__:0 <-> 1/Store`.

```
1  /*@
2  * __nodes__: 4;
3  *
4  * __reachable__:0 <-> {1,4};
5  *
6  * __intruder__: 0 <-2-> 1/ManInMiddle;
7  *
8  * __intruder__: 4/DenialOfService;
9  *
10 @*/
11 configuration Needham {
12 ...
13 }
```

Figure 7. Introducing Impersonators and Insiders

An impersonator can be included in the system using another visually intuitive notation as shown on line 6 in Figure 7. The notation shows that the node 2 is impersonating node 0 to communicate with node 1. The type of impersonator is specified to be a pre-defined intruder type `ManInMiddle`.

An insider is a legitimate node in the network that is compromised. It is therefore essential to specify connectivity to the insider intruder. In Figure 7, line 4 shows that node 0 is connected with node 1 and node 4. Furthermore, line 8 shows that node 4 is compromised to emulate a pre-defined intruder type `DenialOfService`.

Similar to channel types, users may themselves define new intruder types for use in their verification process and the use of an undefined intruder type is illegal. The concrete BNF grammar for topologies is shown in Figure 8. The productions in the grammar are self-explanatory.

4.5 Objectives

Figure 9 shows the syntax of objectives in our approach. An objective specification starts with the special word __objective__ followed by a colon followed by an objective expression. An expression can be a literal, or a negated expression (! expression), enclosed in parenthesis, or an expression preceded by [] or <>, which denotes the temporal conditions *always* and *eventually*. An expression may also consist of two sub-expressions combined by logical and (&&), logical or (||), implication (− >), or equivalence (< − >) operators. These operators have similar meanings as their LTL counterparts and they are translated accordingly. Finally, a literal can be true, false or a user-defined condition.

The user-defined conditions are especially interesting as they allow objectives to be expressed in terms of commands and events in the protocol implementation. For example, assume that the sender `SensorS` in a given protocol declares a command `snd` that takes a parameter of type `TOS_Msg` and the receiver `SensorR` in the same protocol registers an event handler of type `rcvd` with the underlying network layer that also receives a parameter of type `TOS_Msg`. The type `TOS_Msg` is a standard structure for sending and receiving messages available to NesC programs. A trivial property of this implementation can be expressed as the user-defined condition `SensorS.snd (TOS_Msg m)-> <>SensorR.rcvdMsg(TOS_Msg m)`, which means that `SensorS` sending a message `m` implies that `SensorR` will receive the same message `m` eventually. This objective does not account for problems in message transmission, but it can be easily augmented to do so.

Let us now consider the objectives of the Needham-Schroeder protocol. The primary objective of this protocol is that two nodes

$\langle topology \rangle$	::=	$\langle numNodes \rangle \langle opt-reachability \rangle$
		$\langle opt-intruder \rangle$
$\langle numNodes \rangle$	::=	**__nodes__** : $\langle posint \rangle$
$\langle reachability \rangle$	::=	**__reachable__** : $\langle rexpressions \rangle$
$\langle rexpressions \rangle$	::=	$\langle rexpressions \rangle, \langle rexpression \rangle$
		$\langle rexpression \rangle$
$\langle rexpression \rangle$	::=	$\langle posint \rangle \langle rop \rangle \langle posint \rangle \langle opt-chanModel \rangle$
		$\langle posint \rangle \langle rop \rangle \langle posint \rangle \langle opt-intrModel \rangle$
$\langle rop \rangle$	::=	$<-->$
		$-->$
		$<--$
		$<-\langle posint \rangle->$
		$-\langle posint \rangle->$
		$<-\langle posint \rangle-$
$\langle chanModel \rangle$	::=	/c, c $\in$ **Set of defined channel models**
$\langle intrModel \rangle$	::=	/in, in $\in$ **Set of defined intruder models**
$\langle intruder \rangle$	::=	**__intruder__** : $\langle iexpressions \rangle$
$\langle iexpressions \rangle$	::=	$\langle iexpressions \rangle, \langle iexpression \rangle$
		$\langle iexpression \rangle$
$\langle iexpression \rangle$	::=	$\langle posint \rangle$ / $\langle intrModel \rangle$
$\langle posint \rangle$	::=	p, p $\in$ **Set of positive integers**

Figure 8. Syntax of Topologies

$\langle objective \rangle$	::=	**__objective__** : $\langle oexpression \rangle$
$\langle oexpression \rangle$	::=	$\langle literal \rangle$
		| **!** $\langle oexpression \rangle$
		| **(** $\langle oexpression \rangle$ **)**
		| **[]** $\langle oexpression \rangle$
		| **<>** $\langle oexpression \rangle$
		| $\langle oexpression \rangle$ **&&** $\langle oexpression \rangle$
		| $\langle oexpression \rangle$ || $\langle oexpression \rangle$
		| $\langle oexpression \rangle -> \langle oexpression \rangle$
		| $\langle oexpression \rangle <-> \langle oexpression \rangle$
$\langle literal \rangle$	::=	**true** | **false** | $\langle condition \rangle$
$\langle condition \rangle$	::=	$\langle int \rangle$. $\langle qid \rangle$ **(**$\langle params \rangle$ **)**
$\langle qid \rangle$	::=	$\langle qid \rangle$. $\langle id \rangle$
		$\langle id \rangle$
$\langle params \rangle$	::=	$\langle params \rangle, \langle param \rangle$
		$\langle param \rangle$
$\langle param \rangle$	::=	$\langle type \rangle \langle id \rangle$

Figure 9. Syntax of Objectives

```
1  /*@
2  * __nodes__: 3;
3  *
4  * __reachable__:0 <-> 1;
5  *
6  * __intruder__: 0 <-2-> 1/ManInMiddle;
7  *
8  * __objective__: (0.finalState(Nonce n)
9  *                 ->1.finalState(Nonce n))
10 *                 && !(2.knows(Nonce n));
11 *
12 @*/
13 configuration Needham {
14 ...
15 }
```

Figure 10. Introducing Impersonators and Insiders

should share a common nonce for communication. The security objective of this protocol is that any other node (including intruders) in the system may not know this nonce. If the protocol implementation is slightly modified to throw an event `finalState` with a parameter `n` of type `Nonce`, when the communicating principals believe that the final state is reached, these objectives can be expressed in terms of these user-defined events as shown in Figure 10 (lines 8-10). Here the intruder type throws an event `knows` with

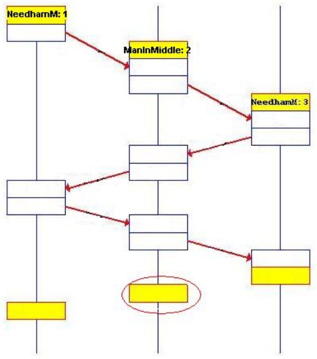

Figure 11. Test Trace Generated after Adding Objectives: Circled State Denotes Objective Violation. (Details elided)

a parameter of type `Nonce`, if it is able to guess a `Nonce`. Using the same parameter name `n` in all three parts of the objective expresses the intent that all three nonce are the same. After adding this objective, the user can run the verification process again to get the resulting trace as shown in Figure 11, which shows that the protocol violates the objective and the message sequence that leads to the violation.

5. Extending the Verification Framework

In this section, we discuss two extension mechanisms provided by our verification framework for including user-defined channel fault models and intruder models.

5.1 Adding New Channel Fault Models

Our verification framework provides a set of primitives to add new channel models. Channel models are used to specify the topology of the network. They are written as separate nesC applications and saved in a library. Figure 12 shows an example of a lossy channel model `Sink`. The module `Sink` has one state that is annotated with our new annotation comments. The special word `__intercepted__` declares that this state of type `TOS_Msg` (structure representing message in *nesC* applications) is the message being sent through the channel. In other words, it is a hint to the verification framework that when this channel model is used, the variable `message` should be initialized with the message being transmitted. The rest of the module simply does nothing, which means that the channel does not forward the message. Note that the channel application is meaningless if treated as an independent application.

```
module SinkM {...}
implementation {
/*@ __intercepted__ @*/
 struct TOS_Msg message;
 ...
}
```

Figure 12. An Example Channel Model

More intelligent channel fault models can be written that may modify the message during transmission arbitrarily. For example, Figure 13 implements a channel that drops half of the transmitted messages. The command `start` in this module maintains an alternating flag and forwards the message when the flag is set to 1 and drops it otherwise.

```
module LossyChannel {...}
implementation {
/*@ __intercepted__ @*/
struct TOS_Msg _msg;
int flag;
command result_t StdControl.init() { flag = 0; ...}
command result_t StdControl.start() {
        IntMsg* msg = (IntMsg *) _msg.data;
        call CommControl.start();
        if (flag == 1) { /* Forward the message */
        call Send.send(msg->dest, sizeof(IntMsg), &_msg);
        flag = 0;
        }
        else flag = 1; /* Drop the message */
        return SUCCESS;
}
 ...
}
```

Figure 13. Emulating 50% probability of Message Loss

5.2 Adding New Intruder Models

Similar to channel models, intruder models are written as separate nesC applications and can be saved in a library for reuse. Three different intruder types are modeled differently. The promiscuous intruder type is modeled similar to the channel models, but it always forwards the messages it receives. An example of the impersonator intruder type is shown in Figure 14. In the declaration of states, the special word `__role__` (line 10) specifies that when the intruder is used by the verification framework, the state `AName` (line 11) should be initialized with the name of the impersonated node. Similarly, `__target__` (line 12) specifies that the state `BName` (line 13) is the node to communicate with. `__name__` (line 14) specifies that `intruderName` is the name of the intruder node. The event handler `ReceiveMsg.Receive` is responsible for receiving messages.

```
1 module ManInMiddle {...}
2 implementation {
3 /*@ __role__ @*/
4 int AName;

5 /*@ __target__ @*/
6 int BName;

7 /*@ __name__ @*/
8 int intruderName;
9 ...
10 event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msgR) {...}
11 }
```

Figure 14. Example Intruder: ManInMiddle

In Figure 7, this intruder model was used as `__intruder__`: $0 <-2-> 1$`/ManInMiddle`. In this context, `AName` will be equal to 0, `BName` will be equal to 1, and `intruderName` will be equal to 2.

6. Specifying Non-Determinism

In order to expose errors in security protocols, one should be able to generate and analyze various possible combinations (models) of interactions between the intruder nodes, channels, and genuine nodes running the protocol. However, additional mechanisms should be provided to keep the state space of these models tractable. Our verification framework utilizes state-based model checking as the underlying technology. State space explosion is a well-known problem in a state-based model checking techniques such as SPIN. Although SPIN uses abstraction and partial-order reduction techniques [39] to mitigate the effects of state space explosion, scalability is still an issue for complex models, if care is not exercised in constructing them.

6.1 Message Templates

Our approach provides constructs to specify limited non-determinism in intruder and channel models. The first such construct is *message template*. A message template is an annotated structure used to specify range of values taken by the members of the structure. An example message template is shown in Figure 15. In this message template, the members of the structure `IntMsg` are annotated with their respective ranges. For example, let us assume a verification scenario where the network topology consists of four nodes and impersonator's node id is 4. In this scenario, we can safely assume that all messages that an impersonator might generate will always have the source and destination address between 0 and 3. This restriction is specified by annotating the structure member `src` and `dest` using the special word `__range__`.

```
1 typedef struct IntMsg {
2   ...
3   /*@ __range__: 0-3 @*/
4   int src;

5   /*@ __range__: 0-3 @*/
6   int dest;
7   ...
8 } IntMsg;
```

Figure 15. Declaring Message Templates

Now, let us suppose that we want to add an intruder model that produces a random response to any message that it receives, hoping to find and exploit a flaw in the subject protocol. Such an intruder is shown in Figure 16. In this example, line 5 shows that the local variable `msg` is annotated with the special word `__usetemplate__`. When this intruder model is translated to the PROMELA code, our verifying compiler automatically expands it to try all possible messages that conform to the message template non-deterministically. We advise to use message templates with caution. If message templates are not designed carefully, the state-space of the model can quickly become unwieldy.

```
1 event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msgR) {
2       /@ __usetemplate__ @/
3         IntMsg msg;
4         call Send.send(msg.dest, sizeof(IntMsg), msg);
5     }
```

Figure 16. Using Message Templates

6.2 Parameterized Message Templates

In practice, intruders may use a variety of message templates in different situations. Our message templates can also be parameterized to suit that need. An example parameterized message template is shown in Figure 17. The parameterized message template is declared by annotating the structure definition as shown on line 1. The syntax is inspired from the common syntax for generics. A parameterized message template can have zero or more parameters (message template being the special case, when the number of parameters is zero). The parameter `selector` here is used to select appropriate ranges for the structure member `src` in a form similar to switch-case statements. The label default is optional. An example usage of the parameterized message templates is shown in Figure 18 that selects values in two different ranges for `src`.

```
1 typedef struct IntMsg /*@ <selector> @*/{
2   ...
3   /*@ case (selector == 1) __range__: 2-3;
4    *  case (selector == 2) __range__: 1,3;
5    *  default __range__: 1-2              @*/
6   int src;
7   ...
8 } IntMsg;
```

Figure 17. Declaring Parameterized Message Template

```
1 event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msgR) {
2       /@ __usetemplate__: 1 @/
3         IntMsg msg;
4         call Send.send(msg.dest, sizeof(IntMsg), msg);
5     }
```

Figure 18. Using Parameterized Message Templates

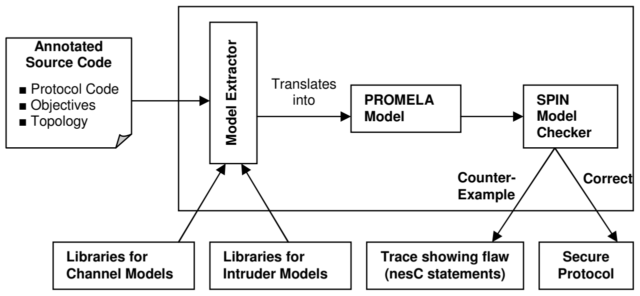

Figure 19. Overview of the Verification Framework

6.3 Non-Deterministic Selection of Commands and Events

The channel fault model and intruder model implementations are allowed to declare one or more commands and event handlers with same signature. When there are multiple commands (event handlers) with exactly same signature, a non-deterministic choice is made to pick one command (event handler) for execution.

7. Verification Framework

Figure 19 shows the components of our framework. The annotated implementation, the intruder models and the channel models are given as inputs to the model extractor. The model extractor generates a PROMELA model from the input files. The model is then given as input to the SPIN model checker, which verifies whether the model violates the objectives which have been translated into LTL formulas. If the objectives are satisfied, the protocol is verified as secure. Otherwise, SPIN produces a counter example that violates the security objectives. This counter example is then translated to a sequence of nesC statements. The protocol verification may not terminate if the PROMELA model is too large. We describe construction of PROMELA model from protocol implementation, channel models, intruder models and objectives below.

7.1 Translation of Protocols

A protocol as a whole is translated into one PROMELA process that is a global object describing the behaviors of the protocol. For each principal (node) involved in the protocol, an instance of the process is instantiated and run.

Events are translated into synchronous global message channels. When an event is signaled, a message is placed on the corresponding channel of the corresponding node. Event handlers are translated into loops having as the guard statement a receive message statement that listens to the channel. This statement is not executable unless some item is placed on the channel, so the code for the event handlers is not executed unless the event is signaled.

7.2 Translation of Channel Models

If a channel is normal, the code for transmitting messages via the channel will be translated into PROMELA. However, if the channel is labeled as some faulty model (e.g., Sink, ManInMiddle, etc.), the code for describing the behaviors of the corresponding faulty model will be translated. For the example shown in Figure 5. The channel connecting node 0 and node 1 is a sink. Therefore, when that channel is translated, no code will be generated since a sink channel does not forward any message.

7.2.1 Translation of Intruder Models

Each intruder model is translated into a separate process that describes the behaviors of the intruder according to the nesC code for the intruder model. The translation is performed in the same way as for a normal protocol.

7.2.2 Translation of Objectives

Since objectives are represented using command calls/ event signals, an array of type `bool` and of size n is initialized to false for every command/event, where n is the number of nodes involved in the protocol. When a node calls a command, the element in the corresponding array is set to true. As for the objective itself, it is translated into a linear temporal logic (LTL) formula.

7.2.3 Model Checking

Once translated into PROMELA model, the protocol is verified against the objectives using the SPIN model checker. If the objectives are not satisfied, SPIN produces a counterexample that violates the objective. This counterexample is then translated into a sequence of statements in the protocol that violated the security objective.

8. Evaluation

In this section, we describe the results of applying our approach on several protocol. For sanity check, we first applied our verification process to the Needham-Schroeder protocol. Even though, this protocol is not designed for sensor networks, it provided a good initial test case.

8.1 Verification of Needham-Schroeder Protocol

Needham-Schroeder protocol has a known flaw. If a malicious node I communicating with node A impersonates A and establish a connection with another node B, B will believe that it is communicating with A and will share its secret key with A. The malicious node I will become aware of this secret key. This violates the protocol objective the secret numbers should be only known to the two nodes.

We verified a nesC implementation of the Needham-Schroeder protocol. Our framework successfully detected and reported the flaw in the protocol. Figure 20 shows the counterexample that violates the protocol objectives in terms of sequence of statements of the protocol. Lines 1, 2 and 3 show that there are three nodes in the network and an intruder node. Line 9 shows that the secret number of node 2 (node B) is revealed to the intruder. The verification of this protocol requires around 7 MB memory, generates around 700K states out of which around 317K states are matched by SPIN. The depth of the generated state space is around 9000. The time taken to verify this protocol is less than a second on a Dell PowerEdge 1850 with dual 3.8 GHz processors and 2 GB RAM.

```
1 NeedhamM.nc.line 15: call Sensor.init()
2   NeedhamM.nc.line 15: call Sensor.init()
3     NeedhamM.nc.line 15: call Sensor.init()
4 SensorM.nc.line 64: call Send.send(...)
5       Intruder1M.nc.line 54: ...
6 SensorM.nc.line 124: call Send.send(...)
7       Intruder1M.nc.line 75: ...

8       // Nonce of node B revealed to intruder
9       Intruder1M.nc.line 76: NonceB = ...
```

Figure 20. Flaw Detection in Needham-Schroeder Protocol

8.2 Verification of the One-way Key Chain Based One-hop Broadcast Authentication Scheme

The one-way key chain based one-hop broadcast authentication scheme was proposed by Zhu et al. [52]. In this scheme, every node (denoted as A) generates a one-way key chain of certain length; that is, k_n, $k_{n-1} = h(k_n)$, $\cdots$, $k_1 = h^{n-1}(k_n)$, $k_0 = h^n(k_n)$, where $h(.)$ is a secure hash function. Then, A transmits the first key of the key chain (i.e., k_0) to each neighbor separately, encrypted with the pairwise key shared between A and this neighbor. When A broadcasts its first message m_0, the message is authenticated with k_1; that is, m_0 is broadcast with message authentication code (MAC) $h(m_0, k_1)$. After the broadcast, k_1 is released alone or with the next broadcast message, which is authenticated with the next key in the key chain (i.e., k_2). To generalize, the i-th message m_i is broadcast along with $h(m_i, k_{i+1})$, and k_{i+1} is released after the broadcast.

One known attack [52] to the above scheme is as follows: First, the adversary prevents a neighbor of A (denoted as B) from receiving the packet from A directly. This can be achieved by, for example, transmitting to B at the same time when A is transmitting message m_i and when A is releasing authentication key k_{i+1}. Second, the adversary sends a modified packet to B while impersonating A. Note that, the adversary has already got the released authentication key before transmitting the modified message to B, hence B will not detect the fabrication and will accept the modified packet. To defend against an outsider (not a neighbor of A) from launching the above attack, the original authentication scheme can be enhanced as follows: A shares a cluster key KC with all its neighbors; when A broadcasts message m_i, the MAC of the message will be $h(m_i, k_{i+1}XORKC)$. However, the defense will not be useful if the adversary has compromised A and therefore has obtained KC [52]. Our approach was able to detect this attack.

8.3 Verification of the Polynomial Based Pairwise Key Establishment Protocol

The polynomial based pairwise key establishment protocol [24] includes two phases: system initialization before network deployment and pairwise key establishment after deployment. Before deployment, the network controller picks n symmetric bivariate polynomials $f_i(x, y)$ $(i = 0, \cdots, n-1)$; every sensor node with ID A is preloaded with $m < n$ univariate polynomials $f_{i_k}(A, y)$ $(k = 0, \cdots, m-1)$, which are shares of m out of n aforementioned bivariate polynomials. After deployment, if neighboring nodes A and B have shares derived from the same bivariate polynomial, for example, $f_0(A, y)$ and $f_0(B, y)$, they can directly establish $f_i(A, B) = f_i(B, A)$ as their pairwise key. Otherwise, A and B will find one or more helping nodes I_1, I_2, $\cdots$, I_s such that, each pair of adjacent nodes on the chain A, I_1, I_2, $\cdots$, I_s have shares derived from the same bivariate polynomial, and thus can set up a pairwise key. Then, A picks a new key, encrypts it with the pairwise key shared with I_1, and sends it to I_1. I_1 and the following nodes in the chain uses the same approach to secretly transmit the new key hop-by-hop towards B. This way, a pairwise key can be estab-

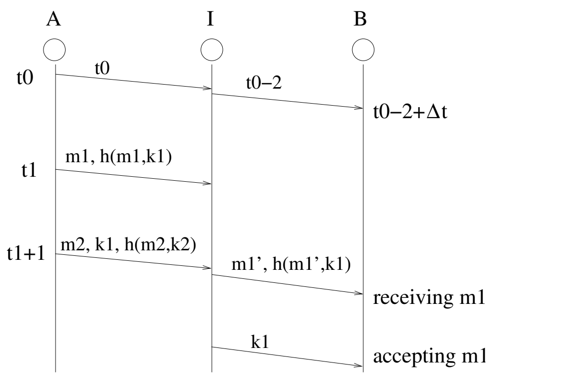

Figure 21. Assumption Violation in $\mu TESLA$ Implementation

lished between A and B. However, the pairwise key between A and B is not exclusively shared by A and B, but is exposed to every intermediate nodes ($I_0, \cdots, I_s$) in the chain. This leads to a flaw of the protocol: if any of the intermediate nodes is compromised, the pairwise key between A and B will be disclosed.

We wrote a simplified version of the protocol, where every node is initialized with two bivariate polynomials (i.e. node A has $f_1(A, y)$ and $f_2(A, y)$, node B has $f_2(B, y)$ and $f_3(B, y)$, and node C has $f_3(C, y)$ and $f_4(C, y)$) so node A can establish a key directly with node B using $f_2(A, B)$, node B and establish a key directly with node C using $f_3(B, C)$. However, node A cannot establish a key directly with node C (no shared bivariate polynomial), thus it has to go through the intermediate node B.

The intruder model for intermediate nodes signals the event `Intruder.knowSecret(secretOfAandB)` when it sends the secret code to the two nodes that want to establish a pairwise key.

8.4 Verification of the $\mu TESLA$ protocol

$\mu TESLA$ [40] was proposed for securing broadcast in sensor networks. This protocol assumes a network model that consists of a broadcast sender (e.g., base station) and multiple receivers (e.g., ordinary sensor nodes). On receiving a broadcast message, each receiver needs to verify whether the message is really from the sender and not tampered by any intermediate nodes. The correct working of the protocol relies on the assumption that all principals (base station and ordinary sensor nodes) are loosely time synchronized. However, the description of the protocol does not specify what time synchronization protocol should be applied or what properties the time synchronization protocol should have. The lack of rigorous treatment of the fundamental assumption may not pose a problem to an network security expert, but it may lead to unexpected mis-implementation if the implementer does not fully understand the protocol. For example, the implementer may choose to implement a simple but not secure time synchronization protocol as the fundation of the $\mu TESLA$ protocol since the $\mu TESLA$ does not have clear specification for the time synchronization protocol.

Figure 21 illustrates the counter example trace. For time synchronization, node A sends out its time stamp t0, which is intercepted by some malicious node I. Node I changes the time stamp to be t0, and then forwards it to nodes B. Since the time synchronization protocol is attacked, the clock in node B will not get synchronized with node A. Later, when node A broadcasts message m1 (which is authenticated with key k1) at time t1, the message is intercepted by node I who will not further forward it. At time t1+1, when node A releases key k1, the key is also intercepted and held by node I. Right after that, node I forges a message m1' authenticated with key k1, and forwards it to B. Then, node I releases key k1 at t1+2. Upon receiving k1, node B will accept message m1' since it can be verified with k1 and the time stamp of the message (i.e., t1) is within the valid scope for acceptance.

9. Related Work

9.1 Specification Languages

Specifications for security protocols range from informal narrations of message flows to formal assertion of protocol properties [2]. The use of natural language to describe the protocol is easy, but lacks rigor. It is also difficult to use such specifications as the basis for reasoning. Other specification techniques such as that proposed by Needham and Schroeder [35] address some of these limitations (also see Liebl [23]). In their notation, the protocol is expressed as a set of messages exchanged between principals. Abadi and Gordon [2] argue that "these notations have a fairly clear connection to the intended implementations of the protocols, but they do not provide a precise and solid basis for reasoning about the protocols." They further state that "other notations such as that by Burrows, Abadi and Needham [9] are more formal, but their relation to implementations may be more tenuous or subtle." There thus seems to be a tradeoff between rigor and intuitiveness in the notations for protocol specification. Our approach is different from these approaches in that it is concerned about verifying the implementations of these protocols and the specifications that are needed are minimal.

Another challenge in specification is to denote security properties. A widely used approach is to formulate the security properties as predicates on the behaviors of the system consisting of a protocol and its environment (e.g. Bellare and Rogaway [6], Bodei *et al.* [7], Gray and McLean [18], Mitchell *et al.* [33], Lowe [26], Paulson [38], Schneider [44], Woo and Lam [49]). These approaches are loosely based on Dolev and Yao's notion of secrecy [12] which states that *a process preserves the secrecy of a piece of data M if the process never sends M in clear on the network, or anything that would permit the computation of M, even in interaction with an attacker.* However, based on the observations of McLean [27], Abadi and Gordon [2] argue that some security properties, such as non-interference, are not predicates on behaviors.

Language-based approaches generally allow richer specification. A number of language-based approaches have also been used for specification such as Real Time Asynchronous Grammar (RTAG) [5] based on attribute grammars [21], PRO-GRAM [37] based on YACC [19], Promela [15] [16] based on Linear Temporal Logic (LTL) [41], etc. As far we are aware, these approaches do not provide a generic mechanism to verify protocol properties, such as security properties. Our approach is essentially to build a domain-specific mechanism based on these language-based approaches.

The common authentication protocol specification language (CAPSL) developed by Millen et al. [31], is closely related. The motivation for the CAPSL project was that it is difficult to apply most cryptographic protocol verification mechanisms. They argued that the reason for this difficulty is that a protocol has to be re-specified for each verification technique that is applied to it and translating published description to the input of the verification tool is difficult [11]. CAPSL project solves this problem by developing a two-layered language design, where higher-level specification is translated to the CAPSL intermediate language (CIL). CAPSL allows clear specification of security properties in the style of Dolev and Yao [12]; however, it is also a message driven specification language that does not fit the sensor network paradigm very well.

9.2 Verification Techniques

There is a significant body of research on verifying security protocols but they don't address challenges of sensor networks security protcols. The best-known and influential approach based on Modal logic is that by Burrows, Abadi and Needham [9, 3], commonly known as the BAN logic. The key idea is to reason about the state of beliefs among principals in a system. Some extensions to the BAN logic are also proposed such as by Oorschot [48].

Meadows developed the NRL protocol analyzer for the analysis of cryptographic protocols [29]. The NRL protocol analyzer was used to find flaws in a number of cryptographic protocols including selective broadcast protocol by Simmons [46], Resource Sharing Protocol by Burns and Mitchell [8], re-authentication protocol by Neuman and Stubblebine [36], etc. Longley and Rigby also developed a tool and demonstrated a flaw in a banking security protocol [25]. Yet another tool was Interrogater developed by Millen *et al.* [32]. Kemmerer [20] used general-purpose formal methods technique as tools to verify cryptographic protocols. Schneider adapted the CSP model for verification of security protocols [45]. For a detailed summary of verification techniques, please refer to a survey by Rubin and Honeyman [43], Meadows [28], Gritzalis *et al.* [47], and a more recent survey by Buttyan [10].

10. Limitations

The current implementation of our verification framework has some limitations partly due to the restrictions of the underlying model-checking technology and due to specific translation approach that we have taken. A limitation is on the number of participant nodes in the verification process. Most implementations of SPIN only allow a maximum of 255 channels for rendezvous communication to keep the model finite. Our approach uses channels to model events and communication between two nodes. The number of channels thus puts a limitation on the number of nodes available to the protocol verifier. In future, we plan to eliminate this limitation by using guarded statements that allow more than one nodes to share a channel. The second limitation stems from our approach to model nodes. We model a node as a separate process in PROMELA. The number of these processes are limited.

In a recent work, Meadows [30] argues that "most of the work on the formal analysis of cryptographic protocols has concentrated on protocols that involve the communication of a fixed number of principals: $\cdots$ Most data structures are close-ended. $\cdots$ The open-endedness is included in the protocol model, but only with respect to the number of protocol executions that may be going on at the same time, $\cdots$. However, open-ended structures are beginning to show up in a number of different applications. By open-ended, we simply mean that the structure may include an arbitrary large number of data fields; either no precise limit is put on them, or the bound is so large that for the purpose of analysis we may as well assume that it does not exist. One example of an open-ended structure is in group communication protocols, in which keys must be shared among the members of a group of arbitrary size. [30, p. 4–5]"

Most security protocols in sensor networks are open-ended in the Meadows terminology. In other words they place no bounds on the principals involved. In the protocol for filtering false messages proposed by Ye *et al.* [51], the number of nodes participating in the source authentication is not limited. In general, in the sensor networks paradigm, the key concern is the group behavior of the nodes. The objectives are usually assigned to a group of nodes, based on locality, type, etc. The behavior of individual nodes is only as important as to satisfy the group objectives. The individual nodes may join or leave the group at times: they may be captured, or may just run out of power. It is therefore necessary to address this limitation.

11. Conclusion and Future Directions

In this paper, we presented our approach for writing lightweight specifications and automatic verification tailored for sensor network security protocols. Our approach automatically extracts PROMELA models from nesC implementation of these security protocols. Annotations are provided to specify network topology, intruders, faulty channels and protocol objectives. The verification framework can be extended to include new intruder and channel models. Our approach also helps fill the impedance mismatch between message-based specification languages and event-based implementation languages of sensor networks.

Our approach opens up a number of interesting avenues that we plan to explore in future. One such area is analyzing the influence of non-functional properties, such as memory, bandwidth, and power constraint on security properties. Sensor nodes are resource and bandwidth constrained. It may not be sufficient in this environment for a node to have excellent security property at the cost of depleting the system resources. The fitness of a protocol for a particular purpose is thus also a function of assumptions about the execution environment. For example, a key management protocol may distribute the shares of a key polynomial among n neighbors so that k fragments are required to reconstruct it. This protocol fails if either $l \geq k$ nodes are captured or $m \geq n - k$ nodes run out of power. Traditional verification mechanisms only assume lost or intercepted messages as failure modes for security protocols making them inadequate to handle situations like the loss of power situation above and the effect of other such non-functional properties on security properties.

References

[1] CCITT draft recommendation X.509. The directory-authentication framework, Version 7, Nov 1987.

[2] Martin Abadi. Security protocols and specifications. In *FoSSaCS '99: Proceedings of the Second International Conference on Foundations of Software Science and Computation Structure, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 1–13, London, UK, 1999. Springer-Verlag.

[3] Martin Abadi and Mark R. Tuttle. A semantics for a logic of authentication (extended abstract). In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 201–216, New York, NY, USA, 1991. ACM Press.

[4] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E.Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4), March 2002.

[5] David P. Anderson and Lawrence H. Landweber. A grammar-based methodology for protocol specification and implementation. *SIGCOMM Comput. Commun. Rev.*, 15(4):63–70, 1985.

[6] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *CRYPTO '93: Proceedings of the 13th annual international cryptology conference on Advances in cryptology*, pages 232–249, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[7] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *CONCUR '98: Proceedings of the 9th International Conference on Concurrency Theory*, pages 84–98, London, UK, 1998. Springer-Verlag.

[8] John Burns and Chris J. Mitchell. A security scheme for resource sharing over a network. *Comput. Secur.*, 9(1):67–75, 1990.

[9] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[10] Levente Buttyan. Formal methods in the design of cryptographic protocols (state of the art). Technical Report SSC/1999/38, Swiss Federal Institute of Technology (EPFL), nov 1999.

[11] Grit Denker and Jonathan Millen. CAPSL integrated protocol environment. In *DARPA Information and Survivability Conference and Exposition (DISCEX'00)*, pages 207–221, Hilton Head, South Carolina, Jan 2000.

[12] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, mar 1983.

[13] L. Eschenauer and V. Gligor. A Key-management Scheme for Distributed Sensor Networks. *The 9th ACM Conference on Computer and Communications Security*, pages 41–47, November 2002.

[14] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[15] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[16] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[17] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication. *MobiCOM '00*, August 2000.

[18] Iii Gray J. W. and J. McLean. Using temporal logic to specify and verify cryptographic protocols. In *CSFW '95: Proceedings of the The Eighth IEEE Computer Security Foundations Workshop (CSFW '95)*, page 108, Washington, DC, USA, 1995. IEEE Computer Society.

[19] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[20] Richard A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, may 1989.

[21] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[22] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[23] Armin Liebl. Authentication in distributed systems: a bibliography. *SIGOPS Oper. Syst. Rev.*, 27(4):31–41, 1993.

[24] D. Liu and P. Ning. Establishing Pairwise Keys in Distributed Sensor Networks. *The 10th ACM Conference on Computer and Communications Security*, 2003.

[25] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Comput. Secur.*, 11(1):75–89, 1992.

[26] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[27] John McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, 1990.

[28] Catherine Meadows. Formal verification of cryptographic protocols: A survey. In *ASIACRYPT '94: Proceedings of the 4th International Conference on the Theory and Applications of Cryptology*, pages 135–150, London, UK, 1995. Springer-Verlag.

[29] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[30] Catherine Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1):44–54, 2003.

[31] Jonathan K. Millen. CAPSL: Common authentication protocol specification language. In *NSPW '96: Proceedings of the 1996 workshop on New security paradigms*, page 132, 1996.

[32] Jonathan K. Millen, Sidney C. Clark, and Sheryl B. Freeman. The interrogator: protocol security analysis. *IEEE Trans. Softw. Eng.*, 13(2):274–288, 1987.

[33] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur/spl phi/. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 141, Washington, DC, USA, 1997. IEEE Computer Society.

[34] Roger M. Needham. The changing environment for security protocols. *IEEE Network*, 11(3):12–15, May/Jun 1997.

[35] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[36] B. Clifford Neuman and Stuart G. Stubblebine. A note on the use of timestamps as nonces. *SIGOPS Oper. Syst. Rev.*, 27(2):10–14, 1993.

[37] Johnny OEberg, Anshul Kumar, and Ahmed Royal. Grammar-based hardware synthesis of data communication protocols. In *ISSS '96: Proceedings of the 9th international symposium on System synthesis*, page 14, Washington, DC, USA, 1996. IEEE Computer Society.

[38] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Secur.*, 6(1-2):85–128, 1998.

[39] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[40] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. Tygar. Spins: security protocols for sensor netowrks. In *Proceedings of ACM Mobile Computing and Networking (Mobicom'01)*, pages 189–199, 2001.

[41] Amir Pnueli. The temporal logic of programs. In *The 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, New York, 1977. IEEE.

[42] B. Przydatek, D. Song, and A. Perrig. SIA: Secure information aggregation in sensor networks. In *Proceedings of ACM SenSys 2003*, 2003.

[43] Aviel D. Rubin and Peter Honeyman. Formal methods for the analysis of authentication protocols. Technical Report CITI Technical Report 93-7, CITI, 1993.

[44] Steve Schneider. Verifying authentication protocols in csp. *IEEE Trans. Softw. Eng.*, 24(9):741–758, 1998.

[45] Steve Schneider. Verifying authentication protocol implementations. In *FMOODS '02: Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems V*, pages 5–24, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[46] Gustavus J. Simmons. How to (selectively) broadcast a secret. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 108, 1985.

[47] Panagiotis Georgiadis Stefanos Gritzalis, Diomidis Spinellis. Security protocols over open networks and distributed systems: formal methods for their analysis, design, and verification. *Computer Communications*, 22(8):697–709, 1999.

[48] Paul van Oorschot. Extending cryptographic logics of belief to key agreement protocols. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 232–243, New York, NY, USA, 1993. ACM Press.

[49] Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *SP '93: Proceedings of the 1993 IEEE Symposium on Security and Privacy*, page 178, Washington, DC, USA, 1993. IEEE Computer Society.

[50] H. Yang, F. Ye, Y. Yuan, S. Lu and W. Arbaugh. Toward Resilient Security in Wireless Sensor Networks. *ACM MOBIHOC*, May 2005.

[51] F. Ye, H. Luo, S. Lu, and L. Zhang. Statistical En-route Filtering of Injected False Data in Sensor Networks. *IEEE Infocom'04*, March 2004.

[52] S. Zhu, S. Setia, and S. Jajodia. LEAP: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks. *The 10th ACM Conference on Computer and Communications Security*, 2003.