Slede: A Domain-Specific Verification Framework for Sensor Network Security Protocol Implementations

Youssef Hanna
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50011
ywhanna@cs.iastate.edu

Hridesh Rajan
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50011
hridesh@cs.iastate.edu

Wensheng Zhang
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50011
wzhang@cs.iastate.edu

ABSTRACT

Finding flaws in security protocol implementations is hard. Finding flaws in the implementations of sensor network security protocols is even harder because they are designed to protect against more system failures compared to traditional protocols. Formal verification techniques such as model checking, theorem proving, etc, have been very successful in the past in detecting faults in security protocol specifications; however, they generally require that a formal description of the protocol, often called model, is developed before the verification can start.

There are three factors that make model construction, and as a result, formal verification is hard. First, knowledge of the specialized language used to construct the model is necessary. Second, upfront effort is required to produce an artifact that is only useful during verification, which might be considered wasteful by some, and third, manual model construction is error prone and may lead to inconsistencies between the implementation and the model.

The key contribution of this work is an approach for automated formal verification of sensor network security protocols. Technical underpinnings of our approach includes a technique for automatically extracting a model from the *nesC* implementations of a security protocol, a technique for composing this extracted model with models of intrusion and network topologies, and a technique for translating the results of the verification process to domain terms. Our approach is sound and complete within bounds, i.e. if it reports a fault scenario for a protocol, there is indeed a fault and our framework terminates for a network topology of given size; otherwise no faults in the protocol are present that can be exploited in the network topology of that size or less using the given intrusion model. Our approach also does not require upfront model construction, which significantly decreases the cost of verification.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General-Security and Protection; D.2.4 [**Software/Program Verification**]: Formal Methods; D.2.4 [**Software/Program Verification**]: Model Checking; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification, Specification technique

General Terms

Security, Verification

Keywords

sensor networks, security protocols, model checking, intruder generation

WiSec'08, March 31–April 2, 2008, Alexandria, Virginia, USA.

1. INTRODUCTION

A *sensor network* is a collection of small size, low power, low-cost sensor nodes that have limited 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 [1] such as target tracking, remote surveillance, and habitat monitoring. The operating environments of sensor networks are often hostile, requiring mechanisms for secure communication. In particular, messages containing missions or queries disseminated by administrators [24], control or data messages for decentralized collaborations, etc, need to be secure. A number of security protocols for sensor networks have been proposed in the past decade (see [6] for a survey).

Establishing the correctness of security protocol implementations continues to be a daunting task as their complexity continue to increase. In the past, even widely-studied security protocols are shown to have faults that are detected much later [8, 41, 31].

Verifying sensor network security protocol implementations is even harder. The reason behind this is that these implementations are developed for a severely resource constrained environment. Efficiency and code size are more likely to weigh over readability and understandability, which in turn increases the likelihood of inconsistencies and errors.

The demand for robust performance in unattended deployment scenarios in hostile environments makes this problem more severe, which necessitates uncovering any errors as early in the development process as possible because on-site bug-fixes and updates are often impossible or, at the very least, prohibitively costly. Therefore, detecting and removing errors from sensor network security protocol implementations is extremely important.

The variety of methodologies that have been suggested for verification may be classified under two broad categories, simulation and formal methods. Functional simulation of the implementation

using simulators such as TOSSIM [26] and/or test runs of the protocol implementation on sensor network test beds are the primary techniques used within the research community to verify implementations due to its simplicity and scalability. However, exhaustive simulation is often impractical, and the likelihood that these tests will uncover subtle errors is diminishing. Therefore, formal methods such as model checking [14] which use mathematical reasoning to systematically explore all possible paths, and which are based on an unambiguous specification of the implementation, have emerged as an alternative. Since the entire space of possible execution paths is searched in order to establish definitive correctness, all subtle errors in the covered space are exposed. These verification techniques have shown significant potential in recent years [3, 16].

Applying model checking technique for verification, however, requires non-trivial efforts primarily because model checking tools often require a specification (model) of the system under verification. This specification is written in a specialized language. Learning this language itself can be a daunting task. This task is further complicated by the impedance mismatch between the implementation language and the modeling language. For example, while the dominant implementation language for sensor network applications (*nesC*) uses an event-based paradigm, an example modeling language (*Promela* for Process or Protocol Meta Language [21]) uses message-driven paradigm.

Moreover, constructing a model from an implementation of the protocol may not be desired for several reasons:

- Building models is time consuming and can take more time to do than to write the implementation of the protocol [17].
- The level of knowledge and effort required may prevent many domain experts from attempting such task [38].
- Such model may abstract many potentially troublesome and error prone details of implementation code that are more likely to contain bugs [4].
- Keeping model and the code synchronized is hard as the code is deployed and maintained. Therefore, even though the abstract model is verified correct, security flaws may be introduced in the implementation during maintenance of the code [4].

This work addresses these problems. We present our approach for automatic formal verification of sensor network security protocol implementations written in nesC [15]. Our approach does not require upfront model construction thereby significantly easing the task of sensor network developer. Instead, a skeleton model is automatically extracted from the *nesC* implementation of the security protocol. This extracted skeleton model is then automatically composed with intrusion models and desired network topologies to create a complete verifiable model. The key technical contributions of our work are:

- An automated technique for extracting a skeleton model of the protocol from its nesC implementations,
- a light-weight language design to semi-formally describe the protocol specification,
- an approach for customizing the skeleton model of the protocol with the help of the protocol specification and inbuilt intrusion detection models to generate a complete verifiable models, and,
- a technique for mapping the results of verification back to the domain terms.

```
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 Module Example

To evaluate our approach we have verified implementations of two security protocols designed for sensor networks. The first protocol is the one-way key chain based one-hop broadcast authentication scheme [45]. The second protocol is μTesla protocol [37]. The intrusion model used for verifying these protocols was a modified version of the Dolev-Yao model [13] that supports the standard node capture attack model where one or more nodes are under control of an attacker and behave maliciously. This is the intruder model currently used by our approach; however, it can be easily extended to allow other intrusion models to be used. Our approach confirmed known flaws in both these protocols.

The rest of this work describes our approach in detail. To make it self-contained, we briefly describe the nesC language and model checking in the next two sections. In Section 4, we describe a motivating example. Section 5 describes our verification framework. In order to automatically generate the intrusion model that represents malicious behavior in the model, our framework requires users to provide information about the protocol message structures exchanged using a light-weight annotation language. Section 6 describes this language through an example. Section 7 describes the results of applying our verification process to a selection of sensor network protocols. Section 8 discusses related work. Section 9 describes future work and Section 10 concludes.

2. THE NESC LANGUAGE

nesC [15] is an extension of the C language designed to develop sensor network applications. A nesC application consists of 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 declarations (shared by other elements of the modules), command declarations (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.

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

3. MODEL CHECKING

In model checking, the system to be verified is represented as a finite-state model, a model that represents all possible scenarios of the system behavior. After the model is built, the verifier (or the model checker) takes the model and the requirement as input and goes through the model to check if any branch of the model is violating the requirement. If one branch has violated the requirement of the model, then the model checker creates a *counterexample*, where the branch that violated the requirement is displayed as a sequence of states.

Verification from abstract specifications using techniques such as model checking has been used to find flaws in cryptographic protocols [29] (see [32] for a survey). The main benefit of model checking over verification techniques like simulating using TOSSIM and manual inspection is that the model verified by the model checker covers all possible scenarios of the behavior of the system. Unlike simulation that does not provide exhaustive coverage, verification using model checking gives a more thorough analysis of the system by checking satisfiability of the requirement through every branch of the model representing the system.

As mentioned previously, one of the main problems that hinders using model checking is the effort required to build the models. The time taken to build the models and the expertise required to learn how to build them make model checking a tedious job. Some work has attempted to automate such process by extracting the models from the code (e.g. [17]). In particular, model checkers that take source code as input such as Java path finder [19], Bandera [12], etc, allow general-purpose Java programs to be model checked, but these techniques do not support the nesC language and do not consider security protocol related aspects such as intrusion models, network topologies, etc. To the best of our knowledge, our work is the first such approach that is applicable to sensor network security protocols.

4. MOTIVATION

To illustrate the problem with current verification techniques, we describe in this section the verification of an example sensor network security protocol, the polynomial pool based pairwise key establishment protocol [27]. The following describes the protocol in some detail.

The protocol 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 established between A and B.

A part of the nesC implementation of this protocol is shown in Figure 2. This part is responsible for the path discovery . First, we can see that the implementation is much more complicated compared to the abstract description of the protocol. Even though the manual inspection and mathematical analysis of the abstract specification of the protocol might ensure security of the protocol, the implementation of the protocol may not accurately implement the abstract protocol, thus leaving room for subtle errors that can be exploited by adversaries. Moreover, verification of the implementation using simulators like TOSSIM [26] or using testbeds may not exhaustively test all paths in the protocol implementation, thus leaving room in the untested paths for possible attacks.

```
1 event uint8_t* Channel.receive(uint8_t *msg){
2  uint16_t node_s,node_d;
3  uint16_t key[4],keys[4],keyd[4];
4  uint8_t cks[2],ckd[2];
5  uint8_t i,j,type;
6  bool same,sames,samed,cp;
7  ...
8  // every sensor that received the msg
9  // checks itself
10 if((type==4)&&(node_s!=sID)&&(node_d!=sID)) {
11  msg[5+ss+ss] = msg[5+ss+ss]-1;
12  sames = check(msg+5+ss,secret,cks);
13  samed = check(msg+5,secret,ckd);

15  if((sames==1)&&(samed==1)){
16   // node_my send *cks *shares to node_s
17   j=cks[1];
18   i=ckd[1];
19   call ComputeKey.compute(
20    (uint8_t *)&secret[j],node_s,(uint8_t *)keys);
21   call ComputeKey.compute(
22    (uint8_t *)&secret[i],node_d,(uint8_t *)keyd);
23   // generate a random number
24   key[0] = call Random.rand();
25   key[1] = call Random.rand();
26   key[2] = call Random.rand();
27   key[3] = call Random.rand();
28   // encrypt key[0] with keys and keyd separately
29   call Primitive.encrypt(
30    (uint8_t *)keys,(uint8_t *)key,(uint8_t *)keys);
31   call Primitive.encrypt(
32    (uint8_t *)keyd,(uint8_t *)key,(uint8_t *)keyd);
33   msg[0] = 5; // type5: send path key
34   memcpy(msg+1,(uint8_t *)&sID,2); // src node
35   memcpy(msg+3,(uint8_t *)&node_s,2); // dest node
36   msg[5] = ckd[0];
37   memcpy(msg+6,(uint8_t *)keyd,8);
38   msg[14] = cks[0];
39   memcpy(msg+15,(uint8_t *)keys,8);
40   if(sendflag==0) {
41    call Channel.send(node_s,msg);
42    sendflag = 1;
43   }
44   ...
45  }
46 }
47 ...
48 }
```

Figure 2: A snippet of the polynomial pool based pairwise key establishment protocol implementation [27]

A model for this protocol in the PROMELA language is shown in Figure 3. This model can be verified by the model checker Spin [22]. One challenge in building such model is to emulate the physical characteristics of sensor networks in the model that plays a key role in verification. For instance, to emulate the wireless channels, the receiver of the message should not prevent other sensors from receiving it as well.

If PROMELA's inbuilt construct for sending and receiving message is used to emulate sending and receiving message in this protocol, the message will be removed from the channel when a receiver receives the message. Thus, a modified version of receiving like the one used in Figure 3 (lines 3-9) is needed. Furthermore, some mechanism to emulate collision and mutual exclusion is also needed. Here the **atomic** construct of PROMELA is used to en-

```
1 do
2   /*Receiving a msg*/
3 :: channel.containsMsg==1 ->
4    atomic {
5      msg.src = channel.msg.src;
6      msg.dest = channel.msg.dest;
7      msg.info = channel.msg.info;
8      ...
9    }
10   if
11   :: msg.type == 4
12       && msg.src == agent_id
13       && msg.dest == agent_id ->
14       ...
15       /* Non-determinism for
16          emulating random call*/
17       if
18       :: key[0] = 1;
19       :: key[0] = 2;
20       :: key[0] = 3;
21       ...
22       fi;
23       /* same for key[1], key[2]... */
24       ...
25       /* Sending message */
26       atomic{
27          channel.msg.src = msg.src;
28          ...
29       }
30    :: else -> ...
31    fi;
32 od;
```

Figure 3: PROMELA model for the code in Figure 2

sure these properties. This construct ensures that all statements between line 5 and 8 are executed as an atomic transaction.

The moral of the story is that, even though model checking allows us to conduct more rigorous verification compared to simulation and test-beds, a simple error in building the model can easily deviate the behavior of the model from the intended behavior.

An approach that provides technique for automatically extracting a model from protocol implementation and verifies them using existing model checkers seems to solve these problems. Once the automatic extraction approach is verified to be correct, the models extracted by it are guaranteed to faithfully represent the implementation. Thus there are no inconsistency issues between the model and the protocol implementation during the model construction.

Such models can be kept synchronized with the actual implementation by a simple regeneration of the model after the implementation has changed. Such regeneration could be as simple as compiling the protocol implementation and can also be incorporated into the compilers.

Furthermore, such approaches make the benefits of rigorous formal verification technique available to a developer without requiring them to learn the intricacies of formal techniques and their specification languages, which in turn significantly reduces the overhead of protocol implementation verification.

Our approach is an example of such technique. In the next section, we describe our framework for verifying nesC implementations of sensor network security protocols.

5. VERIFICATION FRAMEWORK

Our framework provides automatic verification of sensor network security protocols by extracting models from the protocol implementation. In addition, the framework automatically generates intruder models that are necessary for verification of the security protocols. It is built on top of the nesC compiler version 1.1.1 [35] and uses the Spin model checker [23] as the backend.

The overview of the framework is shown in Figure 4. The framework takes the source code of the protocol as input. In order to verify a protocol implementation, besides the source code, the framework requires a small amount of extra information such as the structure of messages used in the protocol, a deployment topology, and properties that need to be verified about the protocol. This information is provided as comments at the top of the main configuration of nesC implementation using our annotation language. The generated protocol model is merged with the generated intruder model and then verified using the Spin model checker [23]. If there is a violation of the protocol objective, the counterexample generated by Spin is translated into nesC statements using the counterexample translator.

In this section, we describe the main components of the framework: the Protocol Model Generator, the Intruder Model Generator and the Counterexample Translator.

5.1 Protocol Model Generator

The protocol model generation phase replaces the original code generation phase of the nesC compiler on top of which our framework is built. It translates the protocol implementation into a PROMELA model.

The major challenge in extracting a model from the implementation is to generate a model of small number of states as possible. If the number of states (also known as state space) increases beyond a certain limit, the model checker will not be able to verify all the model and may never halt. This problem is known as *the state explosion problem*, which is a major challenge when applying the model checking technique.

In order to generate a model with small state space, we have to put some boundaries on the generated model. Our current prototype requires a bound on the number of nodes involved in the protocol as well as the topology of the network. The user has to state the number of nodes and the topology of the network. The user can provide the framework with such information through our new lightweight annotation language that we describe in Section 6. This annotation language is also required for specifying the verification goals of the protocol. In other words, the security properties that need to be satisfied by the protocol are stated using this annotation language so that the framework can verify if the implementation is able to satisfy them or not.

5.2 Intruder Model Generator

A malicious behavior is required in the verification process to ensure that the protocol satisfies its goals with the presence of malicious behavior. In order to automatically generate an intruder that behaves maliciously, the framework requires some information about the protocol. Please note that the framework does not infer the protocols from the nesC code (i.e. the framework does not infer the message sequencing of the protocol from the implementation). What the framework does is verify that the implementations of the security protocols satisfy their goals in the presence of malicious behavior.

In order for the generated intruder model to behave malicioulsy, the intruder should be able to receive messages, read their content if able to decrypt them and send false data. To be able to do so, the intruder needs to know the types of messages exchanged between principals in the protocols and the structure of the message i.e which fields in the message represents information about sender, receiver, data, etc so that it can read and alter the message contents.

For example, imagine a simple protocol that consists of two message types: ping and ack, each of which contain a source and destination address and a sequence number. In order to attack this naive

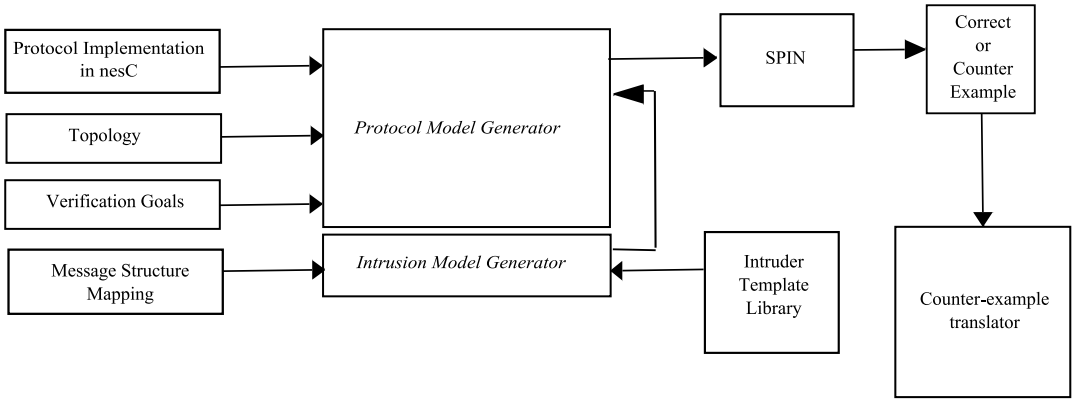

Figure 4: Overview of the Verification Framework

protocol, the intruder may intercept the messages to the destination, exchange the source and destination address and reply back with this modified message as one possible attack. In order to automatically generate this attack the intruder needs to be aware of the structure of these message types in the implementation so that it can retrieve and manipulate fields in the messages that are exchanged by the protocol e.g. sender and destination address. The framework can get this information from the annotation language through which the message structure is analyzed.

The intruder model that the framework currently generates is a modified version of the Dolev-Yao style [13] that allows modeling wireless network channels. An intruder in this model can read all messages and modify their contents, but cannot read encrypted messages unless it has the encryption key; yet, an intruder intercepting a message need *not* block the original recipient of the message from receiving it. This approach models the behavior of wireless network channels where the malicious node and the original recipient are in the reception range of the message source, thus both receive the message. Besides, this modified model can also represent node capture attacks, where the intruder is a legitimate node that has been captured by an adversary and then placed back after being modified to act maliciously.

Even though our framework currently supports the generation of intruder models according to only one intruder template that allows verification against node capture attack, our framework can easily be extended to provide generation of other intruder models according to different patterns. As Figure 4 shows, we intend to allow the users of our framework to add new intruder patterns to be used for generating intruder models. This extensibility feature would help achieve more thorough verification against different types of attacks.

5.3 Verification and Counterexamples

The generated model containing the model of the protocol implementation, the intrusion model and the environment models are given as inputs to the Spin model checker [23], which verifies whether the model violates the objectives stated using our annotation language. 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.

6. ANNOTATION LANGUAGE

Our annotation language provides necessary information required for automatic verification of nesC security protocols. In this section, we describe how our annotation language is used to generate intruder models, specify topology and describe the objectives sought of the protocol. We illustrate how our annotation language can be used through a simple example.

```
1 /*@
2   @ message Ping mapsto IntMsg{
3   @  int sender mapsto src;
4   @  int receiver mapsto dest;
5   @  private int data mapsto info;
6   @  }
7   @ message Ack mapsto IntMsg{
8   @  int sender mapsto src;
9   @  int receiver mapsto dest;
10  @  }
11  @ message_type IntMsg.type {
12  @  1: Ping;
13  @  2: Ack;
14  @ }
15  @ node A{ }
16  @ node B{ A; }
17  @ node Intruder { B; A; }
18  @ A.Send.send(Ping)->
19  @     !Intruder.Receive.receive(Ping)
20  @*/
```

Figure 5: An Example Verification Configuration

An example of the *verification configuration* written in our annotation language is displayed in Figure 5. The verification configuration is defined in comments. The annotation comments start and end with an at-sign (@) and is written at the top configuration of the nesC application. The verification configuration is used to describe message structure mapping, topology and objectives.

6.1 Message Structure Mapping

As mentioned in the previous section, the intruder needs to be aware what kind of messages it is receiving, thus needs to know about the message structures. An example message declaration is given in Figure 5 (lines 2-6), where a message type `Ping` is defined. This message type in specification is mapped to the structure `IntMsg` in the implementation using the **`mapsto`** special word. Similarly, the fields in the message structure are mapped using **`mapsto`** special word. In the field declaration, we have a set of predefined terms for which the fields in the implementation can be mapped to. The set of terms currently used has three terms: $\{sender, receiver, data\}$. In the example message declaration, a message of type `Ping` will contain three integers. Two integer fields represent the sender and the receiver of the message and are mapped respectively to fields `src` and `dest` of the structure `IntMsg`. The third field represent the data carried by the mes-

sage, and is mapped to the field `info` of the structure `IntMsg`. The **`private`** special word is used to declare that the content of the `info` field is encrypted. In case a message contains more than one field with relevant data that need to be mapped, another field mapping is written with that same term **`data`** appended with an incrementing number. For instance, if we have two fields `a` and `b` that carry data, then the mapping will correspond to terms **`data`** and **`data2`** respectively.

One common approach in implementing protocols in nesC is that only one message structure is used for all different types of messages used in the protocol. For instance, we can see that both message declarations `Ping` and `Ack` (lines 7-10) are mapped to the same structure `IntMsg`. To differentiate between different types of messages, usually one field of the message structure in the implementation is used to identify the type of the received message. In the annotation language, this field is referenced using the **`message_type`** special word. The field `type` of the structure `IntMsg` (lines 11-14) is the field responsible for identifying if the type of the message is either `Ping` or `Ack` (where the value of `type` will be either 1 or 2 respectively).

6.2 Topology

To avoid the problem of state explosion, Our current prototype allows for verifying the protocol with one topology at a time. The annotation language allows the user to define the topology of the network against which the protocol will be verified. In lines 15-17 of Figure 5, we define the nodes to be involved in the protocol using the special word **`node`**. After the node name, the nodes to which the node is connected are stated. In this example, we have a linear topology between nodes `A` and `B`. The compromised node that acts maliciously is represented using the special word **`Intruder`**.

Even though sensor networks are supposed to have dynamic topologies, we believe that this prototype is still in cope with the current implementations of sensor network security protocols where handling dynamic topologies has not yet reached the level of maturity to be deployed in real life applications.

6.3 Objectives

The objectives (requirements) of the protocol are described in terms of commands and events used in the implementation. In lines 18-19, the trivial objective in this example means that if node `A` sends a message of type `Ping`, *then* (denoted by the construct **`->`**) node **`Intruder`** will not receive it. Please note that the right hand side of the objective (after the construct **`->`**) is always a negation. This way we are verifying that something bas does not happen (i.e. Intruder does not know a secret that it is not supposed to know).

Since the malicious node acts as a compromised node, then it has the ability to use the same commands and events of the legitimate users. For instance, we stated in the objective that the intruder receives a message of type `Ping` by writing **`Intruder.`**`Receive.receive(Ping m)`. In order to denote that the intruder was able to understand data that is supposed to be secret, one more special word is used: `knowsData`. For instance, since the message type `Ping` has a **`private`** field that the intruder should not be able to read, this would be denoted as `(!` **`Intruder.`**`knowsData(Ping m))`.

If the objectives are satisfied, then the protocol is verified as secure. Otherwise, the counterexample generated by Spin is translated in terms of nesC statements.

7. EVALUATION

In this section, we evaluate our framework prototype. We then describe verification of two sensor network security protocol implementations using our framework. For both protocols, the framework was able to find flaws in the implementation. All experiments described in this section were conducted on a Dell PowerEdge 1850 with dual 3.8 GHz processors and 2 GB RAM. The version of SPIN used for these experiments was 4.2.7.

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

7.1.1 Protocol Overview

The one-way key chain based one-hop broadcast authentication scheme was proposed by Zhu et al. [45]. During the initialization step of this protocol, 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.

The protocol then proceeds as follows: 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.

7.1.2 Known Flaw in the Protocol

As pointed out by Zhu et al. [45], the adversary can launch the following attack: 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, after knowing k_{i+1}, 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 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 $h(m_i, k_{i+1} XOR KC)$. However, the defense will not be useful if the adversary has obtained KC by compromising a neighbor of A.

7.1.3 Verification using Slede

To test if our prototype can automatically detect the above attack, we verified an implementation of this protocol with respect to a property informally stated as follows: "if a malicious node sends data, the receiver should detect that the sender is an intruder."

The verification setup including the verified property is shown in Figure 6 using our annotation language. Two message structures are defined, `KeyMsg` that sends the keys, and `DataMsg` that sends the message authenticated using the next key to be broadcast (the MAC field in this message structure is mapped to **`data2`** as shown in line 11). Note that the field `info` in the implementation message structure `IntMsg` holds the encrypted key in the first message type `KeyMsg` (line 5) and also holds the data for the message of type `DataMsg` (line 10). This is one of the main reason message structure mapping is required since the fields of the message structure in the implementation can be reused in different message types.

The field `type` in the message structure has a value of 1 or

2 corresponding to messages of type KeyMsg and DataMsg respectively (lines 13-16). The objective uses the special word **Intruder** to represent the intruder model generated by the framework as described in Section 6.3. Notice that the intruder can send messages using the command Send.send() since the intruder is a node that has been captured and modified to behave maliciously according to the intruder pattern that our framework is usually using. In our sample nesC implementation for this protocol, node B turns its green led on when it receives a message from an authenticated source. Thus the objective ensures that node B should *not* turn its green led on if a message of type DataMsg is sent (or intercepted and modified) by an intruder (lines 20-21), otherwise the intruder would have successfully impersonated a legitimate node and fooled node B.

Our approach was able to detect this attack. The performance results for verifying this protocol are discussed in Section 7.3.

```
1 /*@
2   @ message KeyMsg mapsto IntMsg{
3   @  int sender mapsto src;
4   @  int receiver mapsto dest;
5   @  private int data mapsto info;
6   @  }
7   @ message DataMsg mapsto IntMsg{
8   @  int sender mapsto src;
9   @  int receiver mapsto dest;
10  @  int data mapsto info;
11  @  private int data2 mapsto MAC;
12  @ }
13  @ message_type IntMsg.type {
14  @  1: KeyMsg;
15  @  2: DataMsg;
16  @ }
17  @ node A{ }
18  @ node B{ A; }
19  @ node Intruder {B; A;}
20  @ Intruder.Send.send(DataMsg) ->
21  @         !B.Leds.greenOn()
22  @*/
```

Figure 6: Verification Configuration for One-way Key Chain Based One-hop Broadcast Authentication Scheme [45]

7.2 Verification of the μTesla protocol

7.2.1 Protocol Overview

μTESLA [37] 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. An implementer of the μTESLA protocol may not fully understand the necessity of implementing secure time synchronization for the security of the protocol, which is not explicitly specified in the description of the protocol itself. Hence, the implementer may choose to implement a simple but not secure time synchronization protocol as the fundation of the μTESLA. As elaborated in the following, our approach can automatically test such mis-implementations.

7.2.2 Verification Using Slede

The verification resulted in the scenario shown in Figure 8. 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 t0-2, and then forwards it to node B. Since the time synchronization protocol is attacked, the clock in node B will not get

```
1 /*@
2   @ message Timestamp mapsto IntMsg{
3   @  int sender mapsto src;
4   @  int receiver mapsto dest;
5   @  private int data mapsto info;
6   @  }
7   @ message Data mapsto IntMsg{
8   @  int sender mapsto src;
9   @  int receiver mapsto dest;
10  @  int data mapsto info;
11  @  private int data2 mapsto MAC;
12  @  }
13  @ message Key mapsto IntMsg {
14  @  int sender mapsto src;
15  @  int receiver mapsto dest;
16  @  private int data mapsto info;
17  @ }
18  @ message_type IntMsg.type {
19  @  1: Timestamp;
20  @  2: Data;
21  @  3: Key;
22  @ }
23  @ node A{ }
24  @ node B{ A; }
25  @ node Intruder {B; A;}
26  @ Intruder.Send.send(Data) ->
27  @         !B.Leds.greenOn()
28  @*/
```

Figure 7: Verification Configuration for μTesla protocol [37]

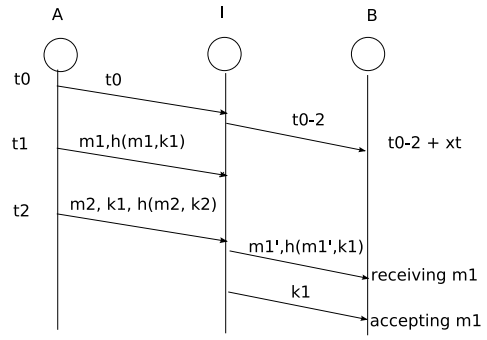

Figure 8: Assumption Violation in μTesla Implementation

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.

7.3 Performance

In this section, we describe the performance of our framework in terms of number of states generated, memory used and time taken to detect the flaws. In Table 1, the properties of the sample implementations we used to verify the two protocols are displayed. The implementation of the μTesla protocol has more lines of code and more number of fields in the message declarations than the one-hop broadcast authentication scheme.

In the performance charts in Figure 9, we can see that when we have two nodes and the intruder can communicate with both of them, the μTesla consumes more time, states and memory than the

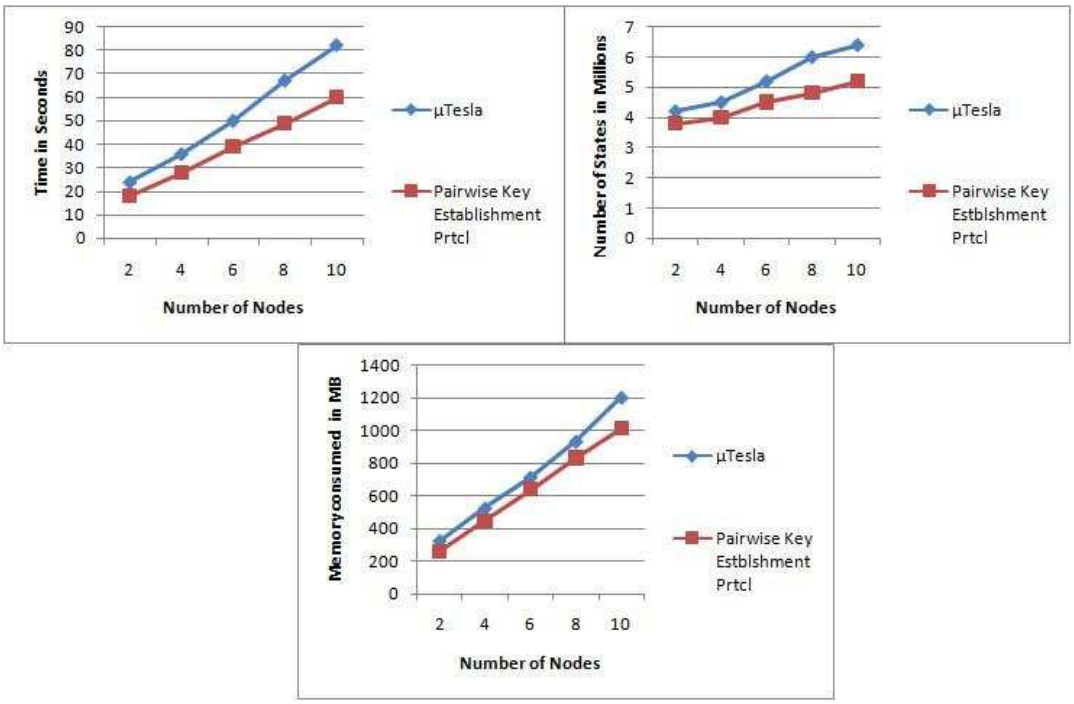

Figure 9: Performance Results

	Lines of Code	Message Declarations	Message Fields
One-hop Broadcast Authentication Scheme	170	2	4
μTesla	200	3	4

Table 1: Implementation properties of the protocols

Broadcast Authentication scheme. In our experiment, the intruder is communicating with only two nodes, even when more nodes are present in the topology. The reason behind that is that when we increased the number of neighbors of the intruder to three instead of two (not shown in the figure), a state explosion occured. Thus, we bound the number of neighbors of the intruder in the verification of both protocols to two neighbors through the whole experiment. We can see in Figure 9 that the verification of μTesla requires more time, memory and has more states than the verification of the Authentication scheme. The reason behind this is that μTesla has more lines of codes and more message declarations than the Authentication scheme.

We have concluded from our experiment that there are four main criteria that affect the performance of our framework: the lines of code of the protocol implementation, the number of message declarations, the number of fields in the messages exchanged and the number of neighbors to the intruder node.

8. RELATED WORK

The closest work related to our approach is by Bhargavan *et al.* [4] and by Goubault-Larrecq and Parrennes [17]. Bhargavan *et al.* [4] present an approach for verifying protocol implementations written in F# using ProVerif [5], a theorem prover as the underlying mechanism. Our work is different in two dimensions: first, we are verifying protocol implementations written in nesC, and second, we use a model checker as the underlying technology.

Goubault-Larrecq and Parrennes [17] present an approach for verifying protocol implementations in C. Their approach models secrecy properties as reachability properties of the C implementation and analyzes these properties. A simple pointer analysis technique is used to keep the verified model as close as possible to the actual implementation. Unlike our approach that provides support for the entire nesC language, this approach is useful only for C implementations; however, the insights described by Goubault-Larrecq and Parrennes [17] could be used to enhance the underlying verification technique for our framework.

Tobarra *et al.* [43] propose an approach for verification of sensor network security protocols using model checking. In their approach, they verify the models of the protocols written in HLPSL [10] modeling language using the model checking tool Avispa [2]. They were able to discover attacks in two security protocols; however, unlike Slede, they verify the protocols from models written manually, which does not provide any guarantee that the implementation of the protocol is correct.

Besides the previous approaches, 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 [8], 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 [44].

Meadows developed the NRL protocol analyzer for the analysis of cryptographic protocols [31]. The NRL protocol analyzer was used to find flaws in a number of cryptographic protocols including selective broadcast protocol by Simmons [41], Resource Sharing Protocol by Burns and Mitchell [7], 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 [28]. Yet another tool was Interrogater developed by Millen *et al.* [33]. Kemmerer [25] used general-purpose formal methods technique as tools to verify cryptographic protocols. Schneider

adapted the CSP model for verification of security protocols [40]. For a detailed summary of verification techniques, please refer to a survey by Rubin and Honeyman [39], Meadows [30], Gritzalis *et al.* [42], and a more recent survey by Buttyan [9]. Compared to these ideas, Slede has two advantages. First, it can be used to verify implementations. Second, that is is highly customized for sensor network applications, which enables many domain-specific optimizations resulting in improved scalability.

Tools for model checking source code directly are also related to this work. In particular, Blast [20], Bandera [12], Java Path Finder [19], CMC [34], etc, have successfully verified C and Java implementations. Like these approaches, our framework also verifies source code directly; however, unlike these techniques our framework is highly customized towards model checking security protocol implementations in nesC.

9. FUTURE WORK

Our approach opens up a number of interesting avenues that we plan to explore in the future. One such area is analyzing the influence of non-functional properties, such as memory, bandwidth, and power constraints on security properties. Sensor nodes are resource and bandwidth constrained. It may not be sufficient in this environment for a node to have an excellent security property at the cost of depleting 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.

We also plan to improve on our current prototype. The current implementation of Slede has some limitations partly due to the restrictions of the underlying model-checking technology and due to the specific translation approach that we have taken. Limitations include the bound on the number of neighbors to the intruder, as well as having only one malicious intruder in the model. A limitation is on the number of participant nodes in the verification process that causes a state explosion when it increases. We are currently studying network decomposition techniques (i.e. [11]), which allow a large network topology to be verified by decomposing it into smaller topologies, will help alleviate this issue.

10. CONCLUSION

In this work, we presented our verification framework for sensor network protocol implementations. The key advantages of the framework is that it automatically extracts verifiable models from nesC implementations and allows automatic generation of protocol specific intrusion models from lightweight annotations. Our framework confirmed the flaws in two sensor network specific protocols.

Our approach is sound and complete within bounds, i.e. if there is a fault scenario in the protocol, the framework will detect it and if the framework terminates for a network topology of given size declaring no faults, then there are no faults in this network or networks of smaller size using the given intruder model.

Security in sensor networks is an important problem. Our approach brings the advantages of explicit-state model checking to the sensor network applications, thereby paving the way to improve their security at a relatively small cost.

Acknowledgments

This work is supported in part by the National Science Foundation under grant CT-ISG: 0627354. The authors would like to thank the anonymous reviewers of WiSec 2008 for their insightful comments that helped improve the presentation of the paper and to the participants of the Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2006) poster session, where this work was first presented [18]. Thanks to Samik Basu for pointing out an appropriate intrusion model for Slede. Thanks are also due to Sencun Zhu for the discussion about the results in this paper.

11. REFERENCES

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

[2] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. S. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In *CAV'2005*, Edinburgh, Scotland, 2005. Springer-Verlag.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[4] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 139–152, Washington, DC, USA, 2006. IEEE Computer Society.

[5] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[6] D. Boyle and T. Newe. Security protocols for use with wireless sensor networks: A survey of security architectures. In *Third International Conference on Wireless and Mobile Communications (ICWMC'07)*, page 54, March 2007.

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

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

[9] L. 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.

[10] Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, J. Mantovani, S. Modersheim, and L. Vigneron. A high-level protocol specification language for industrial security-sensitive protocols. In *SAPS*, pages 193–205, 2004.

[11] E. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *Fifteenth International Conference on Concurrency Theory (CONCUR 04)*, pages 276–291. Springer-Verlag, 2004.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Preanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on*

Software engineering, pages 439–448, New York, NY, USA, 2000. ACM Press.

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

[14] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. 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.

[16] P. Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM Press.

[17] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In R. Cousot, editor, *VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379, Paris, France, Jan. 2005. Springer.

[18] Y. Hanna and H. Rajan. Slede: event-based specification of sensor network security protocols. *SIGSOFT Softw. Eng. Notes*, 31(6):1–2, 2006.

[19] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 23July 2001.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.

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

[22] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–95, May 1997.

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

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

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

[26] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

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

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

[29] W. Marrero, E. Clarke, and S. Jha. Model checking for security protocols, 1997.

[30] C. 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.

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

[32] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *MMM-ACNS '01: Proceedings of the International Workshop on Information Assurance in Computer Networks*, page 21, London, UK, 2001. Springer-Verlag.

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

[34] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[35] nesC Compiler. http://sourceforge.net/projects/nescc.

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

[37] 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.

[38] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11*, pages 267–276, New York, NY, USA, 2003. ACM Press.

[39] A. D. Rubin and P. Honeyman. Formal methods for the analysis of authentication protocols. Technical Report CITI Technical Report 93-7, CITI, 1993.

[40] S. 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.

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

[42] P. G. 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.

[43] L. Tobarra, D. Cazorla, F. Cuartero, G. DÃaz, and E. Cambronero. Model Checking Wireless Sensor Network Security Protocols: TinySec + LEAP. In *WSAN'07*, pages 95–106, Albacete (Spain), September 2007. IFIP Main Series, Springer.

[44] P. 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.

[45] 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.