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

Youssef Hanna and Hridesh Rajan

Submitted for publication

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

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@iastate.edu

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

ABSTRACT

Sensor networks are often deployed in hostile situations. A number of protocols are being developed to secure these networks. Current means to verify these protocols include simulation, manual inspection, and running them on sensor network testbeds. These techniques leave room for subtle errors in protocol implementations that can be exploited by adversaries. The contribution of this work is the design, implementation and early evaluation of a domain-specific verification framework for nesC implementations of sensor network security protocols. We call our verification framework *Slede*. Technical underpinnings of Slede include support for automatic extraction of PROMELA models from nesC source code, an annotation language to guide the verification process, and an automatic intrusion model generator. Preliminary evaluation shows that Slede was able to discover flaws in a canonical cryptographic protocol by Needham and Schroeder and two security protocols specific to sensor networks. We also demonstrate that a protocol aware intrusion model automatically generated by Slede incurs a small extra cost compared to models handwritten by model checking experts. By automating a significant portion of the verification process, Slede promises to make it easier to apply finite-state model checking to verify nesC protocol implementations.

Categories and Subject Descriptors

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, slede, intrusion model generation

1. INTRODUCTION

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 [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 [25], 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).

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 (e.g. see [9, 44, 33]). Finding flaws in the security protocol implementations is even harder because they tend to be more complex compared to abstract protocol specifications [18]. Moreover, often just verifying the protocol specification (model) is not sufficient due to frequently found gaps between the model and its implementation [3]. Analyzing the implementation of protocols is important to guarantee that there are no discrepancies between the specification and the implementation of the protocol.

Verifying sensor network security protocol implementations is even harder primarily due to two reasons. First, these implementations are more complex because the security protocols for sensor networks protect against more cryptographic failure modes compared to their counterparts. Second, these implementations are developed for a severely resource constrained environment. Efficiency and code size is more likely to weigh over readability and understandability, which in turn increases the likelihood of inconsistencies and errors.

Verification techniques that are commonly employed by the sensor network community include simulating the protocol implementation using a commonly used simulator called TOSSIM [29], manual inspection of the protocol code, and test runs of the protocol implementation on sensor network test beds. Considering that sensor networks are often deployed in critical scenarios, the current techniques for verification leave room for subtle errors and vulnerabilities in protocol implementations.

Model checking as a verification technique has shown significant potential in recent years [2, 17]. In particular, model checkers that take source code as input such as Java path finder [20], Bandera [12], etc, make this technology more accessible to a broader audi-

```
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

ence. As Dwyer *et al.* [41] state, "the level of knowledge and effort required ... currently prevents many domain experts, who are not necessarily experts in model-checking, from successfully applying model checking to systems and software analysis problems". In this work we describe our framework *Slede*, which is meant to address this problem for sensor network applications.

Slede allows for more rigorous verification of sensor network security protocol implementations written in nesC [16], a dominant language for the sensor networks paradigm. Key contributions of this work include: an approach for automatic extraction of verifiable models from nesC applications, an annotation language to guide the verification process, a technique for automatic generation of intrusion models, and an industrial strength tool that embodies these techniques.

To evaluate we have verified an authentication protocol by Needham and Schroeder [37], a commonly used example in the protocol verification community. We compared Slede's performance with an intruder model by Maggi and Sisto [31], highly customized for Needham Schroeder protocol. We have also verified two security protocols designed for sensor networks. The first protocol was for establishing pairwise keys in sensor networks [48] and one-way key chain based one-hop broadcast authentication scheme [50]. Slede confirmed known flaws in both these protocols.

The rest of this work describes our approach in detail. To make it self-contained, in the next section we briefly describe the nesC language. A basic familiarity with an imperative programming language such as C is assumed. Section 3 describes our model extraction methodology, our intrusion model generation technique, and our verification framework *Slede*. In order to automatically generate the intrusion model, our framework requires users to provide information about the protocol message structures and sequence of message exchanged using an annotation language. Section 3 also describes the key constructs of this language. Section 4 describes the results of applying our verification process to a selection of sensor network protocols. Section 5 discusses related work. Section 6 describes future work and Section 7 concludes.

2. THE NESC LANGUAGE

nesC [16] is an extension of the C language designed to develop sensor network applications. nesC applications 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. VERIFICATION FRAMEWORK

In this section, we describe Slede. Slede is a domain-specific model checking framework for sensor network security protocol implementations in nesC. It features new mechanisms for extracting PROMELA models from nesC implementations and for generating intrusion models from protocol specifications. The description includes the main components of Slede, namely front-end, protocol model generator, intrusion model generator, environment model library, intruder template library, and counter-example translator as shown in Figure 2.

3.1 Slede's Front-end

The front-end of our framework accepts the complete nesC language and our annotation language (described in the next section). We have modified the nesC compiler to suit our task [38]. The modifications were limited to the parsing phase and the code generation phase. We left the semantic analysis intact so that a syntactic and/or semantic error in the nesC compiler is also a syntactic and/or semantic error in Slede and verification is only attempted on protocol implementations free of compile-time errors.

The front-end generates an abstract syntax tree of the protocol implementation, which is then passed to the protocol model generator, which is responsible for automatically extracting verifiable PROMELA models [22].

It is widely understood that model checkers suffer from two problems, namely state space explosion and environment modeling (or environment generation). The state space explosion refers to exponential increase in the state space explored as the model size increases, whereas environment modeling refers to the process of developing a representation for the parts of the system that are not being analyzed. State space explosion problem is generally addressed by better abstractions and also by automated abstraction generation [11]. The environment modeling issues are alleviated by automated generation techniques [24, 46].

Slede decreases the extent of these problems for its intended domain, although it does not completely solve it. The state space explosion problem is addressed by providing a lightweight annotation language that can be used to provide hints to the verification framework. Environment modeling issue is addressed by providing a library of environment models such as for sending and receiving messages, sensing environmental changes, etc.

In order to verify a protocol implementation, besides the source code, Slede requires an abstract specification of the protocol, a deployment topology, and properties that need to be verified about the protocol. The protocol specification is used to automatically generate an intrusion model customized to the protocol. The deployment topology is needed to bound the generated state space. Slede provides an annotation language for this task. In the next subsection, we describe this annotation language.
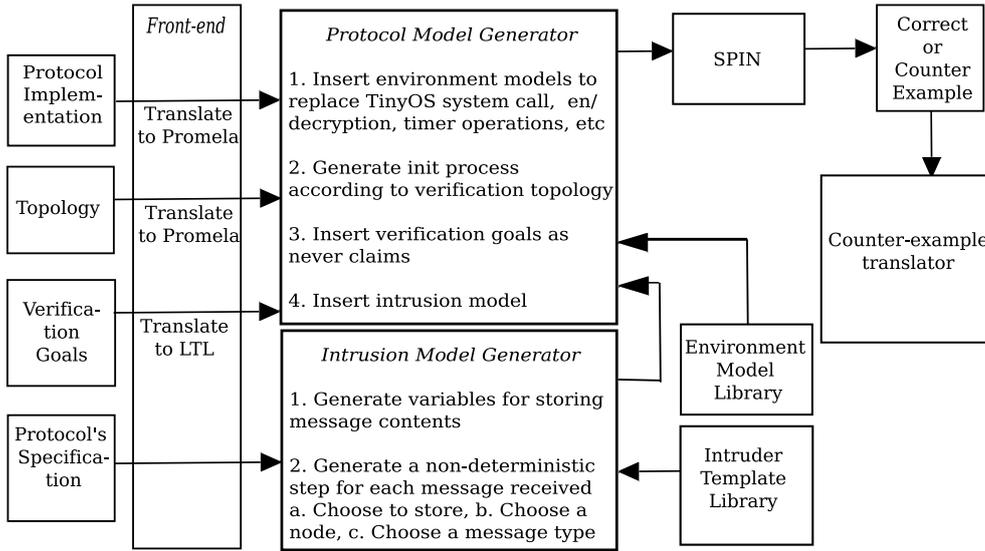
Figure 2: Overview of the Slede Verification Framework

3.2 Annotations in Slede

In this section, we describe the annotation language for Slede. The description includes syntax, small examples, and informal semantics of the constructs.

The abstract syntax of our annotation language is shown in Figure 3. A verification configuration (*V*) consists of a sequence of message declarations (*mdecl**), followed by a sequence of node definitions (*ndecl**), followed by protocol definition (*pdef*), followed by an objective. The objective is the property that is to be verified about the protocol with respect to the specified configuration. We explain message declarations, node definitions, and other pieces of the syntax in the following subsections.

In the concrete syntax, the verification configuration is defined in comments as shown in Figure 4. The annotation comments start and end with an at-sign (@) similar to the annotation comments for the JML specifications [28]. The special words such as **message** are not keywords; they only have special meanings in this context. The verification configuration could also be specified as a command line option for the framework, or as an XML file.

3.2.1 Message declarations

A verification configuration in Slede's annotation language may contain a sequence of message declarations that represent messages exchanged between nodes. A message declaration has exactly one message type (*mtype*) named in the header **message** *mtype*, followed by the keyword **mapsto**, followed by *t*, which is a structure defined in the implementation that is used for exchanging messages between principals. The **mapsto** clause establishes a correspondence between the protocol specification and implementation.

A message declaration may contain a sequence of terms (*term**). A term is a type, followed by a term name, which can be one of the special words $\{sender, receiver, key, nonce, secret, data\}$, followed by the keyword **mapsto**, followed by *f* or a modifier followed by a message type (*mtype*) followed by a message name (*msg*), followed by the keyword **mapsto**, followed by *f*. The *f* is a field in the structure defined in the implementation that is used for exchanging messages between principals. The types of fields that are mapped to each other should be the same.

$V ::= mdecl* \; ndef* \; pdef \; o$
$mdecl ::= \textbf{message} \; mtype \; \textbf{mapsto} \; t \; \{ \; term* \; \}$
$term ::= t \; tname \; \textbf{mapsto} \; f \; ;$
$\qquad | \; modifier \; mtype \; msg \; \textbf{mapsto} \; f \; ;$
$modifier ::= \textbf{private} \; | \; \textbf{public}$
$ndef ::= \textbf{node} \; nt \; n \; \{ \; rdef* \; \}$
$rdef ::= n; \; | \; nt;$
$pdef ::= \textbf{protocol} \; p \; \{ \; mexch* \; \}$
$mexch ::= (nt, nt : mtype)$
$o ::= \textbf{true} \; | \; \textbf{false} \; | \; condition \; | \; ! \; o \; | \; (o) \; | \; o \; \&\& \; o \; | \; o \; `||' \; o$
$\qquad | \; [\,] \, o \; | \; <>o \; | \; o{-}>o \,| \; o<{-}>o$
$condition ::= t \; n.i \; (form*)$
$form ::= t \; var, \; \text{where} \; var \ni \mathcal{N} \cup \mathcal{CT} \cup \mathcal{NT}$

p	$\in$	$\mathcal{P}$, the set of protocol names
m, n	$\in$	$\mathcal{N}$, the set of node names
$mtype$	$\in$	$\mathcal{MT}$, the set of message types
i, j	$\in$	$\mathcal{I}$, the set of event and command names
$tname$	$\in$	$\{sender, receiver, key, nonce, secret, data\}$
msg	$\in$	$\mathcal{M}$, the set of message names
s, t	$\in$	$\mathcal{T}$, the set of types in implementation
e, f	$\in$	$\mathcal{F}$, the set of field names in implementation
mt, nt	$\in$	$\mathcal{NT}$, the set of node types in implementation

Figure 3: Abstract syntax for the core annotation language

The special term names are selected to help during the intrusion model generation. A modifier can be **public** or **private** (with the default value as **public**). It is considered a violation for an intruder to be able to read the private parts of the message.

Note that all terms in *mtype* must have a corresponding field in *t*. More than one *mtype*s can be mapped to one *t*. The set of fields in *t* must be a union of the set of fields in all such *mtype*s.

An example message declaration is given in Figure 4 (lines 11-14), where a message type Ack is defined. This message type in specification is mapped to the structure IntMsg in the implementation. The example defines that a message of this type will contain

```
1  /*@
2  @ message Nonce mapsto MAC {
3  @   int nonce mapsto cipher;
4  @    int key mapsto key;
5  @  }
6  @ message Ping mapsto IntMsg{
7  @   int sender mapsto src;
8  @   int receiver mapsto dest;
9  @   private Nonce data mapsto data;
10 @  }
11 @ message Ack mapsto IntMsg{
12 @   int sender mapsto src;
13 @   int receiver mapsto dest;
14 @  }
15 @ node SensorM A{ }
16 @ node SensorM B{ A; }
17 @ protocol p {
18 @   (SensorM, SensorM, Ping)
19 @   (SensorM, SensorM, Ack)
20 @  }
21 @ A.snd(Ping m)-><>B.rcvd(Ping m)
22 @*/
```

Figure 4: An Example Verification Configuration

two integer fields representing the sender and the receiver of the acknowledgement. These fields are mapped to fields `src` and `dest` of the structure `IntMsg`.

The message declaration `Ping` in Figure 4 is an example of a compound message that is also mapped to the structure `IntMsg` in implementation. This example shows that two different message types may be implemented using the same structure in the implementation. `Ping` contains a **private** field of type `Nonce`. The message type `Nonce` in turn consists of two special integer fields **nonce** and **key**. The way to read this message declaration is "`Ping` message contains **sender** address, **receiver** address, and a **nonce** encrypted with **key**".

Slede's annotation language semantics also follow the same rule. When a message type such as `Ping` contains a **private** field of another message type such as `Nonce` that contains a field **key**, it is assumed that the compound message will have a portion encrypted with the key. Here a message of type `Nonce` will contain **nonce** encrypted with **key**. For the rest of this paper, we will omit the **mapsto** clause, assuming similar structures of messages in specification and implementation.

3.2.2   Node definitions

A verification configuration in Slede may contain one or more node definitions. A node definition has exactly one node type (*nt*) named in the header **node** *nt n*. This node type must be defined in the implementation that is being verified. It may declare several reachability definitions (*rdef**).

A reachability definition is one of the two possible reachability tuples. The simplest kind specifies the concrete node name *n*. This definition means that the node *n* is reachable from the current node. The second kind *nt* does not specify the concrete node name, only its type, which means that any node of type *nt* is reachable from the current node. A simple node definition is presented in Figure 4 (lines 15,16). Here, nodes A and B are defined. The definition of A states that B is reachable.

3.2.3   Protocol definition

A verification configuration in Slede may contain exactly one protocol definition. A protocol definition consists of an ordered sequence of message exchange (*mexch**). A message exchange consists of a node type (*nt*), followed by the node type (*nt*), followed by message type (*mtype*). The first node type specifies the nodes that can send this message and the second node type represents the receiver nodes that can receive. An example protocol definition is shown in Figure 4 (lines 17-20).

3.2.4   Objectives

An objective is a linear temporal logic formula [40] with some additional syntax. An objective can be a literal, or a negated objective (! *o*) enclosed in parenthesis ((*o*)), or opening and closing brackets followed by an objective ([]*o*), which denotes *always*, or opening and closing angular brackets followed by an objective (<>*o*), which denotes *eventually*.

An objective may also consist of two sub-objectives combined by logical and (*o* && *o*), logical or (*o* '||' *o*), implication (*o*->*o*), or equivalence (*o*<->*o*) operators. These operators have the same meanings as their LTL counterparts and they are translated accordingly. Finally, a literal can be **true**, **false** or a *condition*.

A *condition* allow objectives to be expressed in terms of commands and events in the protocol implementation. For example, assume that the sender `src` in Figure 4 declares a command `snd` that takes a parameter of type `Ping`. The receiver `dest` in the same protocol implementation registers an event handler of type `rcvd` with the underlying network layer that also receives a parameter of type `Ping`. A user-defined condition for a trivial property of this implementation is shown in Figure 4 (line 21). The expressed objective means that node A sending a message m implies that node B will receive the same message m eventually.

3.3   Protocol Model Generation

The protocol model generation phase replaces the original code generation phase of the nesC compiler. It translates the protocol implementation as a whole into one PROMELA process that is a global object describing the behavior of the protocol. To generate a finite-state model, we use the following strategies.

☐ **Bounded principals**: We use the verification topology information provided as annotations (see Section 3.2.2) to determine the number of principals involved in the protocol. For each principal (node) involved in the protocol, an instance of the process is instantiated. Following this strategy produces a restricted model, which does not exhibit all behaviors of the protocol implementation; however, it has been shown that even a small version of a system can be very useful for finding errors [26]. Moreover, deployment scenarios for a sensor network application (number of principals, etc) is often known in advance and we expect verification to be performed on these topologies, which further increases the likelyhood of finding errors in the system under verification.

☐ **Abstracting Environment**: During the protocol model generation, all system calls (calls to the libraries of TinyOS) are replaced by calls to environment models provided as Slede's library. This replacement is necessary for two reasons. First, it reduces the size of the model. Second, the TinyOS system calls should not affect the security of the protocol. It is the protocol that we want to verify and not the TinyOS libraries. Our library provides PROMELA models for sending and receiving messages, LED manipulation, etc.

3.3.1   Translating Pointers and Function Symbols

One of the main hurdles in translating a security protocol's implementation to a verification language such as PROMELA is that the protocol code usually includes many constructs that have no equivalents in the verification languages [18]. For example, secure hash functions, encryption and decryption techniques use a lot of

constructs that are far from what current verification techniques offer. Also, pointer arithmetic is common in the nesC implementation of sensor network protocols.

Our solution to this problem is based on a feature provided by SPIN, versions 4.0 and later, that allows embedding C code inside the PROMELA model. The global variables of every nesC module of the protocol are included in the state vector using SPIN's c_state construct, and the statements of the protocol (except for system calls) are embedded one by one as C code using the c_code construct. Thus, no equational theories associated with function symbols nor pointer analysis techniques are required because the operations themselves are included in the model. This solution is efficient when the size of the protocol is small, however, when the protocol size increases, some more abstraction is required. For example, annotations on method calls to encryption/decryption functions may be added similar to Goubault-Larrecq and Parrennes's approach [18]. In the current prototype of Slede, we have not explored these abstraction techniques.

3.3.2 Translating Events

A module in nesC can announce and respond to events (see Section 2). Events introduce concurrency in the language model. Events can be signalled explicitly using the signal construct or by the hardware interrupt. After an event is signalled it executes asynchronously. A common problem with modeling event-driven system is that the target of an event is often not known statically.

The expressive power of the nesC language is deliberately limited by the designers to detect target of a signalled event using a tractable static analysis [16]. Slede uses a similar analysis to translate explicitly signalled events and corresponding event handlers. The target of the signalled event is determined, and the construct signal is replaced by an explicit call to that target.

This solution works for the user implemented events, where the event is signaled from the application, however, for events that are signaled from TinyOS libraries (i.e. event handler for message receiving), this is not the case. Thus, in order to emulate the event-based paradigm, Slede adds a statement to check for pending events between every two statements in the model, where an event handler may be executed [23].

3.4 Intrusion Model Generation

A key challenge in security protocol verification is selection of the right intrusion model. Selecting the right intrusion model can possibly reduce the state space to make the protocol implementations amenable to automatic analysis. On the other hand, selecting an overly general model can create an infinite state space, which may not be tackled by finite-state model checkers. Moreover, sensor network researchers and developers are not model checking experts, which makes it harder for them to write effective finite-state intrusion models in languages such as PROMELA.

Slede automates this process. It generates a Dolev-Yao style intrusion model [13] from the protocol specification. An intruder in this model can intercept all messages and modify their contents, but cannot read an encrypted message unless it has the encryption key.

The generated intrusion model is aware of protocol's message structure and message sequencing, which are almost always available, therefore it generates lesser state compared to a naïve intrusion model, but more states compared to a model that is highly customized by a model checking expert. On the other hand, the generated model does not require specialized efforts from Slede users.

An intrusion model is generated as a separate PROMELA process that describes the behavior of the intruder. The intruder can

```
1  bit knowsNonceA;
2  active proctype Intruder() {
3    Crypt data, saved;
4    mtype msg;

6    do
7    :: network? msg,_,data -> /* Msg Intercepted */
8       if /* perhaps store the message */
9       :: saved = data;
10      :: skip;
11      fi;

13   :: /* Replay or send message */
14      if /* choosing message type */
15      :: msg = msg1;
16      ...
17      fi;
18      if /* choosing recipient */
19      :: recipient = agentA;
20      ...
21      fi;
22      if /* replay saved msg or assemble it */
23      :: data = saved;
24      :: if
25         :: knowsNonceA -> data.info = nonceA;
26         :: data.info = 0;
27         fi;
28      fi;
29      network ! (msg,rcpt,data);
30   od;
31 }
```

Figure 5: Intruder Model Pattern

intercept a message, save it, forward it, discard it or create a new message to send. Thus, the intrusion models that Slede generates also serve to model dynamic topological changes. For example, a broken communication link between two sensor nodes is modeled by a non-deterministic choice in the intrusion model, where the received message is dropped.

The pattern used for the intruder model generation is based on the intruder model described in [34]. The structure of the intruder is shown in Figure 5. There is one loop (lines 6-30) where the intruder either intercept the message (line 9), or send either the intercepted message (line 23) or a new one (lines 24-27). The intruder is allowed to save only one message, and knows about the the structure of the messages sent. These constraints on the intruder model helps prevent the undecidability problem known to accompany the verification of cryptographic protocols [14].

Intruder model acquires knowledge about the protocol from both the verification configuration and the nesC implementation of the protocol. The structure of the message in every message sequence is described in the verification configuration, which allows the intruder to know what to expect in a message when it is received. For example, in the specification shown previously in Figure 4, when the intruder receives a message of type Ping, it knows that there is a nonce that is encrypted (private), and it cannot open it without a key. Yet, the sender and receiver of the Ping messages are public.

The intruder maintains a variable for every private member of every message structure. These variables can be used in order to describe objectives. For instance, objective that the intruder doesn't know the nonce in the example verification setup will be described as {B.rcvd(Ping m) -><>!Intruder.knowsNonce}.

3.5 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 [22], which verifies whether the model violates the objectives which have been translated into LTL formulas [40]. 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.

3.5.1 Translation of Objectives

Objectives in Slede are represented using command calls, event signals, and intruder's state. For every command, event or intruder state, which is of interest, a boolean variable is maintained. These variables are initialized to false. On entering a command or event handler of interest, the corresponding variable is set. As for the objective itself, it is translated into a linear temporal logic (LTL) formula.

4. EVALUATION

In this section, we evaluate Slede. First we describe a comparison of the intrusion model generation technique of Slede with a hand-written model and with a generic intrusion model. We then describe verification of two sensor network security protocol implementations using Slede. For both protocols, Slede 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.

4.1 Intrusion Model Generation Technique

In this section, we evaluate the state space generated by Slede with the state space generated using a general intrusion model that is not aware of protocol structure and an intrusion model written by a model checking expert. We selected the Needham Schroeder's protocol [37] for this experiment.

The objective of this protocol is to exchange two secret numbers between two principals. Figure 6 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 6: Needham-Schroeder Protocol

The Needham-Schroeder protocol has a known flaw. If a malicious node I communicating with node A impersonates A and establishes 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 that the secret numbers should only be known to the two nodes.

The selection of this protocol for evaluation was primarily because a number of hand-written intrusion models are available. As far as we are aware of, any intruder model for a sensor network security protocol is not available in the current literature. The intruder model was taken from Maggi and Sisto's work on model checking cryptographic protocols [31].

The verification configuration for Slede's version is shown in Figure 7. Three compound message types EncryptedNonce, EncryptedNonceNode, and EncryptedNonceNonce are declared. A message of type EncryptedNonce contains **sender** address, **receiver** address, and a **nonce** encrypted with **key**. A message of type EncryptedNonceNode contains **sender** address, **receiver** address, and a **nonce** and the **sender** address encrypted with **key**, and a message of type EncryptedNonceNonce contains **sender** address, **receiver** address, and two **nonce**s encrypted with **key**.

```
1  /*@
2  @ message Nonce {
3  @   int nonce; int key;
4  @ }
5  @ message EncryptedNonce{
6  @   int sender; int receiver;
7  @   private Nonce data;
8  @ }
9  @ message NonceNode{
10 @   int nonce; int sender; int key;
11 @ }
12 @ message EncryptedNonceNode {
13 @   int sender; int receiver;
14 @   private NonceNode data;
15 @ }
16 @ message NonceNonce{
17 @   int nonce; int nonce; int key;
18 @ }
19 @ message EncryptedNonceNonce {
20 @   int sender; int receiver;
21 @   private NonceNonce data
22 @ }
23 @ node NeedhamM A{ }
24 @ node NeedhamM B{ A; }
25 @ protocol Needham {
26 @   (NeedhamM, NeedhamM, EncryptedNonceNode)
27 @   (NeedhamM, NeedhamM, EncryptedNonceNonce)
28 @   (NeedhamM, NeedhamM, EncryptedNonce)
29 @ }
30 @ [] ((Leds.greenOn()) ->
31 @ (<> (!Intruder.knowNonceA || !Intruder.knowNonceB)))
32 @*/
```

Figure 7: Verification Setup for Needham Schroeder's protocol [37]

This verification configuration contains only two nodes A and B. Both these nodes run the code for the NeedhamM module in the implementation. The **protocol** Needham is defined in terms of declared message types and node types and it represents the sequence of messages exchanged between two symmetric principals. The objective in this configuration states that when both the nodes share the nonces (representing by the turning on the green LEDs in the implementation) then the intruder does not know either nonces.

Figure 8 shows the states explored, transitions, and memory taken by the same version of SPIN for three different intrusion models. All experiments were run on the same machine. All three intrusion models were trying to find flaws in the same protocol implementation.

The second column contains the results for the intrusion model

	Hand-written intrusion model	Slede generated intrusion model	Generic intrusion model
States	692	1647	5000000
States %	42.02	100	303582.27
Transitions	5536	13082	7000000
Transitions %	42.32	100	53508.64
Memory (MB)	2.83	3.24	1800
Memory %	87.36	100	55624.23
Time	fraction of second	fraction of second	120 seconds and more

Figure 8: Comparsion of hand-written, generated, and generic intrusion models for Needham Schroeder Protocol

described by Maggi and Sisto [31]. The third column shows the results for the intrusion model generated using the protocol specification provided as annotations to Slede, and the fourth column shows the results for a generic intrusion model. As can be observed, the hand-written intrusion model generates 58% fewer states and transitions but only consumes only about 12% less memory.

These results demonstrate that for a small extra cost, our technique for automatically generating intrusion models has the potential to make the formal verification techniques based on finite-state model checking more accessible to sensor network security researchers and developers, who may not be model checking experts. Our intrusion model generation technique also has room for improvements, which is likely to lower the gap between a hand-written model and an auto-generated intrusion model even further.

4.2 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. [50]. 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 ith 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 [50] 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 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 has obtained KC by compromising A [50].

We verified an implementation of this protocol with respect to a property informally stated as follows: "when nodes share the secret, the intruder shouldn't know about the secret." The verification setup including the verified property is shown in Figure 9 using our annotation language. Like Needham-Schroeder's protocol new message types are declared, nodes are defined, protocol sequence is defined, and finally protocol objectives are given in terms of the state of the implementation and the state of the automatically generated intrusion model.

```
1  /*@
2  @ message KeyMsg{
3  @   int sender; int receiver;
4  @   int key;
5  @ }
6  @ message Mac{
7  @   int data;
8  @   int key;
9  @ }
10 @ message DataMsg{
11 @   int sender; int receiver;
12 @   int data;
13 @   private Mac MAC;
14 @ }
15 @ node TeslaM A{ }
16 @ node TeslaM B{ A; }
17 @ protocol p {
18 @   (TeslaM, TeslaM, KeyMsg)
19 @   (TeslaM, TeslaM, DataMsg)
20 @   (TeslaM, TeslaM, KeyMsg)
21 @ }
22 @ [] ((Leds.greenOn()) -> (<> (!Intruder.knowsdata)))
23 @*/
```

Figure 9: Verification Configuration for One-way Key Chain Based One-hop Broadcast Authentication Scheme [50]

Our approach was able to detect this attack. In order to detect this flaw using the model generated by Slede, SPIN produced 239933 states, made 1.9 million transitions, consumed 118.129 MB memory and took 17 seconds to detect the problem.

4.3 Verifying the Probabilistic Based Pairwise Key Establishment protocol

The probabilistic based pairwise key establishment protocol [49] includes two phases: system initialization before network deployment and pairwise key establishment after deployment. Before the deployment of a network, i.e., during a *key pre-distribution* phase, every node is loaded with a small fraction of keys out of a large pool of keys by a key server. The allocation of the keys to each node is done using a probabilistic scheme, which enables every pair of nodes to share one or more keys with certain probability. If two nodes share keys, then they can use these keys to encrypt messages between them. If no direct keys are shared between the two nodes, then the probabilistic key sharing scheme enables them to communicate securely using logical paths obtained via a logical path discovery. However, if the two nodes share keys, they cannot use these keys directly in their communication because these keys are not known exclusively to these two nodes, and there is a possibility these keys are allocated in some other nodes as well. Thus these two nodes need to establish a new key that is shared only between them.

In case of establishing a key between two nodes *u* and *v* sharing keys, the protocol works as follows: Node *u* creates a secret key *S*. Let R_{uv} be the keys shared between *u* and *v*. To send the key

S, node u divides it into shares of equivalent sizes. To deliver each share, node u computes the XORed key $k_{enc} = \text{XOR } k_i, \forall k_i \in R_{uv}$, then encrypts each share with k_{enc}.

```
1 /*@
2  @ message Mac{
3  @  int data;
4  @  int nonce;
5  @  int key;
6  @  }
7  @ message SecretMsg{
8  @  int sender; int receiver;
9  @  int data;
10 @  private Mac MAC;
11 @  }
12 @ message Nonce{
13 @  int nonce;
14 @  int key;
15 @ }
16 @ message EncryptedNonce{
17 @  int sender; int receiver;
18 @  private Nonce data;
19 @  }
20 @ node ZhuM A{ }
21 @ node ZhuM B{ A; }
22 @ protocol p {
23 @  (ZhuM, ZhuM, SecretMsg)
24 @  (ZhuM, ZhuM, EncryptedNonce)
25 @ }
26 @ [] ((Leds.greenOn()) -> (<> (!Intruder.knowsdata)))
27 @*/
```

Figure 10: Verification Configuration for Probabilistic Based Pairwise Key Establishment protocol [49]

After extracting the model from the protocol implementation, implementing an intruder model that behaves as a node that's a part of the protocol and has been compromised, the verification resulted in a scenario where the secret key *S* will be known to the compromised node. The problem is that if the compromised node had a set of keys that when XORed will give the same key as k_{enc}, then the compromised node can intercept the message communication between the two nodes, decrypt the message sent, save the content and then encrypt it and send it again to the intended destination. They keys of the nodes in the verification were 9 and 10, and the compromised node had two keys of value 5 and 6, where the two pairs have the same value when XORed. The probability of having such a collision in the key when the network is being deployed has been mentioned in [49], and though a small probability, SPIN was able to detect it from the extracted model. The verification setup including the verified property is shown in Figure 10 using our annotation language. SPIN produced 2040 states, made 603050 transitions, consumed 35.33 MB memory and took 3 seconds to detect the problem.

5. RELATED WORK

The closest work related to our approach is by Bhargavan *et al.* [3] and by Goubault-Larrecq and Parrennes [18]. Bhargavan *et al.* [3] present an approach for verifying protocol implementations written in F# using ProVerif [4], 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 symbolic model checker as the underlying technology.

Goubault-Larrecq and Parrennes [18] 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 [18] could be used to enhance the underlying verification technique for Slede.

Besides these two 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 [9], 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 [47].

Meadows developed the NRL protocol analyzer for the analysis of cryptographic protocols [33]. The NRL protocol analyzer was used to find flaws in a number of cryptographic protocols including selective broadcast protocol by Simmons [44], Resource Sharing Protocol by Burns and Mitchell [8], re-authentication protocol by Neuman and Stubblebine [39], etc. Longley and Rigby also developed a tool and demonstrated a flaw in a banking security protocol [30]. Yet another tool was Interrogater developed by Millen *et al.* [35]. Kemmerer [27] used general-purpose formal methods technique as tools to verify cryptographic protocols. Schneider adapted the CSP model for verification of security protocols [43]. For a detailed summary of verification techniques, please refer to a survey by Rubin and Honeyman [42], Meadows [32], Gritzalis *et al.* [45], and a more recent survey by Buttyan [10].

Tools for model checking source code directly are also related to this work. In particular, Blast [21], Bandera [12], Java Path Finder [20], CMC [36], etc, have successfully verified C and Java implementations. Like these approaches, Slede also verifies source code directly; however, unlike these techniques Slede is highly customized towards model checking security protocol implementations in nesC.

6. 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.

Another area of exploration is to design an intermediate language similar in style to Action language [7], BIR for Bandera [12], etc, for Slede. Developing this intermediate language representation, and translating nesC programs will make it easier to incorporate new model checkers as the backend of Slede.

We also plan to improve on our current prototype. The current implementation of our verification framework 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. A limitation is on the number of participant nodes in the verification process. We model a node as a separate process in PROMELA. The number of these processes are limited, which limits the number of nodes in Slede's topology.

There is a large body of work on parameterized model checking such as by Bouajjani *et al.* [5], Clarke *et al.* [11], Emerson and Namjoshi [15], Henzinger *et al.* [21], etc that has shown to that it is possible to scale model-checking to infinite-state systems. Recent work on BLAST [21] and YASM [19] has shown the viability of tools that utilize these techniques. In our current prototype, we have not considered using these abstraction techniques; however, our future work will include investigations along these directions.

7. CONCLUSION

In this work, we presented *Slede* our verification framework for sensor network protocol implementations. The key advantages of Slede is that it automatically extracts verifiable models from nesC implementations and allows automatic generation of protocol specific intrusion models from lightweight annotations. We used Slede to verify Needham Schroeder's protocol, and two sensor network specific protocols. We were able to confirm their flaws using an automatically generated intrusion model.

Our approach is sound but incomplete in that all traces provided are real flaws in the implementation with respect to the specified objectives, but not all flaws in the implementation are provided as traces. Nevertheless, our experience with verifying some protocols appears to indicate that Slede can be useful for finding cryptographic errors in sensor network security protocols.

Security in sensor networks is an important problem. By bringing the advantages of explicit-state model checking to the nesC language that is used in a number of wireless sensor network applications, Slede paves the way to improve the security of these applications at a small cost.

8. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 403–418, London, UK, 2000. Springer-Verlag.

[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] T. Bultan. Action language: a specification language for model checking reactive systems. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 335–344, New York, NY, USA, 2000. ACM Press.

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

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

[10] 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.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[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] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In N. Heintze and E. Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP, Trento, Italy*, July 1999.

[15] E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, page 70, Washington, DC, USA, 1998. IEEE Computer Society.

[16] 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.

[17] 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.

[18] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In R. Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379, Paris, France, Jan. 2005. Springer.

[19] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A software

model-checker for verification and refutation. In *CAV '06: Proceedings of the 18th International Conference on Computer Aided Verification*, pages 170–174, August 2006.

[20] 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.

[21] 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.

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

[23] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 597–607, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[24] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *ISSTA '07: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, page To appear, New York, NY, USA, July 2007. ACM Press.

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

[26] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7):484–495, 1996.

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

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

[29] 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.

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

[31] P. Maggi and R. Sisto. Using spin to verify security properties of cryptographic protocols. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 187–204, London, UK, 2002. Springer-Verlag.

[32] 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.

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

[34] S. Merz. Model checking: A tutorial overview. In F. C. et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Berlin, 2001.

[35] J. K. Millen, S. C. Clark, and S. B. Freeman. The

interrogator: protocol security analysis. *IEEE Trans. Softw. Eng.*, 13(2):274–288, 1987.

[36] 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.

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

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

[39] 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.

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

[41] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press.

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

[43] 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.

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

[45] 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.

[46] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated environment generation for software model checking. In *ASE '03: Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, page 116, 2003.

[47] 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.

[48] S. Zhu, S. Xu, S. Setia, and S. Jajodia. Establishing pairwise keys for secure communication in ad hoc networks: A probabilistic approach. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, page 326, Washington, DC, USA, 2003. IEEE Computer Society.

[49] S. Zhu, S. Xu, S. Setia, and S. Jajodia. Establishing pairwise keys for secure communication in ad hoc networks: A probabilistic approach. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, page 326, Washington, DC, USA, 2003. IEEE Computer Society.

[50] 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.