

# Manual for Slede Annotation Language

Youssef Hanna  
Iowa State University  
ywhanna@cs.iastate.edu

## Abstract

*Verifying sensor network security protocol implementations using testing/simulation might leave some flaws undetected. Formal verification techniques have been very successful in detecting faults in security protocol specifications; however, they generally require building a formal description (model) of the protocol. Building accurate models is hard, thus hindering the application of formal verification. In this work, a framework for automating formal verification of sensor network security protocols is presented. The framework Slede extracts models from protocol implementations and verifies them against generated intruder models. Slede was evaluated by verifying two sensor network security protocol implementations. Security flaws in both protocols were detected.*

## 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 [13], 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 [4] 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 [5, 21, 16].

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 [15] 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 [7] 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 [2, 9].

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 [11]) 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 [10].
- The level of knowledge and effort required may prevent many domain experts from attempting such task [20].
- Such model may abstract many potentially troublesome and error prone details of implementation code that are more likely to contain bugs [3].
- 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 [3].

This work addresses these problems. We present our framework *Slede* for automatic formal verification of sensor network security protocol implementations written in nesC [8]. *Slede* 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.

To evaluate *Slede* 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 [22]. The second protocol is

$\mu$ Tesla protocol [18]. The intrusion model used for verifying these protocols is the Dolev-Yao model [6]; however, our framework can be easily extended to allow other intrusion models to be used. Our approach confirmed known flaws in both these protocols.

## 2 Verification Framework

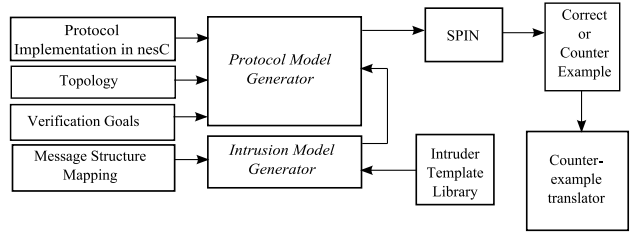


Figure 1. Overview of *Slede*

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 [17] and uses the Spin model checker [12] as the backend.

The overview of *Slede* is shown in Figure 1. 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 [12]. 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 annotation language required for protocol verification from implementation.

### 2.1 Annotations in Slede

Verification of security protocols requires the presence of a principal that acts maliciously 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

```

V ::= mdecl* tmap iter? ndef obj;
mdecl ::= message mtype mapsto t { term* }
term ::= private? tname mapsto f;
tmap ::= msgtypes mapsto t.f { mmap* }
mmap ::= d:mtype;
iter ::= iteration:d;
ndef ::= node n { rdef* }
rdef ::= n;
obj ::= objective oname { (o)* }
o ::= condition | ! o | (o) | o && o | o '||' o | o->o
condition ::= (p:)? mname.iname.fname (form*)
form ::= mtype | d

```

where

- $m, n \in \mathcal{N}$ , the set of node names
- $p \in \mathcal{N} \cup \{\text{Intruder}\}$
- $mtype \in \mathcal{MT}$ , the set of message types
- $tname \in \{\text{sender}, \text{receiver}, \text{data}\}$
- $e, f \in \mathcal{F}$ , the set of field names in implementation
- $c, d \in \mathcal{I}$ , the set of integers
- $oname \in \mathcal{O}$ , the set of objective names
- $mname \in \mathcal{MN}$ , the set of module names in implementation
- $iname \in \mathcal{IN}$ , the set of interface names in implementation
- $fname \in \mathcal{FN}$ , the set of function (command/event) names in implementation

**Figure 2. Abstract Syntax for the Core Annotation Language**

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 to convey such information to *Slede* to generate the malicious behavior, we developed an annotation language for describing such information required by our framework. This language is also used to describe the security goals against which the protocol should be verified as well as some information required to bound the verification process such as the network topology and the number of iterations of the protocol.

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

**Abstract Syntax:** The abstract syntax of our annotation language is shown in Figure 2. A verification configuration ( $V$ ) consists of a sequence of message declarations ( $mdecl^*$ ), followed by the message type mapping ( $tmap$ ), followed by the bound on the number of iterations of protocol execution ( $iter$ ), followed by a sequence of node definitions ( $ndef$ ) that specifies the topology, followed by the set of objectives ( $obj$ ). The objectives are the properties that are to be verified about the protocol with respect to the specified configuration. We explain message declarations and other pieces of the syntax in the following subsections.

In the concrete syntax, the verification configuration is defined in the file of extension `sld` as shown in Figure 3.

```

1 message Data mapsto IntMsg{
2   sender mapsto src;
3   receiver mapsto dest;
4   data mapsto info;
5   private data mapsto authen;
6 }
7 message Ack mapsto IntMsg{
8   sender mapsto src;
9   receiver mapsto dest;
10 }
11 msgtypes mapsto IntMsg.type {
12   1: Data;
13   2: Ack;
14 }
15 node 0 { 1; }
16 node 1 { 0; }
17 objective Auth{
18   Intruder:SensorM.Send.send(Data)->
19   !SensorM.Send.send(Ack)
20 }

```

**Figure 3. Example Verification Configuration**

## 2.2 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**, 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.

To represent which fields in the message are sender, receiver or data, a message declaration may contain a sequence of terms ( $term^*$ ) responsible for mapping the special words  $\{\text{sender}, \text{receiver}, \text{data}\}$  to fields  $f$  of the message structure  $t$  in the code using the **mapsto** keyword.

An example message declaration is given in Figure 3 (lines 7-10), where a message type `Ack` is defined. This message type in the specification is mapped to the structure `IntMsg` in the implementation. The example defines that a `Ack` message contains two 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 `Data` in Figure 3 (lines 1-6) is, similar to `Ack`, 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. We show in Section 2.3 how to distinguish different message types that are mapped to the same message structure in the implementation. `Data` contains a **private** field `authen` (line 5), meaning that this field is a MAC. Therefore, if the intruder intercepts this message, it cannot modify this field unless it has the key. As for other fields not preceded by **private**, like `src` (line 2), the intruder can read and/or modify on them without any keys

For the generated intruder to launch attacks such as forwarding the message to a different node or trying to modify

the MAC of the message, it now can understand which fields of the message structure represent sender, receiver or data.

### 2.3 Differentiating between Message Types

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. 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 `msgtypes` special word, followed by `mapsto`, followed by `t`, which is a structure defined in the implementation that is used for exchanging messages between principals, followed by the field `f` responsible for identifying the type of messages, followed by the mapping (`mmap`) that maps values of the field `f` to the message types `mtypes`.

In Figure 3, we can see that both message declarations `Data` and `Ack` are mapped to the same structure `IntMsg`. The field `type` of the structure `IntMsg` (lines 11-14) is the field responsible for identifying the type of the message (where the value of `type` is 1 for `Data` or 2 for `Ack`).

### 2.4 Topology

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. The number of states is directly proportional to the possible executions of the protocol.

To avoid the problem of state explosion, our current prototype allows for verifying the protocol with one topology at a time. This way, we reduce the execution possibilities (as opposed to checking that every broadcast message was received by all sensors if verification is done against all possible topologies), therefore the size of the model is reduced. This is a limitation in our current prototype that we plan to solve in the next version of *Slede*.

To define the topology, every node `n` is declared using the special word `node`, followed by `rdef*`, which defines the reachable nodes to `n`.

In lines 15-16 of Figure 3, 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 0 and 1. Our framework allows the intruder to listen

to all the wireless channels, thus there is no need to include the intruder in the topology.

### 2.5 Number of Iterations

The sensor network security protocols are usually intended to run for as long as the resource constraints of the nodes can handle. In other words, most of the protocols (and sensor network applications in general) tend to put no bound on the number of executions of the protocol, leaving the protocol to run as much as the nodes can survive.

While it is possible to implement such behavior, verifying such infinite systems is hard in model checking. The main reason is that infinite executions will lead to an infinite model, thus leading to state explosion. Therefore, a bound is required to make the model finite.

We chose to make the bound as the number of firings the node's timer is triggering. The user should specify how many timer firings represent one iteration of the protocol execution. To define the timer bound, the user uses the special word `iteration`. In case there is no timer used in the protocol implementation, this bound is not necessary and should not be specified. For instance, in the example in the figure, there was no timer used so we did not need to add the `iteration`.

### 2.6 Objectives

An objective is a linear temporal logic formula (LTL) [19] 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`).

An objective may also consist of two sub-objectives combined by logical and (`o && o`), logical or (`o '||' o`) or implication (`o->o`). These operators have the same meanings as their LTL counterparts and they are translated accordingly.

A *condition* allows objectives to be expressed in terms of commands and events in the protocol implementation. A *condition* can either describe an action done by an intruder or a legitimate node. If the action is done by an intruder, then this is represented using the `Intruder` special word, otherwise it is a legitimate node, so the *condition* starts directly with the module name where the function (command/event) is implemented (`mname`). This is followed by the interface name of the function (`iname`), followed by the function name (`fname`) that may take arguments (`form`). The arguments can be either a normal argument, or in the case that the command is for sending, it can be one of the message types declared in the annotation (`mtype`).

In lines 17-20 of Figure 3, the objective named `Auth` is that if the intruder sends a message of type `Data`, then

any legitimate node should not send a message of type `Ack` in return. Any message sent from the intruder is assumed to be corrupt (i.e. if the intruder is just forwarding the message `Data` without modifying its content, then the protocol is safe and there is no violation of the property). The intruder is described using the **Intruder** special key word. The intruder can use the functions used by legitimate users for sending as shown. For specifying legitimate node activity (line 19), we directly write the module name `SensorM` where the function of sending was called, followed by the interface of sending `Send`, followed by the command `send`. Note that the parameters can be of the message types declared in the annotation language (in this example messages were declared in lines 1 and 7).

Even though our framework currently supports the generation of intruder models according to Dolev-Yao model, our framework can easily be extended to provide generation of other intruder models such as node capture attack according to different patterns. As Figure 1 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.

## 2.7 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 [12], 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.

## References

- [1] I. Akyildiz, W. Su, Y. Sankarasubramanian, 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] 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.
- [5] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.
- [6] 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.
- [7] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
- [12] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
- [13] C. Intanagonwivat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication. *MobiCOM '00*, August 2000.
- [14] 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.
- [15] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SensSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.
- [16] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
- [17] nesC Compiler. <http://sourceforge.net/projects/nesc>.
- [18] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. Tygar. Spins: security protocols for sensor networks. In *Proceedings of ACM Mobile Computing and Networking (Mobicom'01)*, pages 189–199, 2001.
- [19] 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.
- [20] 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.
- [21] G. J. Simmons. How to (selectively) broadcast a secret. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 108, 1985.
- [22] 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.