

# SenSec: A Scalable and Accurate Framework for Wireless Sensor Network Security Evaluation

Yi-Tao Wang

Department of Computer Science  
University of California, Los Angeles  
Email: yitao@cs.ucla.edu

Rajive Bagrodia

Department of Computer Science  
University of California, Los Angeles  
Email: rajive@cs.ucla.edu

**Abstract**—Developing secure wireless sensor networks (WSNs) is a complex process that involves careful design of attack test cases and security countermeasures, as well as meaningful evaluation of the impact of the attack and performance of the countermeasure. In this paper, we present the design and implementation of SenSec, a scalable framework that facilitates the development and evaluation of secure WSNs, protocols, and applications. We also demonstrate key benefits of SenSec: (1) it allows security evaluation of real sensor applications, (2) it can be easily extended to any real application simulator, (3) it provides a modular structure for defining attack cases and supports extensibility to a broad set of attacks, (4) it supports automatic generation of sophisticated and previously unconsidered attack cases, and (5) it facilitates the analysis and identification of vulnerabilities in security systems and a quantitative evaluation of the impact of attacks and countermeasures in WSNs.

**Index Terms**—SenSec, Security, Simulator, Sensor Networks

## I. INTRODUCTION

The use of wireless sensor networks (WSNs) is expanding rapidly in a variety of fields. As the use of sensors evolves from merely capturing data to real-time event detection and response, sensor networks must provide accurate and authentic data to guarantee operational integrity. Security solutions are necessary to prevent malicious attacks on sensor networks. With large-scale deployments of motes running a diverse set of applications, it is beneficial to first evaluate and validate the design and performance of secure sensor networks through simulation prior to deployment.

Furthermore, WSN security should be evaluated in realistic environments. Ideally, we want the actual application and security systems to run in the same condition in the simulation as in actual deployment to minimize discrepancy between testing and deployment. When the attack test cases are evaluated, the behavior of the WSN security system can be observed and studied under realistic attacks. Furthermore, meaningful measurement and evaluation can be conducted to perform qualitative and quantitative assessments.

Our past experience has also taught us that this process is non-trivial and time-consuming, and it is desirable to have an evaluation environment to automate and facilitate the testing process. Although there are some such tools available for wired networks, little work has been done in the WSN domain and, to the best of our knowledge, no such WSN security evaluation systems exist in the open literature.

In this paper, we present SenSec, a framework for evaluating secure sensor networks that together satisfies all of the following criteria:

- **Heterogeneity:** SenSec provides a diverse set of accurate models so that simulations closely reflect the conditions of real-world deployments. It supports complex scenarios involving complex attacks. Most importantly, it offers an extensible system for defining and composing attacks that can enhance the simulation with variations of defined attack scenarios that developers did not initially anticipate.
- **Fidelity:** SenSec accurately captures the behavior of sensor motes under attack. This requirement involves not only providing an extensible system for defining attack cases but also emulating real sensor applications. By emulation, we mean that SenSec executes real implementations of the application rather than using models of their behavior. While using models for sensor motes offers better performance and allows the evaluation of high-level network interactions, only software emulation will allow us to evaluate the specific impact of attacks on real applications.
- **Scalability:** As use of WSNs continues to expand, the size of deployed networks will increase. A framework that is able to efficiently simulate large networks for long simulation horizons is necessary to meet future requirements.

Our primary goal in designing and implementing the SenSec framework is to facilitate research in sensor network security by providing an environment to evaluate complex WSN security scenarios that has, thus far, been absent. The framework separates the attack simulation from the network simulation and application development which provides it support for and adaptability to any discrete event simulator. Furthermore, because it runs real applications that will be deployed on sensor motes, it can detect security flaws resulting from implementation choices or bugs.

## II. THE SENSEC FRAMEWORK

We have designed and implemented a framework to facilitate security development and evaluation in WSNs that meets the above goals. This section focuses on the design of the SenSec framework. The details of our implementation are discussed in Section III.

The SenSec architecture, shown in Figure 1, includes the following components:

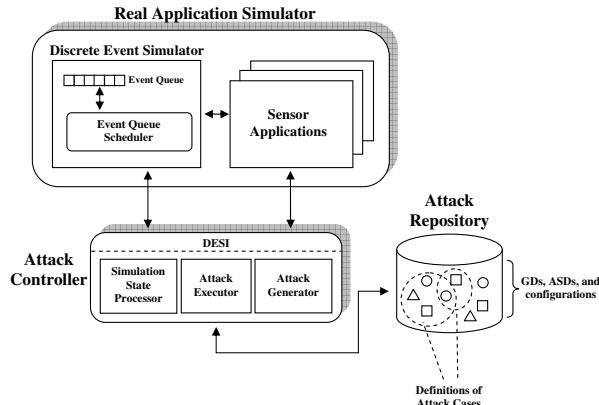


Fig. 1. SenSec's architecture

- **Real Application Simulator (RAS):** A number of simulators exist that can execute application code in simulated environments [4], [2], [19], [18]. Once the application has been evaluated, they are then deployed, without modification, on real networks. We refer to these as real application simulators (RAS). To meet our goal of evaluating real applications, we made an early design decision to leverage an existing RAS, specifically a TinyOS (an OS for motes) [16] simulator, rather than starting from scratch. Our implementation used TiQ [19], a scalable framework for executing TinyOS applications in any discrete event simulator. Although we chose to use TiQ, due to our familiarity with it, the SenSec framework is designed to be composable with any RAS, such that the simulator can be extended to also evaluate security-related issues.

The TinyOS simulator, and RASs in general, consist of two parts: the discrete event simulator (DES) and the TinyOS application. The DES is responsible for providing simulation of the MAC and physical layers. Most network simulators already provide extensive libraries of MAC protocols and physical layer models [8], [21], which negates the need for us to reimplement these models in a new simulator. TiQ uses the QualNet [11] network simulator as the DES.

The TinyOS applications are unmodified code that runs in SenSec for each node in the simulated WSN. This functionality provides more realistic scenarios, involving real applications, and support for application-level evaluations.

- **Attack Repository (AR):** The Attack Repository is a system for defining attack cases. Its structure allows subtasks to be defined as separate components. Those components can be composed to define sophisticated attacks. Section II-A discusses this in more detail.
- **Attack Controller (AC):** The Attack Controller is responsible for executing the attacks in simulations. It separates the simulator from the AR. This separation allows attack cases defined in the AR to be ported, without modification, to any simulator that implements the AC. The AC consists of the Attack Generator (AG), Simulation State Processor (SSP), Attack Executor (AE), and DES Interface (DESI). The AG is responsible for automatically

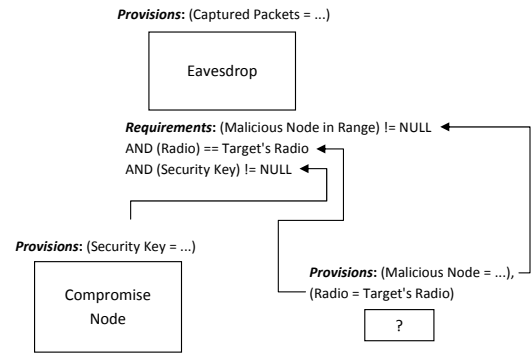


Fig. 2. Sample components with their requirements and provisions: Items in boxes are components and those in parenthesis are CONDs.

generating possible attacks from components defined in the AR. The SSP and AE are responsible for launching attacks in the simulation. The DESI is a standardized set of interactions between the simulator and SenSec. The DESI allows SenSec to be extended to any simulator running real sensor applications with only changes to support its interactions. The AC is detailed in Section II-C.

#### A. Attack Repository

The Attack Repository is a system for defining attack cases in a modular structure to assist researchers in leveraging existing definitions to create new ones. Attacks can be separated into subtasks, each defined in a separate component. The components can then be composed to define sophisticated attacks.

1) *Components:* Components in the definition of an attack represent abstract subtasks in an attack scenario, such as gaining control of a node or figuring out the network's security key. Figure 2 illustrates components in SenSec. In general, a subtask requires certain conditions to be met before it can be performed. For example, to eavesdrop on an encrypted network, (1) a malicious node needs to be in range of the target, (2) its radio must be capable of receiving transmissions from the target, and (3) it must know the security key used to decrypt packets. Similarly, subtasks can provide results that can be used to enable other subtasks. For example, compromising a node in the network will allow us to extract the shared security key used by the network. The security key gained from the compromise can then be used to eavesdrop on other nodes. To represent these concepts and relationships, we introduced CONDs.

CONDs are the required information or situation for that particular subtask of the attack to occur, such as (1), (2), and (3) in the example above. They can represent details, such as operating system used, or abstract concepts, such as whether an insider attack is possible. Formally, they are semantically typed attributes.

A component may have *requirements*, a boolean expression of CONDs that must be met in order for a component to be a valid part of an attack definition and *provisions*, a set of CONDs that exist after the represented subtask executes. The CONDs from one component's provisions can be used to meet another component's requirements and allow us to compose these components to define attacks. Section II-A2 discusses the composition in greater detail.

We further divide components into: Generalized Definitions (GDs), Application-Specific Definitions (ASDs), and configurations that specify attacks composed of GDs and ASDs. GDs specify subtasks of attacks that do not depend on the application used in the simulation (e.g., replaying a spoofed packet).

In contrast, ASD are application-specific subtasks. ASDs are optional, but support simulations that focus on specific scenarios with knowledge about the network without simulating the acquisition of that knowledge. For example, it allows us to get the security key from a variable in the sensor application without having to simulate the process of eavesdropping on the target and cracking of the key from the overheard transmissions. Unlike GDs, which can have no requirements, an ASD must always require its corresponding application.

Configurations are user-defined compositions that represent specific attack cases. Like other components, these can be composed to form other, more complex, attack cases if their requirements are satisfied.

2) *Composing Components*: Using the CONDs in components, it is straightforward to compose them to define attack cases. The CONDs provided by one component are available to meet the requirements of other components. Therefore, the composition that defines an attack can be represented as a mapping where each node is a component and each link is a mapping between provided CONDs and required CONDs. A composition is valid if there are no unsatisfied requirements.

It should be noted that the provisions of a task are not guaranteed to be true when it completes execution. SenSec's framework for composing components is only a means of ensuring that defined attacks are valid for simulation with a given WSN. Whether or not the possible attack successfully provides the CONDs (i.e., whether an attack succeeds) is evaluated during the simulation along with detailed analysis of its behavior and effects on the target WSN.

The simulation also provides a set of CONDs at the start of the simulation. These CONDs allow attacks to be defined in relation to specific hardware, software, or environmental conditions. This is particularly important for real-application simulator like SenSec when considering the effects of specific scenarios (e.g., if the target network starts with 2 compromised nodes, which is needed for an insider wormhole attack).

The design of our attack definitions provides several key features. First, the requirements and provisions capture the realistic equivalence of components, that is, attackers can use multiple methods to achieve the same goal. For example, different DoS attacks may have the same effects on a target. All possible variations of attacks can be modeled.

Second, components can represent abstract situations. For example, an ASD that returns the security keys by reading it from the sensor application directly can be used to evaluate if a security protocol is robust against stolen security keys. Thus, we can model hypothetical attacks where some requirement, such as knowing the security key, is satisfied without regard to how it is done.

Third, components can be created independently without prior knowledge of their use. The separation of components

means that we can create theoretical components without defining how its requirements will be met or how its provisions will be used. Rather than focusing on specific scenarios, the framework focuses on relationships between components.

Lastly, the components can be composed automatically to form sophisticated and previously unknown attacks. Due to the large number of variants in most WSNs, we cannot explicitly know all possible scenarios. The use of provisions and requirements allows us to compose components in ways not previously considered. By matching provided CONDs with required CONDs, the system can automatically generate new attacks to simulate, which is discussed in Section II-B.

### B. Automatic Generation of Attack Cases

SenSec can automatically generate attack cases from existing components. As we will show in a later case study (Section IV-B), this feature allows the identification of vulnerabilities through previously unconsidered attack cases.

The generation is goal-based. Goals are like requirements and can be anything that corresponds to a COND (e.g., a successfully compromised node). The generation of attacks proceeds in the same manner as that of attack trees [13], [7]. Each node is a set of requirements and each link is a component. A link between two nodes in the tree represents a component that satisfies a set of requirements (in the parent node) through its provisions and generates its own requirements (in the child node). The root node is the goal. If a component's requirements cannot be satisfied, then that component is removed from the tree and another component that fulfills the parent requirement is considered. The generation terminates when all requirements are satisfied.

Beyond the linking of components, the composition is affected by mutation, repetition, and timing. Mutation is the altering of the arguments to attack events. For example, changing which node is compromised or which node is attacked. Mutations are captured through variables in the component. A list or range of possible values for each variable must be provided by every component.

Repetition denotes both the reuse of a single component and that of a chain of components (loops) in the composition tree. The number of times that a component is used will affect the resulting attack. A global limit specifies the number of repetitions and loops allowed for any given generation.

Varying the timing of components will also affect the composition (e.g., launching attacks earlier vs. later). Rather than allow every possible component to be delayed, which may not be logical for some components, we decided to capture timing differences by having components that need it add variables to specify the simulation time that it executes.

Limiting these factors will reduce the number of generated attacks and speed up simulations. While the reverse would increase the generated number and allow more thorough testing of security systems against possible attacks. Details on bounding the state space during attack generation are discussed in Section III.

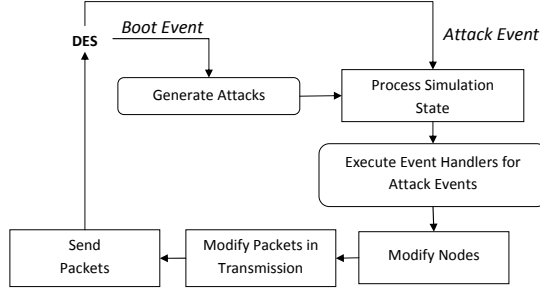


Fig. 3. Control Flow and interactions of the Attack Controller: The italicized words correspond to events that shift control from the DES to the AC. The rectangles represent interactions with the DES that do not shift control away from the AC. The rectangles with round corners represent actions that don't interact with the DES.

Although not a formal solution to validating the security of a system, the generation of attacks can facilitate the identification of vulnerabilities in both the design *and implementation* of security systems without the need for lengthy analysis.

### C. Attack Controller

The Attack Controller (AC) is a software layer that separates the DES from the attack definitions. Our goal in isolating them as much as possible is to extend the portability of the Attack Repository so that it can be used by any DES that implements the SenSec framework.

Figure 3 shows the control flow and interactions of the Attack Controller (AC). The DES is responsible for advancing the simulation time. Recall that the AC includes the Attack Generator (AG), to generate attacks, the Simulation State Processor (SSP) to process the simulation state, and the Attack Executor (AE) to execute the attack case.

1) *Execution*: DESs, in general, view the event-based simulation in terms of entities, each representing a node in the simulated network. Every entity has event handlers to process events for it, which may generate more events in turn. In a RAS, entities corresponding to sensor nodes are actually separate processes running their sensor applications. However, from the DES's perspective, the abstraction still holds and they are just a collection of event handlers and generators.

To support any DES, the AC is abstracted as a node without a radio in the DES. From the DES's perspective, the AC is just an entity that processes certain events using event handlers, possibly generating more events in the process. When an event whose event handler is in the AC, which we will refer to as an *attack event*, is processed, the DES will transfer control to the AC. After the event handler in the AC is done, it will return control to the DES.

At the start of the simulation, when nodes are initialized by the DES, a *boot* event is triggered for every node in the simulation. Event handlers initialize the corresponding node to the appropriate starting conditions. For this event, the AC will check if the user wanted automatic generation of attacks. If so, the AG generates the attacks, stores them, and sets the simulation to use the first attack. Then the SSP checks if the

simulation has attacks defined. If so, it queues an appropriate list of attack events. Otherwise, the AC will never execute in that simulation again.

After the AC is initialized, the SSP and AE are responsible for executing the attack. Every attack event will cause the DES to transfer control to the AC. At that point, the SSP will get the state of the simulation and call the appropriate handlers in the AE. These event handlers may, in turn, use the DESI to queue more events. Once the attack event has been processed by the AE, control returns to the DES.

The base set of events provided by most DESs (e.g., *Boot*, *Received Packet*, etc.) is not adequate to correctly execute an attack and custom events need to be added to specify the handling of the attack. For example, a two-part attack would require a custom event like *Launch part 2* to be queued for when it needs to launch the second phase. Normally, it would be necessary to define such custom events in the DES along with their corresponding event handlers. However, different attack definitions may require different custom events and it's impractical to require all custom events for all attack definitions to be added to the DES. Thus, the SSP, and not the DES, supports any custom events in an attack definition. It queues and dequeues them and calls the corresponding event handlers. This design allows the AC to retain control over the execution of attacks and hide all custom events from the DES. Since the AC is abstracted as a node to the DES, called the AC-node, all attack events can be linked to the AC-node. When an event for the AC-node arises, the DES calls the same handler in the SSP, which will transfer the event to the appropriate event handler in the AE.

This approach separates the AC from the DES. The DES does not require knowledge about the AC or attack events. Moreover, this approach supports the execution of attack cases that have no attacking nodes, because the AC will still execute. For example, when testing a system that detects node compromise through communication silence, the AC can randomly isolate nodes for short periods to simulate people physically compromising the nodes.

If multiple attacks are automatically generated by the AG, the SSP will store the initial state of the simulation and queue an event at the end of the simulation. When the event triggers at the end of the simulation, the SSP will reset the simulation state to the stored initial state and restart the simulation with the next attack case. The process can be stopped at the first attack to meet the goal or continue until all generated attacks are tested. However, from our experience, many successful attacks exploit the same vulnerability, so stopping at the first successful attack is more efficient.

2) *DES Interface*: The DES Interface (DESI) is a set of "helper" functions that provide and modify the data and events needed by the Attack Controller to support the execution of attack cases. The implementation of the DESI is dependent on the underlying DES, however, its interactions between the DES and AC are limited to: (1) queuing events from the AC in the DES, (2) calling the AC when its event is triggered in the DES, (3) providing needed simulation states to the AC, and (4)

changing simulation states in the DES according to the AC's instructions.

Specifically, the interactions (as shown in Figure 3) are:

- *Boot*: This interaction corresponds to the *Boot* event previously discussed. The event will cause the execution of the AG and SSP. The AC will then queue up events as necessary to correctly simulate the desired attack.
- *Process Simulation State*: The access and modification the simulation state is handled by the SSP. The SSP gets the state for every event. If multiple attack cases are simulated, the entirety of the state is saved to support restarting of the simulation for every simulated attack.
- *Send/Receive Packets*: Receiving a packet is a triggered event from the DES. Sending a packet queues an event in the DES. The DES is responsible for simulating the transmission of the packet using its physical layer models.
- *Modify Packets*: The DESI will provide access to all the packets being transmitted in the simulation. The AC can then insert, modify, or remove them according to the attack case.
- *Modify Nodes*: The positions and status of all the nodes in the simulation are accessible by the AC through the DESI. This includes getting and setting their location and on/off states.
- *Read/Write Application Variable*: There is one more interaction in the DESI, but it's between the AC and the sensor application instead of the simulator. Recall that attack cases are not limited to external attacking nodes, so data of applications (e.g., the routing table) can be accessed directly. To support this functionality, the DESI interacts with the executing application to access the variable directly.

It should be noted that, although these interactions may appear difficult to implement, many simulators that run real applications [4], [18], [17], [19] compile the application and DES code into the same executable. Since SenSec is also compiled together with them, the application variables and simulation state can be accessed directly.

### III. IMPLEMENTATION

Since any RAS can be extended with the SenSec framework, we implemented the design in TiQ [19] due to our familiarity with the simulator. There were several key issues that may be applicable to implementations of SenSec for other simulators.

First is the implementation of the COND. We represented the COND as an identifier and variable mapping. The identifier represents the abstract concept of the COND (e.g., a compromised node), which allows simple matching of CONDs across different components. The variable stores the value that is represents the state of the COND (e.g., how many nodes are compromised). This design allows computationally inexpensive preliminary checks that requirements can be satisfied (i.e., identifiers exist) to be performed.

The second concern was optimization of the attack generation. Although our design was mainly concerned with providing unconsidered attack cases, our implementation can shorten the

time needed to generate attack cases. We used the concept of dynamic programming to cache subtrees in the composition. When generating future attacks, the possible subtrees can be substituted in without having to recalculate them. Furthermore, we maintain an index of the components grouped by the CONDs that they provide to quickly find possible components that satisfy a requirement.

Implementing the AC's support for multiple attackers also proved challenging. The AC must execute the same functions but change the state for different nodes. This was accomplished through the use of context switches. The AC executes in its own thread. For each attacking nodes, the AC tracks its variables and program counter. When another attacking node needs to execute (e.g., handle a received packet), the AC will swap out the current state for the state of the new node.

Lastly, the resetting of the simulation to test multiple attack cases generated by the AG was slightly modified. We found it easier to restart the simulator than to replace its state. The generated attack cases are written out to file, which is already done for the caching, along with a record of which attack cases still need to be simulated. At the end of one simulation, the AC will remove that simulated attack from the record, do a system call to start another instance of the simulator, and then exit the current instance of the simulator. The new instance will have the same starting configuration as previous instances, and hence the same simulation state, but will simulate the next attack in the list. Since we did not explicitly copy the entire simulation state, we were able to omit the *Get/Set Simulation State* interaction from our implementation.

3) *Attack Implementation*: We implemented a set of common attacks, listed below. We go over two of the attacks in detail to show our implementation of the definition framework. Pseudo-code is used to avoid trivial technical details of programming languages.

**Selective Forwarding**: This is an attack in which a malicious node drops messages that should be forwarded. Depending on the frequency of drops, it can be difficult to differentiate this attack from occasional message loss [15].

The pseudo-code for the attack's GD is:

```
onReceive(packet) :  
  if not dropCondition(packet) :  
    send(packet)
```

As previously discussed, the attack cases are event-based. The receive and send functions are the basic interfaces discussed above. The drop condition function is the ASD. The default implementation for this attack defines `dropCondition` as `return rand(100) < THRESHOLD`, which picks a number less than 100 and checks if it is less than a threshold, defined in the configuration. The definition of this ASD is an example of how they can be application-independent and used to evaluate any application. However, if it's desirable to extend this attack to a specific application, it's possible to provide another definition of `dropCondition` with application-specific code (e.g., using the *Modify Application Variables* interface) and create a new configuration that links the new

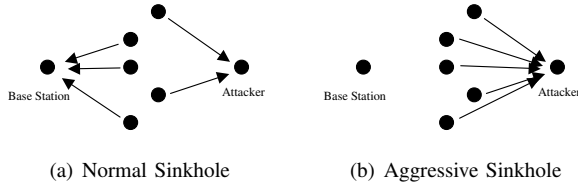


Fig. 4. Difference between a normal and aggressive sinkhole dropCondition with the same GD.

**Sinkhole:** A sinkhole attack refers to a malicious node which uses false routing information to capture surrounding nodes, causing them to route through the malicious nodes, which will then drop all received packets. WSNs, in particular, are especially vulnerable to this attack due to the presence of few base stations [10].

The pseudo-code for the attack's GD is:

```
onReceive(packet):
  if not dropCondition(packet):
    send(packet)

onTimeAdvanced():
  if sendFalseReqCond():
    sendFalseRequest()
```

The first part of this definition is identical to that of the selective forwarding attack. In fact, the sinkhole attack definition reuses the GD of the selective forwarding attack, with a different definition for `dropCondition`, linked through the configuration. This is an example of how GDs from one attack definition can be used in defining more complex attacks.

`onTimeAdvanced` corresponds to an event that is executed at each time step in the simulation. `sendFalseReqCond` and `sendFalseRequest` are ASDs. However, there is no application-independent definition for `sendFalseRequest`, because the routing packets depend on the routing protocol used in the application. However, the definition of `sendFalseRequest` can create different attacks. For example, for a tree-routing protocol, a basic definition would be:

```
sendFalseRequest():
  packet = newRoutingPacket()
  packet->metric = 0
  send(packet)
```

In this case, the malicious node pretends to be a base station. However, the sinkhole can be made more aggressive if `sendFalseRequest` is:

```
sendFalseRequest():
  packet = newRoutingPacket()
  packet->metric = 0
  send(packet)

foreach known neighbor n:
  foreach n's known neighbor m:
    packet = newRoutingPacket()
    packet->source = m's address
    packet->metric = 100
    send(packet)
```

The difference between this sinkhole implementation and the

previous one is illustrated in Figure 4. In this case, the malicious node not only pretends to be a base station, but also makes its neighbors' neighbors look worse in comparison to itself. This more aggressive sinkhole may be able to capture neighbors against a competing base station.

`sendFalseRequest` can also be defined in a manner that bypasses the evolution of the attack. Suppose that we are not interested in the creation of the sinkhole, but rather what the effects of a successful sinkhole are on the WSN. In this case, `sendFalseRequest` can automatically change the routing tables of its neighbors to evaluate what happens when a successful sinkhole is in the WSN. The three sinkhole attack cases, which only differ in their ASD definitions, demonstrates that, although a defined attack case (consisting of GDs, ASDs, and configuration) may be application-specific, their parts can be extended and reused in the definition of other attack cases.

The previous examples illustrate the framework for defining attack cases and how existing definitions can be extended for new attack cases. All attacks in our implementation are defined in the same manner. For brevity, we omit their pseudo-code:

- **Replay Attack:** In this attack, a malicious node will resend legitimate packets multiple times in order to flood the network with extra traffic or disrupt communication protocols that are affected by redundant messages. Our implementation consists of a `send` inside the receive packet event handler.
- **Sybil:** A sybil attack is when an attacking node claims multiple identities, impersonating either existing or fake nodes, in order to prevent legitimate nodes from communicating with each other. [9] discusses the attack in more detail and shows that it can be used against most aspects of sensor networks including routing, data aggregation, and resource allocation. The core implementation of this attack consists of an event handler that uses a given list of identities and modifies the source of all outgoing packets.
- **Wormhole:** This is a denial-of-service attack in which the attacker records packets in one part of the network and replays them in another part. Several, mostly routing, protocols drop redundant messages so legitimate packets that arrive after the replayed packet are ignored, which prevents the sender from getting a reply. [14] outlines several variations of wormhole attacks on sensor networks. Our basic implementation takes received packets by one malicious node, *node A*, and has another, *node B*, send them. A more detailed implementation has *node A* replay the packet using a more powerful radio and *node B*, upon reception of the packet, replay the packet using the original radio.
- **Pulse Delay:** In a pulse delay attack, malicious nodes, positioned between a sender and receiver, will record a transmission, jam the transmission so that it cannot successfully reach the receiver, and then replay the transmission at an arbitrary time in the future. It has been shown in [20] that this attack is feasible through the use of broadband jamming, cannot be prevented by conventional cryptographic primitives, and requires significant resources

to detect. The incorrect transmission delay caused by this attack is used to defeat synchronization protocols [1]. Our implementation drops any packets in transmission that are overheard by malicious nodes. The node will replay the packet later in the simulation.

#### IV. CASE STUDIES

Simulation fidelity and flexibility are necessary to evaluate real applications for WSN security. SenSec's attack system facilitates the identification of vulnerabilities and development of countermeasures. These case studies highlight the key benefits of using SenSec through case studies.

The first case study uses SenSec to evaluate routing protocols to show how SenSec can provide quantitative and meaningful analysis of the impact of attacks and the effectiveness of countermeasures. The last case study uses SenSec to identify new attacks against a secure synchronization protocol that undermines the network security beyond initially predicted bounds, which demonstrates how SenSec can easily detect and prevent security flaws in security protocols.

##### A. Evaluating the Security of Routing Protocols

SenSec provides quantitative analysis of the impact of complex attack cases on the security of network designs and meaningful feedback on the performance of countermeasures. We demonstrate this by analyzing two TinyOS routing protocols, whose implementations are provided in the TinyOS repository: Multihop and MintRoute.

Multihop is a shortest-path-first algorithm, which prioritizes routes that are the least number of hops to the base station. It also uses two way link estimation, which is an estimation based on both the quality of the communication coming from the node (the receive quality) and the communication going towards it (the send quality). The receive quality is calculated using an exponentially weighted moving average using a coefficient of 0.25. It uses the fraction of correctly received messages expressed in a range from 0 to 255. The receive quality values are sent to the neighbors who use them as their own send quality. These quality estimates are only used as a tiebreaker for routes with the same length. This means that shorter routes will always take precedence over longer routes.

MintRoute is an adaptation of Multihop in which the link quality is used instead of the hop count. This can lead to longer routes than those created by Multihop. The parent changing mechanism is triggered each time the link quality of one or more nodes becomes 75% better than the link quality of the current parent, or the link quality of the current parent drops below 25 (in absolute value). In such case, the node with the highest quality becomes the new parent. However, if two of such candidate nodes happen to have the same link quality, the new parent will be the first one found in the Neighbor Table.

In the subsequent experiments in this section, the network is a 7 node x 7 node grid. Each node can only hear its 2-8 neighbors. The base station is the top left node and the attacker is a randomly selected node in the grid. The topology was chosen in order to more easily control the number of neighbors each node

has and the hop count to the base station. The placement of the base stations helps identify how far attackers that impersonate the base station have to be in order to successfully capture nodes. All scenarios presented below were run 20 times for each protocol and the results were averaged. The attacks that we tested were selective Forwarding, sybil, and passive and aggressive sinkhole, which are discussed in Section III-3.

Consider a scenario in which the objective is to select one of the protocols for use in a sensor network to detect fires. The secrecy of the information is unimportant but we need to ensure that the network is not disabled due to one of the attacks discussed above. For our first experiment, we measured the maximum expected effectiveness of attacks against the routing protocols without any extra security measures (Figure 5). We can see that the selective forwarding attack can capture about 15% of the network for the Multihop case and 10% for the MintRoute case. However, as the attacking node drops more than about 27% of the packets, the decrease in the link quality estimates causes neighbors to switch away and route through other nodes with higher link quality. Although the sybil attack works well against both routing protocols, it requires more packets to be sent posing as other nodes than the sinkhole attacks. Each attacker has to pose as 8 nodes in order to fill its neighbors' routing tables. MintRoute, which uses dynamic metrics and does not change routes easily, is inherently more resistant to the passive sinkhole attack, but more vulnerable to the neighbor spoofing in Sinkhole (Aggressive). In contrast, Multihop, which uses static metrics, is inherently more vulnerable to Sinkhole (Passive) but resistant to neighbor spoofing.

It should be noted that the quantitative results on the relative effectiveness of the attacks are dependent on the topology of the network; in an extreme case, if a node has no other neighbors besides the attacker, then it will be forced to route through the attacker. The main focus of this experiment is to demonstrate the benefits of using SenSec to obtain a quantitative analysis and prediction of the general effectiveness and limitations of attacks for given WSN deployments.

For our second experiment, we demonstrate SenSec's role in testing and refining countermeasures. Continuing with the previous example, it is instructive to explore variations to the network design to minimize the impact of these attacks, prior to investing in more expensive security systems. We evaluate two modifications: increasing the network density and adding more base stations. Both of these countermeasures create more routes to bypass the attacker.

Figure 6 shows the average effects of increasing the network density. We see that, for both protocols, increasing alternative routes prevents the selective forwarding attack and makes it more difficult for the sybil attack to remove legitimate nodes from the routing tables. Closer inspection of the sinkhole attacks show that their decreased effectiveness is mainly due to interference from the extra network traffic preventing them from communicating with other nodes.

Figure 7 illustrates the average effect of adding more base stations on the attacks. Some nodes at the four corners became base stations. The corner nodes were chosen to avoid unrealistic

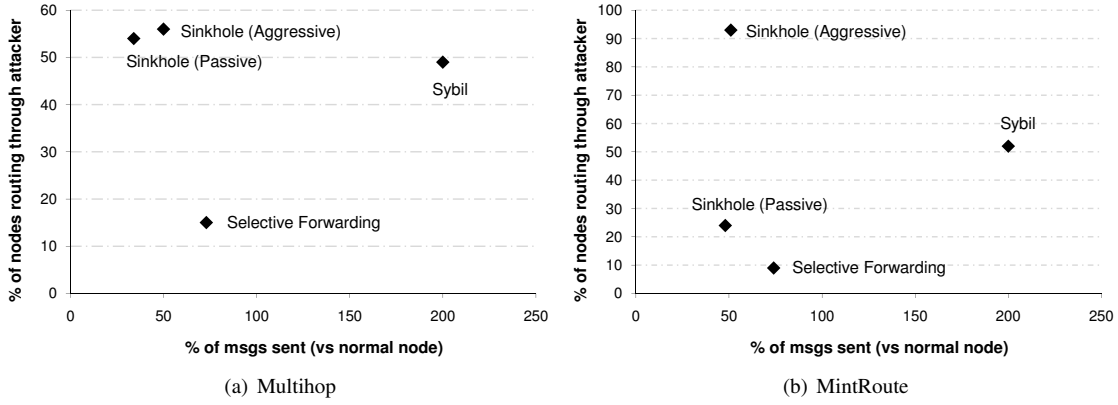


Fig. 5. Effectiveness of various attacks on routing protocols

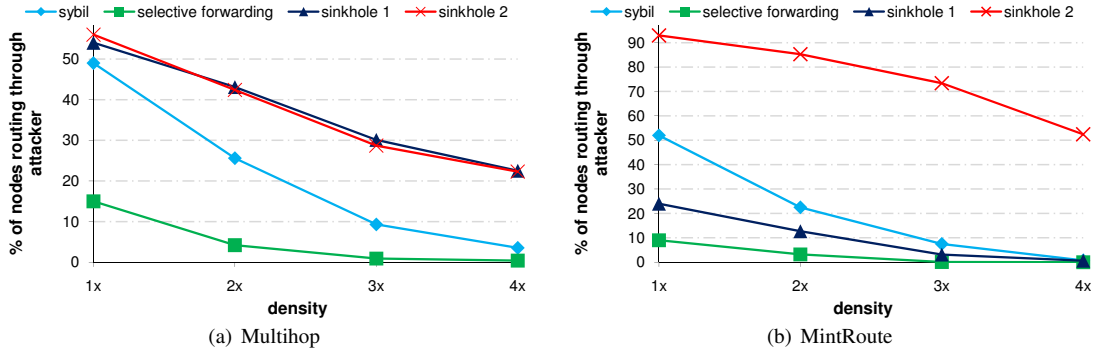


Fig. 6. The effects of increasing the network density on the attacks

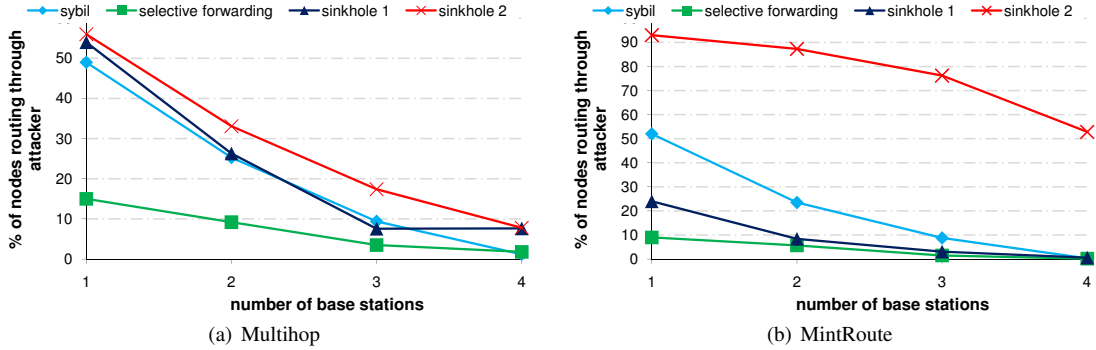


Fig. 7. The effects of adding more base stations on the attacks

scenarios such as neighboring base stations. Although both countermeasures create alternative routes to bypass the attacker, we see that adding more base stations has a larger impact. Closer inspection shows that there is a spatial factor involved. That is, each extra base station is added to one of the four corners so the routes are more distributed than those from increasing the network density.

For Multihop, we see that all of the attacks become ineffective due to the increased competition from routes to legitimate base stations. The effectiveness of the sinkhole attacks decreases to a minimum point where the attacker can only capture its nearby neighbors. In contrast, the use of a dynamic metric in MintRoute allows the Sinkhole (Aggressive) attack to capture a large portion despite more base stations.

From these results, we conclude that using Multihop with 3 or 4 base stations at the corners is a simple method of minimizing the impact of these attacks against the network without deploying a costly security system. Network designers can analyze various metrics and pick the optimal countermeasure based on their individual criteria.

This case study shows how SenSec can quantitatively compare the impact of various attacks on a network, identify the inherent strength and weakness of protocols, applications, and networks, and aid the development of countermeasures.

### B. Secure Time Synchronization

In this section, we demonstrate SenSec's ability to facilitate the detection of security vulnerabilities. When security protocols are designed, simulators not specifically designed for

WSN security are used to analyze them. The use of these tools requires that the behavior of specific attack cases be individually modeled and added to the simulation. Unfortunately, this approach limits the scope of the evaluation to the test cases considered by the authors. SenSec, however, uses test suites to provide systematic evaluation against a large set of test cases, which facilitates the detection of vulnerabilities that would not have been apparent otherwise.

For this study, we consider the issue of secure clock synchronization, which is essential for any sensor networks that aggregate sensor readings in a hostile environment, such as a battlefield. Without a global agreement on time, the data from different sensors cannot be accurately matched up for later analysis. However, the synchronization protocol can be attacked to create larger clock errors than those that would naturally occur. Secure clock synchronization is a response to that threat by securing the synchronization from attackers. We tested the security of one such protocol - Secure Opportunistic Multihop Synchronization (SOM) [1].

SOM is a synchronization protocol that includes various security measures. In SOM, the sender and receiver, which can be multiple hops away, share a secret key. Synchronization and acknowledgement packets use the key to generate a Message Authentication Code (MAC) to guarantee message integrity and authenticity. SOM was designed to detect pulse delay attacks by calculating the average delay:

$$d = \frac{(T_2 - T_1) + (T_4 - T_3)}{2} \quad (1)$$

where  $T_1$  and  $T_2$  are the send and receive times for the synchronization packet and  $T_3$  and  $T_4$  are the send and receive times for the acknowledgement packet. If the calculated delay is above an expected delay threshold ( $d^*$ ), then a pulse delay attack has likely occurred. The expected delay threshold bounds the maximum time that an attack can delay the packet without detection. From their experimental results, the authors of SOM provide some experimental results for comparison. The results for that experiment show that maximum attacker impact is  $6\sigma$  where  $\sigma$  is the standard deviation of the end-to-end delay. Thus, SOM is unsuitable for WSNs that cannot tolerate clock errors greater than this value.

For our initial investigation, the motes start with no initial clock skew and a clock drift of 0 to 40  $\mu$ s, the reported drift range for Mica2 motes [6]. The radio range was about 100m. A base station was placed at the center of the grid to receive and aggregate sensor readings. A grid network is set up where each node is at most 3 hops from the base station. The base station acts as the global reference clock (each node shares a key with it and synchronizes to its clock). To test for vulnerabilities, we set SenSec to generate attacks and use them against SOM.

SenSec was able to identify an unexpected attack that circumvented SOM's security. The attack is an extension of the pulse-delay attack that SOM was created to counter. It consists of a pulse delay attack against the synchronization packet combined with a wormhole attack against the acknowledgement packet. The pulse delay attack will increase the time difference when

the receiver synchronizes its clock to that of the sender and the wormhole attack will prevent the delay from being detected through the average end-to-end delay. Furthermore, the average delay will even out the difference. In this case, the maximum attacker impact is:

$$2 \times (d - d^*) + d_W \quad (2)$$

where  $d_W$  is the decrease in delay caused by the wormhole.

The counter to this composite attack is an additional check that  $(T_2 - T_1) - (T_4 - T_3)$  is within some threshold, because a large difference between the two delays would signal this composition attack. This value is already calculated in the original SOM, so adding it presents trivial overhead.

For our setup, we found the maximum attacker impact to be  $18\sigma$ , with maximum clock errors of 536  $\mu$ s. It should be noted that this number is dependent on the topology, network traffic, and other factors. Our objective is to highlight how SenSec can facilitate the detection of vulnerabilities and allow researchers to perform more comprehensive evaluations by using our framework and the library of existing attacks without the need to develop a new set of attack cases through analysis.

## V. SCALABILITY

We evaluated the scalability of SenSec in terms of its overhead as measured by the execution time (i.e., the time required to execute a simulation). These results, which show the overhead of the SenSec framework, are summarized in Table I.

In order to determine the overhead presented by the addition of SenSec to a simulator, we selected 20 random TinyOS applications and ran them in numerous simulations in TiQ *without* the SenSec framework. Each simulation had 100 to 200 nodes randomly deployed over a 1000m $\times$ 1000m terrain and lasted for 1 to 10 simulated hours. We then replicated the simulations in TiQ *with* the SenSec framework and used the difference to determine the overhead. All subsequent scalability experiments replicated this setup with minor changes.

We found the average overhead to be 0.07 seconds regardless of the network size or the simulated duration. Thus, the addition of the SenSec framework to a simulator presents negligible overhead to simulations that don't use it.

We reran the same experiment but, for the SenSec version, added in an attack case with a custom event, *TEST*, and queued 1,000 to 1,000,000 of them at the start of the simulation. The attack definition had an empty *TEST* event handler. This experiment tells us the base overhead for every attack event. Accounting for the previous overhead, the average overhead of having SenSec execute in the simulation is 0.0000012 second per event. The network size did not affect this overhead.

| Item                     | Execution Time Inc. |
|--------------------------|---------------------|
| Adding SenSec            | 0.07 s              |
| Handling any event       | 0.0000012 s / event |
| Adding a component       | 0                   |
| Adding an attacking node | 0                   |

TABLE I  
OVERHEAD OF SENSEC

To find the base overhead of adding more components to the attack definition, we reran the experiments but had the attack case be composed of 100 to 10,000 empty GDs. We found no statistically significant overhead in any of the simulations. The same was true when the experiment was repeated with 100 to 10,000 attacking nodes that did nothing. Thus there is no inherent overhead with adding more attacking nodes or using more components.

These results demonstrate that SenSec can provide accurate emulation of real applications and attacks with low overhead. Our design and implementation of the framework presents overhead that scales, at worst, linearly with the number of events generated by an attack. The majority of overhead in using SenSec will be caused by the simulation of sophisticated attacks. SenSec's accuracy and scalability makes it a beneficial framework for evaluating security in large WSNs.

## VI. RELATED WORK

To the best of our knowledge, SenSec is the first extensible evaluation framework for WSNs reported in the open literature. In the context of middleware, Ranson et al. proposed a security evaluation framework that shares similar goals as SenSec's attack system to detect, evaluate and prevent security flaws [12]. However, it focuses on the security needs of the application. Zhang et al. proposed an emulated environment that is limited to testing routing attacks in MANETs [22]. LARIAT is a well-known IDS testbed for wired networks used in DARPA Intrusion Detection evaluation [5]. Although it does allow attack cases to be added to the testbed, it does not provide a framework for extending existing attacks to form new, more complex attacks.

In the more general area of sensor network evaluation, there has been significantly more work. Existing work mainly consists of sensor emulators that are enhanced with network models. TOSSIM [4] comes with the TinyOS distribution and provides a rudimentary simulation of the physical environment. EmStar [3] takes a similar approach to TOSSIM but targets higher end sensor nodes. These emulators use highly abstract models, especially for the wireless channel, can lead to inaccuracies in the predicted performance and behavior of simulated sensor networks [18]. Another focus in the area is to extend network simulators to support sensor network evaluation, such as the case with SenQ [18] and TiQ [19]. The benefit of this approach is that it leverages the physical layer models in the underlying network simulators without having to re-implement them for a stand-alone simulator. Given the significant amount of effort that has already been invested in these simulators, SenSec is designed to be composable with any of these, or other, simulators. The SenSec framework also allows the integration of emerging network simulators like ns-3 into existing sensor emulators, and we hope that this work suggests an approach in that direction.

## VII. CONCLUSION

In this paper, we have presented the design and implementation of the SenSec framework, which provides high fidelity

evaluation of security issues in WSNs involving complex network scenarios, multiple attackers and heterogeneous networks. We have demonstrated the key benefits of SenSec: (1) it can extend any real application simulator to support evaluation of WSN security, (2) it provides a modular framework for defining attack cases with the flexibility to extend and compose existing attack cases to define new or more sophisticated attacks, (3) it can execute unmodified real applications to provide evaluation of the security of real-world implementations, (4) it can automatically generate and evaluate attack cases not previously considered, and (5) it allows attacks and countermeasures to be easily analyzed, compared, and optimized prior to deployment of the WSN.

Furthermore, our implementation has shown that the SenSec framework introduces minimal overhead in order to provide these benefits. We believe that SenSec will be a useful tool for the WSN research community.

## REFERENCES

- [1] Ganeriwal, S., et al.: Secure time synchronization in sensor networks. *ACM Trans. Inf. Syst. Secur.* 11(4), 1–35 (2008)
- [2] Girod, L., et al.: A system for simulation, emulation, and deployment of heterogeneous sensor networks. In: *SenSys '04*. pp. 201–213 (2004)
- [3] Girod, L., et al.: Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks. *ACM Trans. Sen. Netw.* 3(3), 13 (2007)
- [4] Levis, P., et al.: Tossim: accurate and scalable simulation of entire tinys applications. In: *SenSys '03*. pp. 126–137 (2003)
- [5] Lippmann, R., Haines, J.W., Fried, D.J., Korba, J., Das, K.: Analysis and results of the 1999 darpa off-line intrusion detection evaluation. In: *RAID '00*. pp. 162–182. Springer-Verlag, London, UK (2000)
- [6] Maróti, M., et al.: The flooding time synchronization protocol. In: *SenSys '04*. pp. 39–49 (2004)
- [7] Mauw, S., Oostdijk, M.: Foundations of attack trees. In: *ICISC 2005*. pp. 186–198. Springer (2005)
- [8] McCanne, S., Floyd, S.: ns-2. <http://www.isi.edu/nsnam/ns> (2009)
- [9] Newsome, J., Shi, E., Song, D., Perrig, A.: The sybil attack in sensor networks: analysis & defenses. In: *IPSN '04*. pp. 259–268 (2004)
- [10] Ngai, E., et al.: On the intruder detection for sinkhole attack in wireless sensor networks. In: *ICC '06*. vol. 8, pp. 3383–3389 (June 2006)
- [11] Qualnet: <http://www.scalable-networks.com> (2009)
- [12] Ransom, S., Pfisterer, D., Fischer, S.: Comprehensible security synthesis for wireless sensor networks. In: *MidSens '08*. pp. 19–24 (2008)
- [13] Schneier, B.: Attack trees. *Dr. Dobbs' Journal of Software Tools* 24, 21–29 (dec 1999)
- [14] Sharif, W., Leckie, C.: New variants of wormhole attacks for sensor networks. In: *ATNAC '06*. pp. 288–292 (2006)
- [15] da Silva, A.P.R., et al.: Decentralized intrusion detection in wireless sensor networks. In: *Q2SWinet '05*. pp. 16–23 (2005)
- [16] TinyOS: <http://www.tinyos.net> (2000)
- [17] Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: a scalable sensor network simulation with precise timing. In: *IPSN '05*. p. 67. IEEE Press, Piscataway, NJ, USA (2005)
- [18] Varshney, M., Xu, D., Srivastava, M., Bagrodia, R.: Senq: a scalable simulation and emulation environment for sensor networks. In: *IPSN '07*. pp. 196–205 (2007)
- [19] Wang, Y.T., Bagrodia, R.: Scalable emulation of tinys applications in heterogeneous network scenarios. *MASS '09*. pp. 140–149 (Oct 2009)
- [20] Xu, W., et al.: The feasibility of launching and detecting jamming attacks in wireless networks. In: *MobiHoc '05*. pp. 46–57 (2005)
- [21] Zeng, X., et al.: Glomosim: a library for parallel simulation of large-scale wireless networks. *SIGSIM Simul. Dig.* 28(1), 154–161 (1998)
- [22] Zhang, Y., an Huang, Y., Lee, W.: An extensible environment for evaluating secure manet. *SecureComm '05*. 0, 339–352 (2005)