DRHIP: A HOST IDENTITY PROTOCOL FOR SECURING WIRELESS SENSOR NETWORK

by

Panneer Selvam Santhalingam
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Information Security and Assurance

Committee:

_____	Dr. Robert Simon, Thesis Director
_____	Dr. Xinyuan Wang, Committee Member
_____	Dr. Kris Gaj, Committee Member
_____	Dr. Sanjeev Setia, Chairman, Department of Computer Science
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering.

Date: _____ Spring Semester 2016
George Mason University
Fairfax, VA

DrHIP: a Host Identity Protocol for Securing Wireless Sensor Network

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Panneer Selvam Santhalingam
Bachelor of Science
Anna University, 2010

Director: Dr. Robert Simon, Professor
Department of Computer Science

Spring Semester 2016
George Mason University
Fairfax, VA

Acknowledgments

I would like to thank my advisor Dr. Robert Simon for his advice and support. I would like to thank my family for their constant support and encouragement.

Table of Contents

List of Tables

List of Figures

Abstract

DRHIP: A HOST IDENTITY PROTOCOL FOR SECURING WIRELESS SENSOR NETWORK

Panneer Selvam Santhalingam

George Mason University, 2016

Thesis Director: Dr. Robert Simon

As the use of wireless sensor networks (WSNs) are rapidly becoming a fundamental part of the global infrastructure, effective and reliable authentication schemes are now essential. For WSNs this problem is difficult, since a typical system is comprised of devices that are resource constrained. The purpose of this thesis is to study the design of a lightweight experimental authentication scheme based, on the Host Identity Protocol (HIP). In particular, we propose the *DrHIP* (DoS Resilient Host Identity Protocol) protocol, which is designed to support WSN group key management while offering resistance to Denial of Service attacks. The DrHIP protocol utilizes the puzzle mechanism of the Host Identity Protocol in mitigating the impact of an attacker launching spurious authentication requests. We have implemented DrHIP in a working testbed of Z1 motes using the Contiki system, and have experimentally shown that DrHIP can offer high levels of goodput even in the event of DoS attacks against the authentication system.

Chapter 1: Introduction

Over the last decade Wireless Sensor Networks (WSNs) have seen their use dramatically increase because of the advancements in semiconductor, networking, and material science technologies [1]. Today, WSN are used in health care monitoring, environmental monitoring, industrial monitoring, and several other purposes [2]. Information gathered by these networks is utilized in making critical decisions, hence security is a key aspect in WSN. It is also increasingly desirable to provide *e*nd-to-end security. Here end-to-end security implies the ability to protect data from all the intermediary parties and reveal the data only to the actual communicating parties (sender and recipient). Part of this processes includes authentication, in order to setup secure end-to-end protection. Further, since WSNs are constrained devices, there is a need to provide this security with little utilization of the resources.

The most common security measure that is currently being used is 802.15.4, which provides link layer security. IEEE standard 802.15.4 provides a set of protocols targeted at low power devices, personal area networks and sensor nodes. Acting at the data link layer the protocol provides access control, message integrity, message confidentiality and replay protection. The problem is, being at link layer, this protocol only provides a hop by hop security [3]. While there has been many attempts to provide end-to-end security with various protocols, there is no common standard as of now. This thesis builds upon a recent experimental protocol called Host Identity Protocol [4] in its *d*iet exchange flavor as a building block to providing end-to-end security in the WSN. Our major contribution is a modified version of the protocol to support group key management and Denial-of-Service (DoS) resistance. The protocol is called DrHIP, for Denial of service resistant Host Identity Protocol. We have implemented DrHIP in the Contiki WSN framework, and have evaluated its performance under a number of scenarios.

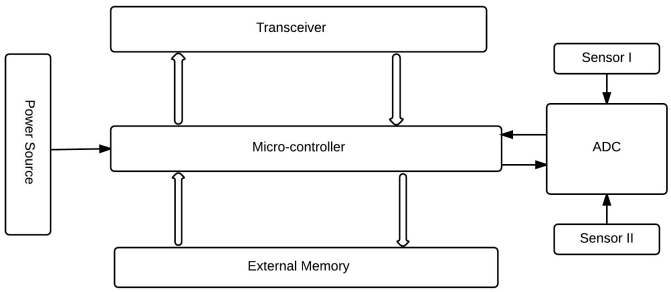


Figure 1.1: Components of a typical sensor node

Before we elaborate on the protocol, to get an understanding on the WSN and their components, we will brief on the structure of sensor nodes, interaction between them, their resource constrained nature, and the challenges faced in securing them.

1.1 Wireless Sensor Networks

Wireless sensor network (WSN) are spatially distributed autonomous nodes used to monitor physical or environmental conditions such as temperature, sound, pressure, etc., and to cooperatively pass their data to a main location [5]. From surveillance to industrial control systems WSNs are being used in several places. WSN are usually comprised of resource constrained devices which we will refer to as *s*ensor nodes, to indicate their functioning in sensing the physical environment. The sensor nodes gather information from the environment and pass it to a sink or gateway. As the channel between the nodes and the sink is wireless anyone can intercept these messages. The integrity of the messages are critical as the system decisions are based on them. Hence it is important that these messages are secured.

1.1.1 Sensor Nodes as Constrained Devices

A sensor node is constrained in its functioning in several ways. These constraints set an upper bound on what these devices can handle. According to [6] a constrained device or node can have any of the following constraints.

- Constraints on the maximum code complexity (ROM/Flash).
- Constraints on the size of state and buffers (RAM).
- Constraints on the amount of the computation feasible in a period of time (Processing Power).
- Constraints on available power.
- Constraints on the user interface and accessibility in deployment.

A sensor node with any of the above mentioned constraints cannot function like a normal wireless node. For example, if a sensor node constraints on the available power, it has to operate in a way that maximizes its performance with the available power. Sensor nodes usually have more than one of the above mentioned constraints. Hence, they have to be treated differently in all aspects like neighbor discovery, data exchange, security, etc. We now provide further technical details:

- Sensor Node

  A WSN comprises multiple sensor nodes, each coordinating with each other and exchanging data. Figure-1.1 depicts the different components of a typical sensor node. The major function of a sensor node is to sense a change in the environment and forward the change to the sink. Hence, each sensor node is equipped with sensors and analog to digital converters (ADC) at the least. For a sensor node to be able to exchange data it will need transceivers. Other components include power source, external memory, and micro-controller which acts as the central controller. Power sources include batteries or solar power, depending on the environment the sensor node is deployed.

- Microcontroller

  A microcontroller serves as an embedded processor used in scheduling tasks, controlling the function of other components, and processing the collected data. There are wide range of microcontrollers that are being used in different sensor nodes. Based on the required functionality other embedded processors like Digital Signal Processor (DSP), Field Programmable gate array (FPGA), and Application Specific Integrated Circuit (ASIC) are also used.

- Radio Transceiver

  It is used for wireless communication. Sensor nodes mostly use ISM bands for their communication of the available spectrums. A transceiver can operate in one of the following states Transmit, Receive, Idle, and Sleep.

- Memory

  Sensor nodes operate with very limited memory, for both program and data. The memory includes in-chip flash memory, RAM of the microcontroller, and external memory. Based on the size of the node the memory capacity varies, for example the external memory of motes range from 1024 KB to 1 GB [5].

- Sensors

  Sensors are used to sense different environmental conditions. There are three types of sensors [7], passive, omni-directional, and active sensors. Passive sensors do not probe the environment, they just sense the data, while active sensors actively probe the environment while collecting data. Omni-directional sensors lack the notion of direction in their sensing. Nodes include different sensors like, temperature sensors, humidity sensors, light sensors, pressure sensors, accelerometers, magnetometers, or chemical sensors based on their application.

- Analog to digital converters

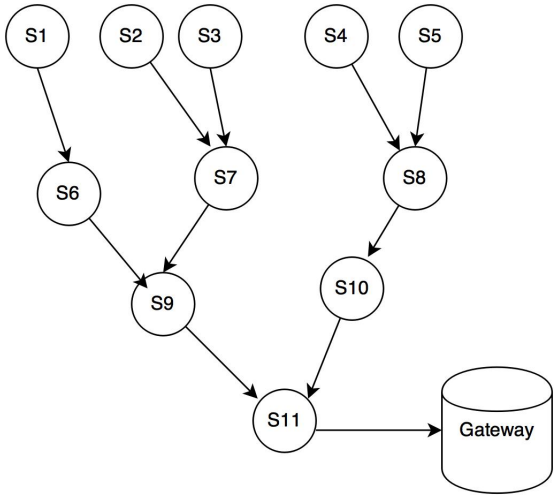


Figure 1.2: Typical wireless sensor network

The data gathered by the sensors is analog. Hence, there is a need for the data to be converted to digital before further processing like aggregation. This is where the analog to digital converters come into picture.

1.1.2 Communication between the nodes

Communication between the nodes usually happens through the transceiver, the data from the nodes is gathered by the aggregator node and sent to the gateway. For example, Figure-1.2 depicts a typical wireless sensor network with a total eleven nodes, each node gathering data which is aggregated by the node S11 and reported to the gateway. The entire communication is wireless, making it vulnerable to several attacks. Hence, securing the communication is important as most of the data is used in making critical decisions in the environment the node is deployed. Additionally, there is a need to secure the channel with minimal resources as possible, which leads to the discussion of challenges faced in securing the WSN.

1.2 Challenges faced in securing wireless sensor networks

In terms of securing the WSN, we are looking for a protocol that is light weight and provides access control, message integrity, and message confidentiality, along with protection against well-known attacks [8]. One of the key challenges is to provide security with limited resources. This is the reason for the existing security mechanisms well defined for the normal networks, are being proved to be useless when considered for WSN. While 802.15.4 provides the required security mechanisms, the provided security is respect to the link layer as mentioned earlier. Adding to this, there are challenges with respect to the environment where the nodes are deployed. At times, nodes operate from hostile environments making it hard to protect them from physical attacks. Also, the nodes are usually managed wirelessly and the nodes need to aggregate data to avoid redundancy and be more efficient. This leads to the problem of secure aggregation [9]. Finally, being wireless the connection is unreliable and prone to frequent packet losses.

A large number of protocol have been specifically designed for WSNs [10], [11], [12], [13]. They address issues such as modifying well-known security protocols in making them compatible for WSN [14], and identifying attack vectors and providing counter measures for them.

In this thesis, we have taken the approach of studying an existing protocol for its aptness in securing the WSN. Our major contributions are

- An expanded version of Host Identity Protocol called DOS resilient Host Identity Protocol (DrHIP) aimed towards sensor nodes.
- A flexible group key management system suitable for sensor nodes.
- A puzzle based mechanism capable of deep in-network resisting denial of service attacks against the authentication subsystem.
- Evaluation of DrHIP under different attack models.

From here, the thesis is structured as follows. In the next Chapter we will see the

associated background, followed by an overview of the protocol in Chapter 3 and its implementation in Chapter 4, performance evaluation in Chapter 5 and finally, conclusion and future work in Chapter 6.

Chapter 2: Background

We have emphasized the need to secure the data exchanged in a WSN with minimal resource utilization. It has been widely established that for data to be exchanged securely, the best way is to encrypt the data using cryptographic keys in conjunction with encryption algorithms. This leads to the problem of key establishment; how two sensor nodes can agree on a common key securely with little utilization of the available resource? Any key establishment technique must incorporate the following features [15]:

- Confidentiality : Protect the data from being disclosed to unauthorized sources.
- Integrity : The keys should be accessible by only the nodes in WSN, the privilege to change the keys should only be with the base station.
- Scalability : The provided security features shouldn't be constrained on the network size, should be able to scale to large networks if needed.
- Flexibility : Should be able to deploy nodes dynamically.

In addition to the listed features, the key management techniques for WSN need to have certain other features [15]. These features are necessary to overcome the constraints of the sensor nodes.

- Resistance : Should be able to withstand node replication by adversaries and guard against such attacks.
- Revocation : Should have a method to revoke the compromised nodes.
- Resilience : Capture of node in the WSN shouldn't reveal information of other nodes.

There has been a great deal of work in addressing the key establishment problem in the WSN. In this Chapter we will first discuss some related existing work on securing the WSN and from there we will cover some relevant background for our work in DrHIP.

2.1 Related Work

A very good overview on the security threats posed to WSNs, along with counter measures, is presented in Chris and David [16]. Some of the attacks discussed include Selective forwarding, Sinkhole attacks, Sybil attacks, Wormholes, HELLO flood attacks, etc. Another good discussion of DoS attacks against WSNs is presented by Anthony and John in their work [17].

A game theoretic approach on preventing DoS attacks on wireless sensor networks has been studied by Afrand and Sajal [18]. In their work, they have modeled the interaction between the nodes and Intrusion detection system in the network as repeated game and use cooperation among the nodes and a sense of reputation among the neighbors in sensing if a node is malicious. In another work Dimple and Neha [19] have used an ant based framework in sensing and preventing DDoS attacks in wireless sensor networks.

The focus of my research is trying to understand if the HIP approach of using cryptographic identifiers to replace some of the functioning of IP addresses can be applied in a WSN context. Relatively little work has been done in studying HIP for its feasibility in securing WSN. One example is Nie, Pin, et al. who have evaluated the effectiveness of HIP-DEX considering network initialization, data transmission, dead node resurrection and new node replenishment [20]. They have come up with practical attack models for HIP-DEX: Radio jamming, DDOS, Replay attack, Sybil attack, and Warmhole/ MITM. They suggest using whitelists to prevent against Sybil attack. The experiments and evaluations were conducted on SunSPOT. In another work [21] Campbell, Andrew T., et al. have implemented and evaluated HIP-DEX in performing 3GPP-based authentication. They have used HIP-DEX as an authentication protocol between user equipment and authentication

gateway. Usage of HIP in communication between medical sensor inside patients and the backend server was performed by Kuptsov, et al. [22].

Hummen, Rene, et al. in their work on [23] end-to-end security for the Internet of Things have addressed some important problems with respect to HIP association. They came up with an implementation of HIP-DEX on *Contiki operating system* along with *Relic tool kit* for cryptographic primitives. They have proposed a comprehensive session resumption mechanism, collaborative puzzle based DOS protection and retransmission mechanism refinements. For collaborative puzzle based DOS protection an upper bound is set on the number of DH operations allowed in a particular time window. If the DH operations exceed this upper bound the puzzle difficulty is increased to protect against DOS. Set the difficulty based on the level of trust with a particular node has been suggested. Some of their suggestions on retransmission has been incorporated into the current version of HIP-DEX draft.

Of the listed works, the work done by Hummen, Rene, et al. [23] is most similar to ours in that they have also studied HIP's puzzle mechanism. But, their work studied HIP's puzzle mechanism between two nodes, and evaluated different puzzle difficulties in maintaining the DH operations over a time window. In our work, we have studied the effectiveness of puzzle mechanism in mitigating the attacker's effect on the network. Also, we have equipped the GW with the capability to sense the attack thus reducing the work load on the constrained sensor nodes.

2.2 Key management in WSN

Key management in the WSN has been deeply studied in literature and there are several techniques which are currently being used. According to [15] some of the well-known key management schemes include single network-wide key, pairwise key establishment, trusted base station, public key schemes, key predistribution schemes, dynamic key management, and hierarchical key management. We will be discussing some of the key management schemes which are relevant to our work in here. To know about the other key management

schemes please refer [15] [24] [25].

2.2.1 Pairwise key establishment

In pairwise key establishment each node in the WSN establishes a pairwise key with every other node in the WSN. This key will be used for providing the necessary security. Thus, for a WSN with n nodes each node will have to store n-1 keys in their memory. Pairwise key establishment technique offers a node-to-node authentication and an increased resilience as it doesn't reveal information on the nodes that are not directly communicating with captured node [24]. The problem with pairwise key establishment techniques is the memory overhead it brings on each node in the WSN. For a WSN of 1000 nodes, each node needs to store keys for 999 nodes.

2.2.2 Public key schemes

Public key schemes are used to generate the private/public key pair, of which one is available public (public key) and the private key is retained by the owner. The data which is being encrypted with a public key can be decrypted only with the private key which is with the owner. The problem with this scheme is that, it is computationally expensive and using this for securing data transfer is not an efficient solution. Hence public/private key pairs are used in exchanging symmetric keys and these keys are used for the data encryption. This is done by using methods like Diffie-Hellman. For our implementation we will be using Elliptic Curve Deffie-Hellman (ECDH) for exchanging the symmetric keys securely.

2.2.3 Hierarchical key management

In wireless sensor network, the hierarchical key management system is comprised of different keys used for different purposes. All the packets in the wireless sensor network cannot be authenticated. For example, certain packets like the initial hello packet and the notification packets do not contain critical data and are more often exchanged in the network. Securing them would add to computational overhead for the constrained devices and thus ignored.

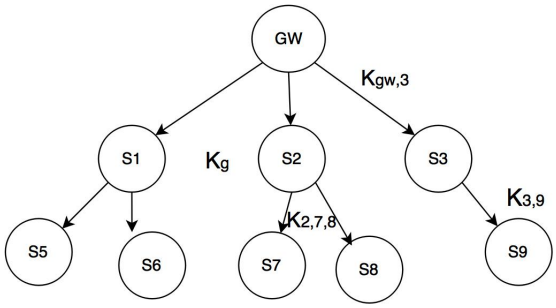


Figure 2.1: Hierarchical Key management

Hence, based on the type of the packet and the devices between which the packets are going to be exchanged different keys can be used. These keys include, unique key between the gateway and the node, pairwise key between the nodes, cluster key which is common among the node and its neighbors and Group key which is shared between the gateway and all the other nodes. To know more about how different keys are being established in a hierarchical key management system refer [15].

Figure-2.1 depicts a typical wireless sensor network, with the gateway at the root of the network, followed by other sensor nodes which are one to multiple hops away from the gateway. The different keys that can be established in such a network, using hierarchical key management scheme include the individual key($K_{gw,3}$), pairwise key($K_{3,9}$), cluster key($K_{2,7,8}$), and group key(K_g).

2.3 Cryptography primer

In this work we use Elliptic Curve Diffie-Hellman (ECDH) for exchange of the symmetric keys along with Advanced Encryption Standard in Cipher Block Chaining (AES-CBC) mode for encryption and Secure Hash Algorithm 1(SHA-1) for generation of hashes. We will review the fundamental operations of these cryptographic protocols before proceeding further.

2.3.1 Elliptic Curve Deffie-Hellman (ECDH)

The Elliptic Curve Deffie-Hellman is used for exchanging cryptographic keys securely in an insecure network. The security provided is based on discrete logarithm problem [26]. For two parties to exchange keys securely they need to agree on few initial domain parameters to start with which vary based on the way in which ECDH is used. There are two ways in which ECDH can be implemented either using Binary fields or prime fields. In our implementation we have used the ECDH based on prime fields. The domain parameters for the prime fields include

- Field: given by p
- Elliptic curve constants: a and b used in defining the equation
- Generator: given by G
- Order: given by n
- Cofactor: given by h

Once the involved parties agree on the domain parameters, each one of them generate a public/private key pair. Public key is a point on the elliptic curve and the private key is an integer. Each party exchange their generated public key to other, the private key is never revealed. After receiving the public key, each party combines it with their private key to obtain the symmetric key. If someone obtains the symmetric key and public key of one of the parties, they will have to solve the discrete logarithm problem to obtain the private key.

2.3.2 Advanced Encryption Standard Cipher Block Chaining (AES-CBC)

Advanced Encryption Standard also know Rijindael is a specification for the encryption of electronic data established by U.S National Institute of Standards and Technology (NIST) in 2001. AES is based on Rijindael cipher developed by two Belgian Cryptographers, Joan Daemen and Vincent Rijmen, who submitted a proposal to NIST during the AES selection process. For AES, NIST selected three members of the Rijndael family, each with a block

size of 128, but with different key lengths: 129, 192, and 256 bits [27]. While operating in Cipher Block Chaining [CBC] mode, every block of the plain text is XORed with the cipher text of the previous block, for the first block an initialization vector is used in place of the cipher text. As encryption can only be performed in specified block sizes, one might have to use padding to account for a different size than the chosen block size.

2.3.3 Secure Hash Algorithm 1(SHA-1)

SHA is a family of cryptographic hash functions, used for generating message digests. SHA-1 produces cryptographic hashes of 160 bits in length. SHA-1 is part of four NIST standard SHA algorithms that are structured differently. The algorithms include SHA-0, SHA-2, and SHA-3.

Cipher-based message authentication code

Cipher-based message authentication code is block cipher-based message authentication code (MAC) generating algorithm. A message authentication code is used for authenticating a message and it is obtained by passing the entire message into the MAC algorithm which computes the MAC using the provided secret key. Although, MAC might seems similar to cryptographic hashes they vary in their security requirements. And MAC is different from digital-signature in that it uses the same key for generating and verifying the MAC.

2.4 Host Identity Protocol

The Host Identity Protocol (HIP) is an experimental protocol designed to address the problem of IP addresses being used both as an identifier and a locator [4]. Accordingly, HIP provides an identifier called *Host Identifier*, which is the public key of a public/private key pair and the upper layer protocols are bound to this instead of the IP address. Along with the host identifier, HIP provides another identifier called the *Host Identity Tag* (HIT), this is the operational representation of the host identifier. It's 128 bits in length and derived from

Table 2.1: Supported HIT Suites.

HIT Suite	ID
RESERVED	0
RSA,DSA/SHA-256	1 (REQUIRED)
ECDSA/SHA-384	2 (RECOMMENDED)
ECDSA_LOW/SHA-1	3 (RECOMMENDED)

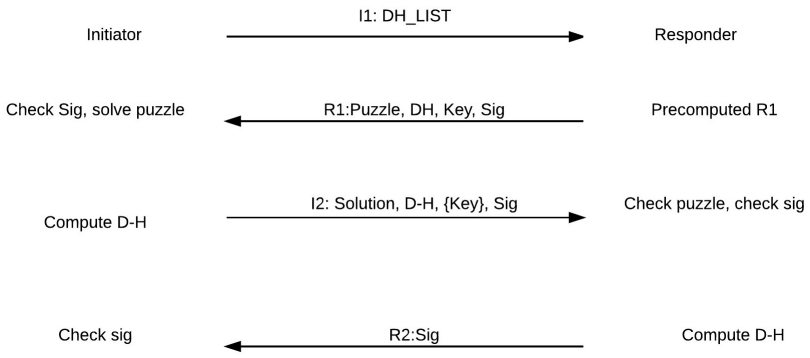


Figure 2.2: HIP Hand Shake

the host identifier by hashing it. HIT should be derived according to ORCHID generation method as stated in [28]. HIT is similar in length to an IPv6 address and it is self-certifying (given an HIT it is computationally hard to find the HI that matches it), the probability of HIT collision with two hosts is very low [29]. HIT can be generated by using different hash functions. *HIT Suites* group the set of algorithms required to generate a particular HIT. The suites are encoded into HIT Suite IDs and transmitted in the ORCHID Generation Algorithm field in the ORCHID. Table I lists the different HIT suites supported. HIP can be combined with Encapsulated Security Payload and other end-to-end security protocols. HIP comes in two flavors Base Exchange (HIP-BEX) and Diet Exchange (HIP-DEX).

2.4.1 Base Exchange(HIP-BEX)

This is a two-party cryptographic protocol used to establish communication context between hosts [29]. The handshake is SIGMA-compliant four packet exchange [30]. The two parties are named according to the role they play, the one who initiates the connection is called the initiator and the one who responds is called the responder. Once the association is established these distinctions are not needed anymore, hence they are forgotten. The four packets involved in the association establishment are named as I1, R1, I2, and R2 respectively. Figure 2.2 shows the four packet handshake between an initiator and responder. The figure doesn't include all the parameters that are sent as part of the packets. All the packets include the initiator's and responder's HIT(if not operating in opportunistic mode [29]).

- The first packet I1 from the initiator acts as the trigger packet. This contains the Deffie-Hellman group list.
- On receiving the I1 packet the responder sends one of the precomputed packets. This packet R1 includes the following parameters:
  - Cryptographic puzzle
  - Counter
  - Deffie-Hellman parameters
  - Host Identity
  - HIP Cipher
  - HIT Suite List
  - Transport format list
  - Signature
- Initiator receives R1 and uses the Deffie-Hellman parameters in generating a session key. This session key can be used in encrypting the host identity. The I2 packets includes the following parameters

  - Puzzle solution
  - Counter
  - Deffie-Hellman parameters
  - Host Identity (might be encrypted)
  - Transport format list holding the chosen transport format
  - HIP MAC
  - HIP Signature

- Finally the responder receives I2 and checks the puzzle solution and computes the session key from the Deffie-Hellman parameters. The final packet R2 just contains the signature and the HIP MAC. This packet is used to prevent replay attacks.

Protection against DoS attack

The purpose of the cryptographic puzzle is to protect the responder against the Denial of service (DoS) attack. This allows the responder not to hold a state until I2 is received. The basic mechanism on how this works is as follows. The puzzle exchange starts in the I1 packet as stated previously, the responder provides a random number I and wants the initiator to return a number J. The returned J should be such that when passed through the equation

$$Hash[I + InitiatorHIT + ResponderHIT + J] \tag{2.1}$$

The lowest order of K bits of the returned hash must be zeros. Here K determines the level of difficulty. The responder would use the received J in the above equation to verify if the initiator completed the assigned task. Based on the validation the responder will decide on whether to continue with the association or drop it.

Replay protection

Responders are protected against the replay attacks by not holding a state until a solution for the puzzle is returned. The initiator is protected against replay attack by the monotonically

increasing counter in the I1 packet. This is 64 bit counter which can be initialized to any value.

Downgrade Protection

This involves an attacker modifying the HIP packets such that the initiator or responder is forced to choose a poor cryptographic suite than what both parties could support. Most of the cryptographic algorithm negotiations are signed except for initial Deffie-Hellman group list (DH_LIST), which is being sent in plain in the I1 packet. To avoid a downgrade attack the responder should pick one of the groups from the list and include its own list in the signed part of the R1 packet. When the initiator receives the R1 packet it can cross check the group chosen with its DH_LIST and judge the decision made by the responder based on the DH_LIST included in the packet. If it suspects any downgrade attack, it can drop the association or start from the beginning.

Update and Notify packets

The UPDATE packet is used for updating an existing association. The update might be for one of the following reasons, rekey expiring security associations, add new security associations, or change IP address associated with hosts. UPDATE packets contain monotonically increasing sequence number and are explicitly acknowledged by the peer. They are protected by signature and message authentication code. NOTIFY packets are used to let the other party know about the possibility of an error or any information in common. Unlike UPDATE packets these are neither acknowledged nor protected by signature. No state changes should be made based on the NOTIFY packets.

2.4.2 Diet Exchange(HIP-DEX)

The diet exchange flavor of HIP was built with resource constrained devices in mind. This is built on the basis of HIP-BEX with some modifications to match the constrained nature of the devices. As sensor nodes are resource constrained by default HIP-DEX is the best

choice for them. Some of the important changes in HIP-DEX with respect to HIP-BEX are [4]

- Minimum collection of cryptographic prmitives:
  - Static Elliptic Curve Diffie-Hellman key pairs for encryption of the session key.
  - AES-CTR for symmetric key encryption and AES-CMAC for MACing.
  - A simple fold function for HIT generation.
- Forfeit perfect forward secrecy by dropping ephemeral Diffie-Hellman.
- Forfeit of digital signatures with removal of hash functions.
- Diffie-Hellman derived key only used to protect HIP packets. A separate secrete exchange within the HIP packets creates session key(s).
- Optimal retransmission strategy tailored to handle the potentially extensive processing time of the cryptographic protocols.
- Host Identifiers are generated using the Elliptic Curve Diffie-Hellman (ECDH) key exchange no additional algorithms are supported in this exchange.
- The default puzzle value is set to zero, only changed if the responder senses a threat of Denial of Service (DoS). Cipher-based Message Authentication Code(CMAC) is used instead of RHASH in solving the puzzle.

Two different security associations are formed in case of the HIP-DEX, one is the Deffie-Hellman derived key or the *Master key* and the other is for *Session or Pairwise key*. Master key is used for protecting very few elements and hence it's long lived and doesn't require rekeying. While the session key used to authenticate and encrypt user data is refreshed using *UPDATE* packet.

2.5 WSN Architectural Support

My work assumes the availability of several basic protocols and toolkits for building DrHIP.

2.5.1 IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL)

RPL is used for IPv6 routing over constrained networks [31]. RPL utilizes Destination oriented Directed Acyclic Graphs (DODAG) in establishing multihop routing. The packet forwarding decisions are taken by the DODAG root, which is the single node through which all the nodes in an RPL instance can be reached. There can be multiple RPL instances in a single network each comprising of its own DODAG root. My work assumes the presence of an RPL-like protocol for multi-hop routing

2.5.2 Contiki

Contiki is an open source operating system designed for the internet of things [32]. It provides powerful low-power internet communication. It has full support for IPv6, IPv4 and other lower power standards: 6lowpan, RPL, and CoAP. Contiki applications are written in C, they use protothreads for thread implementation. Each program is written as a separate process. Recent versions of Contiki have inbuilt network simulator the *Cooja network simulator*. The network simulator has support to wide variety of motes which can be used to test the code before uploading them to the physical motes.

2.5.3 Zolerita Z1 motes

For this implementation we have used the Z1 motes, which are equipped with MSP430F2617 low power microcontroller, which features a powerful 16-bit RISC CPU @16 MHZ clock speed, built-in clock factory calibration, 8 KB RAM and 92 KB of Flash memory. This also includes the CC2420 transceiver which operates at 2.4 GHz.

2.5.4 Relic Toolkit

Relic is cryptographic meta-toolkit, which offers different cryptographic algorithms with flexibility and efficiency [33]. The supported algorithms include

- Multi-precision integer arithmetic

- Prime and Binary field arithmetic
- Elliptic curves over prime and binary fields (NIST curves and pairing-friendly curves)
- Bilinear maps and related extension fields
- Cryptographic protocols (RSA, Rabin, ECDSA, ECMQV, ECSS (Schnorr), ECIES, Sakai-Ohgishi-Kasahara ID-based authenticated key agreement, Boneh-Lynn-Schacham and Boneh-Boyen short signatures, and Paillier and Benaloh homomorphic encryption systems)

2.5.5 Byte AES

Byte AES is a slower version of AES capable of operating on systems with byte operation capabilities alone [34]. This consumes very little processing power, hence it can be used on embedded systems and thus sensor nodes.

Chapter 3: Protocol

The DoS resilient HIP(DrHIP) protocol is similar to the HIP-DEX protocol, expanded to include group key management capabilities and to account for the resource constraints. In this Chapter we will elaborate on the design of the protocol starting with an overview, which will be followed by details on packet exchanges, node associations, multihop routing, and DoS resilience.

3.1 Overview

The DrHIP protocol includes a four way handshake, along with node authentication before allowing a node to join the network. Each node association will involve two keys. The first is the symmetric key between the node and the gateway and the second is the group key, which is shared by all nodes under a particular gateway. A node must be authenticated by an authentication server before being made part of the network.

3.1.1 Network Setup

The network setup on which the DrHIP protocol is expected to work is depicted in Figure 3.1. Here AS stands for the Authentication Server, which is assumed to have the public keys of all the nodes that are expected to join the network. Here we take advantage of the design of HIP protocol, where the public key of a public/private key pair is used to uniquely identify the host. It is called the Host Identifier (HI) and a hashed version of it is called the Host Identity Tag (HIT). The HIT acts as the operation representation as discussed in Section 2.4. So, the AS will hold all the host identifiers along with their corresponding HIT for all the nodes that could be part of the network. GW stands for the Gateway, which acts as a middle man between the AS and the nodes. In order to be authenticated and

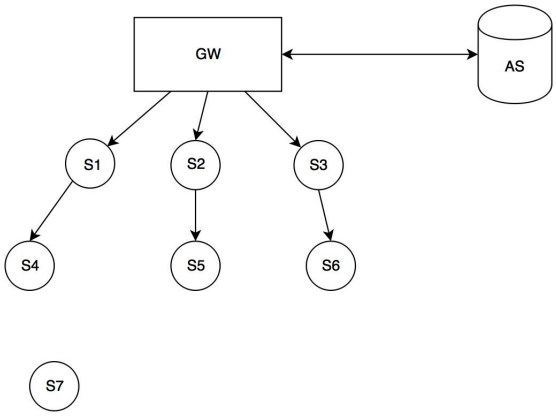


Figure 3.1: Network Setup

become part of the network, the nodes should reach the gateway and take part in the four way handshake. Both the GW and AS are assumed to have sufficient computational power. S1-S6 represent the nodes which are already part of the network and S7 is the new node which is trying to join the network.

3.1.2 Join the network

For a node to join the network it needs to be authenticated by the AS and then establish an association with the GW. For example, the node S7 in the Figure 3.1 will have to reach the GW so as to initiate the association. This leads to two scenarios through which the node can reach the GW.

- Directly connecting to the GW, for this to be achieved the GW should have an established one hop connection to the node.
- Reach the GW through other nodes that are already part of the network via multi-hop routing.

If the node has a direct link to the GW, it will request for association and will be authenticated by the AS. The other possibility is for the node to be not able to reach the GW and will have to depend on the other nodes which are already part of the network

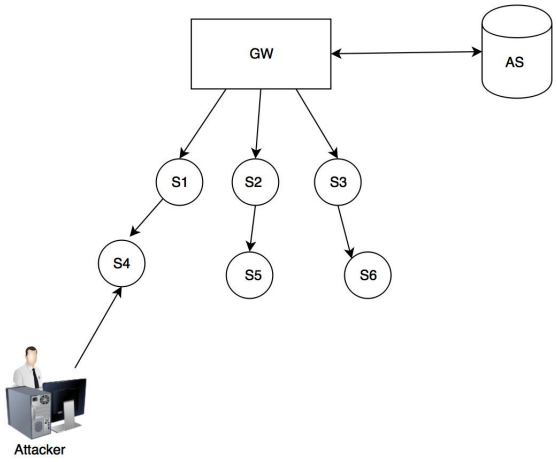


Figure 3.2: Threat Model

in joining the network. This involves multi-hop routing using a protocol such as RPL. In our example network setup seen in Figure 3.1 node S7 is not within reach of the gateway, hence it will pick the second option. Now the nodes between the GW and S7 will act like as relay nodes and enable S7 in becoming part of the network. Here, if S7 happens to be a malicious node, and floods the other nodes with requests to reach the GW, it can exhaust the resource of the nodes in the path to the GW leading to a DoS attack.

3.1.3 Threat model

In Chapter 2 we have discussed some of the threats posed to WSNs. Due to unattended wireless operation, wireless sensor networks are susceptible to include Denial of Service (DoS), Sybil attack, Traffic analysis attacks, Node replication attacks, Attacks against privacy, Physical attacks, etc. [16]. Of these attacks DrHIP focuses on the DoS attack, which involves denying some of the services or sometimes all of the services. When it comes to sensor networks this is something which could be achieved with relative ease as the devices forming the network are vulnerable to resource exhaustion. Resource exhaustion in sensor networks can be achieved in several ways like, draining out the power by forcing unwanted transmission, exhaust the memory by forcing complex computations, etc.

In the Figure 3.2 we have the same network setup with which we started our discussion, but this time for a difference instead of the benign node S7, we have a malicious attacker. The attacker is assumed to have arbitrary computational power and resources. Now this attacker is communicating with node S4. In order to these attacks to work S7 must be authenticated by the system, in this case the AS. We assume that this authentication mechanism itself cannot be breached by the attacker. However, even if the AS prevents the attacker from becoming authenticated, the malicious node can still disrupt the network with bogus authentication requests. In this example S7 can flood the node S4 with multiple requests and force it to exhaustion. Once S7 is done with the node S4, it can move on to the next node and thus DoS large parts of the network over time.

Mitigation of this attack is to stop the attacker right in the periphery of the network and therefore limit the effectiveness of the DoS against the authentication mechanism. The DrHIP protocol has been designed with this in mind. The motive of the protocol is to thwart the attacker in an effort to prevent penetrating deep inside the network. For this to be achieved the protocol should have the ability to discriminate between normal packet flow and packet flow during times of attack. In upcoming sections we will elaborate on how the DrHIP protocol achieves this and how well it's in stopping the attacker.

3.2 Packet exchanges

Each packet includes the source HIT and destination HIT by default unless the packet is a broadcast packet, in which case the destination HIT is left blank. There are a total of four packet exchanges between the node and the GW before a node is authenticated and made part of the network. The Figure 3.3 shows the different packet exchanges between the new node Sn and the GW. The first packet is the HELLO_PACKET, in this packet the new node includes a predefined identifier common to the network encrypted with its private key. This packet is sent as a broadcast in order to reach the GW. If the packet reaches the GW, the GW will forward this packet to the AS and AS will authenticate the node and recommend the GW to continue with the association.

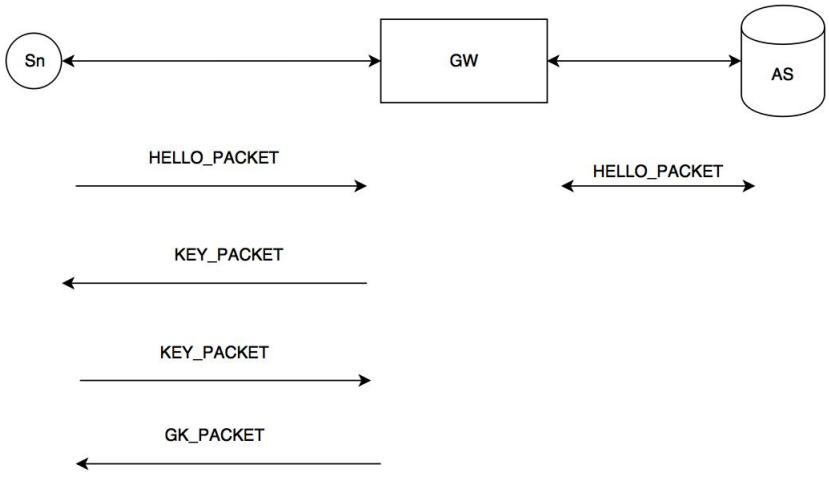


Figure 3.3: Packet Exchanges

Once the node is authenticated, the GW will send the next packet which is the KEY_PACKET to the node as a unicast. This packet will contain the public key of the GW. Now the node will use ECDH in generating the symmetric key from the received public key of the GW. Once the node generates the symmetric key, it will include its public key in the next packet which will also be a KEY_PACKET and send it to the GW. Once the GW has received the public key of the node, it will also generate the symmetric key using ECDH and rekey the group key. The final packet from the GW is the GK_PACKET which contains the new group key encrypted with the recently established symmetric key between the node and GW. Also the GW sends the newly generated group key to the other nodes which are part of the network. With this the packet exchange ends and the node becomes part of the network.

3.3 Group Key generation and rekeying

Group key is used for authenticating internode communications and the communication between the nodes and the GW, once a node becomes part of the network. Hence it is required that the group key should be regenerated every time a new node joins the network or an existing node leaves the network. The group key is generated as a cryptographic hash

over all the symmetric keys of the nodes which are currently part of the network along with some random bytes. If there are n nodes which are part of the network then the equation used for generating the group key is given by

$$Groupkey = Hash[S1Key + S2Key + ... + SnKey + Randombytes] \quad (3.1)$$

Here the random bytes is used to ensure perfect forward secrecy. If the group key is generated without using the random bytes, whenever a node leaves the newly generated group key would be the same as the one before the previous one.

3.3.1 Active nodes

The GW has to maintain a list of active nodes to know exactly the nodes that are part of its network, for this the GW runs a periodic check to determine who is active. This is achieved via the BEACON_PACKET in the DrHIP protocol. The BEACON_PACKET is sent as a broadcast in regular intervals of time. Once the BEACON_PACKET is received the node has to respond back with an ACTIVE_PACKET. If a node doesn't respond back within a particular period of time, the GW assumes that the node is not active anymore and removes the node from its current list of active nodes. This is followed by the rekeying of the group key.

3.4 Multihop

When a node cannot reach the GW directly, it will have to depend on the existing nodes which are already part of the network in reaching the GW. For this to be achieved each node must know its position in the existing network and which would be the best path to take when there is more than one way to reach the GW. This is achieved using mechanisms derived from the RPL protocol [31]. In RPL routing and forwarding decisions are made on the basis of a metric called "Rank." DrHIP assumes that each node has to know its Rank in the particular network and the way to reach the GW. In the DrHIP protocol the Rank

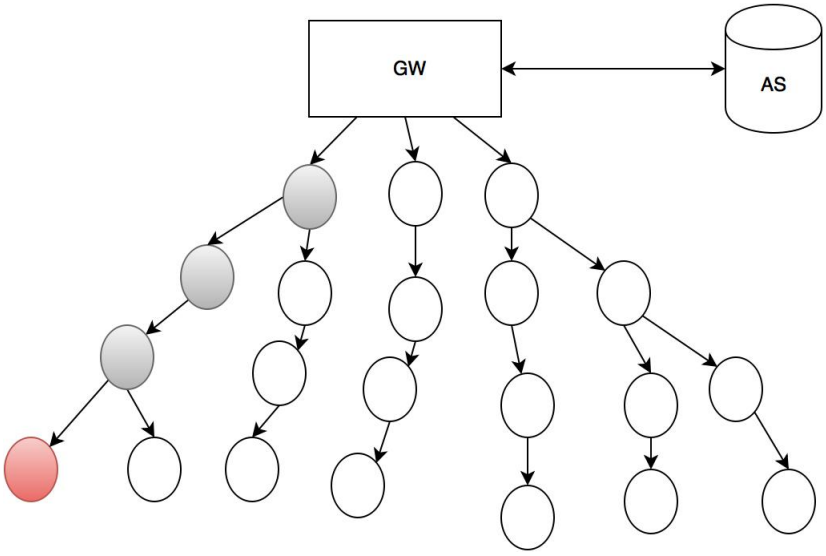


Figure 3.4: DOS attack

represents the position of the node with respect to the GW.

Periodically the GW sends a ROOT_PACKET with a Rank of zero as a broadcast, any node receiving the packet will update their Rank based on the Rank that was received and forward the packet down the network. This way all the nodes that are part of the network will know their rank with respect to the GW and on how to reach the GW. Now when the new node sends its HELLO_PACKET, it will be received by the nearest node which is part of the network and will be forwarded to the gateway. The association between the new node and the GW will be relayed by the intermediate nodes.

3.5 Denial of Service threat

Whenever a new node is trying to join the network and it does not have one hop connectivity with the gateway, the only way it can reach the gateway is through other nodes which are already part of the network. While this can be established in a multihop fashion, if the new node is not benign and happens to be malicious, there is a good chance that the attacker can initiate a DoS attack on the nodes by taking their resource limitation into consideration.

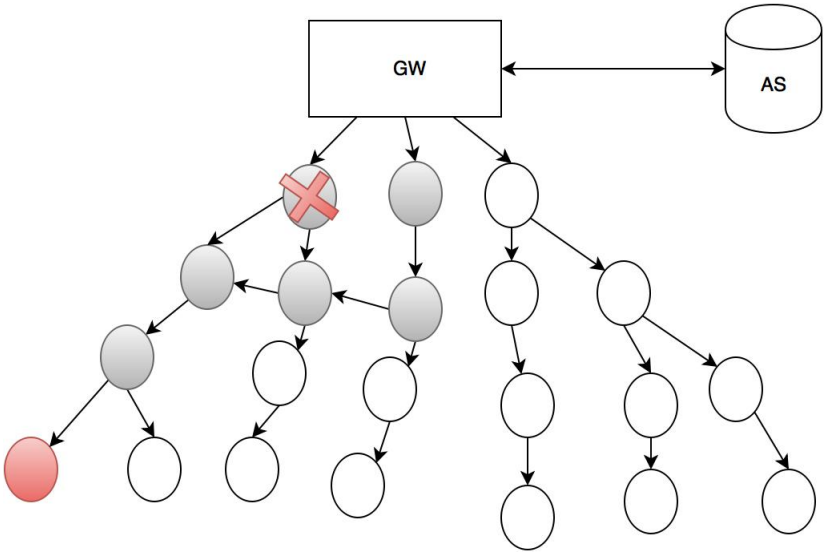


Figure 3.5: DOS attack after one of the nodes is down

So, as discussed in Section 3.1.3 our primary motive here is to thwart the effort for DoS attacks as early as possible, because if the attacker sustains for a period of time he can penetrate deep into the network.

We can use the puzzle implementation of the HIP-DEX in thwarting the DoS attacks, but, for this to be achieved the network needs to have knowledge on when there is a threat and when there is not a threat. The WSN should be able to collectively sense the DoS attack and utilize the puzzle. Here the problem is on how to do this with little utilization of resources and also not to be fooled with false positives. Here the penalty for false positive might be high, as the puzzles might prove to be complex for legitimate nodes to solve. Also, we want to prevent the attack right on the periphery of the network before it gets any deep. For example, refer the network setup in Figure 3.4, here an attacker trying to join the network will have to take one of the seven peripheral nodes available. If we assume that the attackers initial HELLO_PACKET reaches the node colored in red, it will take the path highlighted in grey for its communications to the gateway.

If an attacker is trying to cause the DoS attack, it is usually easy to exhaust the resource of the nodes closer to the GW, especially when one takes into account the fact that these

nodes are usually highly utilized. Here we are assuming that all the nodes in the network have equal resources. Now when the attacker is flooding with packets along the path highlighted in gray in Figure 3.4, if our assumption on the nodes closer to the GW being highly utilized is correct, then the node that is close to the GW would be the first one to go down. Once this node is down, the previous path taken to reach the gateway is no longer available hence the attacker has to reach the GW through the new path as seen in the Figure 3.5. This way attacker will slowly crawl through the entire network exhausting most of the nodes one by one.

3.6 Defense with the puzzle

In our attempt to hold the malicious attacker right on the periphery we utilize the puzzles that are part of the HIP protocol. The puzzle basically mitigates the effect produced by the malicious attacker, by reducing his activity. Here the attacker is tasked with finding the solution for the puzzle which as detailed in section 2.4.1 might require multiple trials on the attackers side before he comes up with a solution. The computational power of the attacker will be the deciding factor on how much time it will take for him to solve the puzzle. The puzzle difficulty can be used in combating the computational power of the attacker, but, care should be taken on setting up the puzzle difficulty as highly difficult puzzles might be impossible for legitimate nodes to solve with their limited resources.

The decision on when to deploy the puzzle is taken by the GW. In taking this decision the GW considers the current network topology and the number of incoming connections over a particular window. If it suspects a sudden increase in incoming connections, it will put the puzzles into action. While increase in incoming connections is a cue for anomaly, does this guarantee the existence of an attacker? To confirm this GW gets the help from the AS, while checking for the number of incoming connections over a particular window the GW also looks for the number of failed authentications and analyses if there is anything suspicious in this. This gives the GW a clear picture on if there is an imminent threat of DOS.

Having arrived with the decision to deploy the puzzle, the next question is where to deploy the puzzle? In deciding on where to deploy the puzzle we are considering two possible locations, one is at the GW and the other one is at the peripheral node, which acts as the port of entry for the attacker. One big advantage of deploying the puzzle at the GW is that, the GW has better computational power and it can handle several puzzle validations parallely. While this is an advantage there is also a disadvantage to this, the motive behind using puzzles is to hold the attacker right at the periphery. But, if we deploy the puzzle in the GW every time the puzzle solution has to be evaluated it has to reach the GW, this defeats the purpose.

Hence, the DrHIP protocol is designed to deploy the puzzle at the periphery of the network. But the decision to use the puzzle is always made by the GW, on having made the decision the GW signals the node at the periphery to utilize the puzzle going forward. Along with the signal, the GW also provides the information on puzzle difficulty to the peripheral node. From this point whenever the peripheral node receives a HELLO_PACKET destined to the GW, it replies the initiator (node which sends the HELLO_PACKET) with the puzzle. The initiator solves the puzzle and includes the solution in the next HELLO_PACKET destined for the GW. If the solution included is correct the peripheral node would forward the packet to the GW, else it will be dropped. The complete defense is based on the fact that it takes some time for the attacker to solve the puzzle and more the time less the impact.

Chapter 4: Implementation

The protocol is implemented on the Contiki Operating system and the performance evaluation is done on the Cooja simulator. The purpose of the implementation is to focus on the interaction between four different types of devices AS, GW, sensor nodes, and the attacker. As discussed in Section 3.1.3, our implementation assumes that the channel between the GW and the AS is secured and immune against compromise.

4.1 Authentication Server(AS)

The AS serves as the single point authenticator for all the join requests. As such the AS holds the list of public keys of all the incoming nodes, along with their HIT's, as explained in Section 3.1.1. In the implementation the only function of the AS is to receive the incoming HELLO_PACKET from the GW and let the GW know if the node is to be trusted or not. Each nodes' HELLO_PACKET will include a HIT and a passphrase encrypted with the nodes' private key. Once the AS receives the HELLO_PACKET it will look for HIT in the lookup table and try to decrypt the received encrypted text with the public key on the lookup table. If the node is actually the one that it claims to be, then the public key is used to decrypt the cypher text and thus authenticating the node. We disregard any possibility of an attack in the channel between GW and AS.

4.2 Gateway(GW)

The GW plays multiple roles including processing nodes that wish to join, managing the group key, advertising its presence, keeping track of active nodes, and preventing the DOS attack. To accomplish this the GW has far greater computational power than the other

Table 4.1: Parameters of SECG P-160 Curve.

Parameter	Value
SECG_P160_A	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFC
SECG_P160_B	C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45
SECG_P160_X	4A96B5688EF573284664698968C38BB913CBFC82
SECG_P160_Y	23A628553168947D59DCC912042351377AC5FB32
SECG_P160_R	100000000000000000001F4C8F927AED3CA752257
SECG_P160_H	1

nodes. There are multiple GW's throughout the network and the incoming node will chose the one in its proximity. The GWs are provided with the ability to directly reach the AS and verify the credibility of the incoming node. The GWs are also aware of the entire network structure and utilizes Routing Protocol for Low-Power and Lossy Networks (RPL) for the routing purposes.

4.2.1 Accepting the incoming nodes

In accepting the incoming nodes the gateway uses a HIP four way handshake starting with the HELLO_PACKET. Here according to HIP the incoming node acts as the initiator while the GW acts as the responder. Along with the required information for the node to be authenticated the HELLO_PACKET should also include the Deffie-Hellman(DH) parameters for the nodes to be able to establish the association. In our implementation we assume that the GW and the incoming nodes utilize the same DH parameters hence this information is excluded from the HELLO_PACKET. Once the node is authenticated by the AS, the GW sends the KEY_PACKET to the node, which includes the public key of the GW obtained from the elliptic curve public/private key pair. Once the node receives the KEY_PACKET it will use the obtained public key in deriving the symmetric key and then reply the GW with its public key in another KEY_PACKET. On receiving the KEY_PACKET the GW would create the symmetric key and end the handshake by sending the newly joined node with the group key in the GK_PACKET.

Cryptographic algorithms

All the cryptographic algorithms used in establishing the association are implemented using the cryptographic library Relic [33]. The relic library supports several cryptographic algorithms and takes a large amount of the node's physical memory. In order to overcome this we had to optimize the library by removing some of the functions which were not relevant to our implementation. We have implemented elliptic curve over prime finite fields using the elliptic curve SECG P-160. The parameters of the curve are shown in the Table 4.1. The Relic library depends on the operating system in obtaining the seed for the random number generator, without a perfectly random seed our implementation won't be secure. To overcome this we used the sensor data gathered by the nodes in generating the random seed. Figure 4.1 illustrates on how the random seed is being generated from the sensor data. It works by using sensor readings as initialization vector and initialize the counter with some random value. The key for the startup is being generated using the random number generator which is inbuilt in Contiki. This key is used to encrypt the counter value with the sensor data as initialization vector. After this the counter is incremented and stored along with encrypted data in a file. For the next time the encrypted data from the file will be used as the key and the process will be repeated for future generations.

4.2.2 Group Key management

The group key is common to all the nodes that are under the management of a particular GW. The group key is derived and distributed by the GW. The GW utilizes the symmetric key of the existing nodes along with a nonce, in generating the group key as discussed in the Section 3.3. The group key is generated whenever a new node joins the network or whenever an existing node leaves the network. In both instances the GW generates the new group key and forwards it to the existing nodes in the list. In our implementation the GW runs a separate process which is invoked every time a new group key is generated. A total of two packet exchanges are involved in distributing the group key to a node as seen in Figure 4.2. The first packet is the GK_PACKET which is sent from the GW to the node. It contains

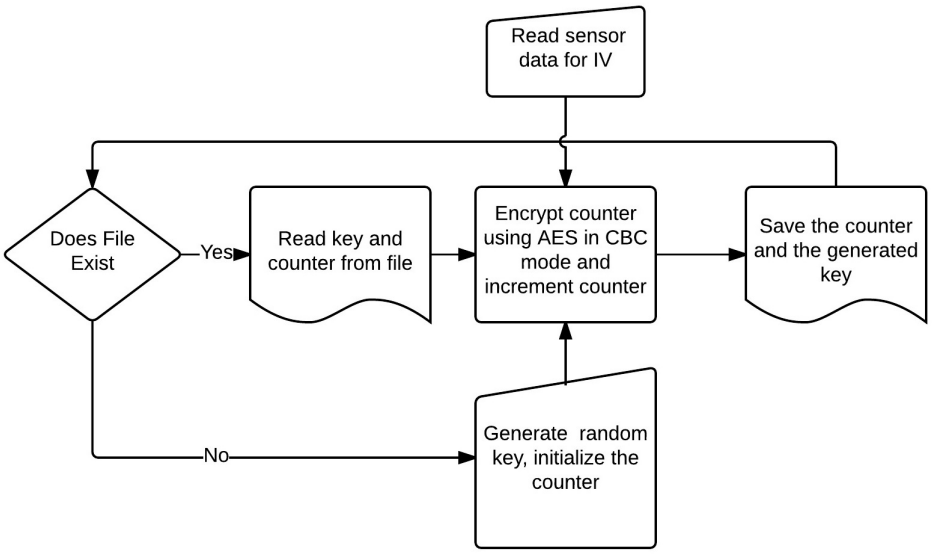


Figure 4.1: Generation of random seed

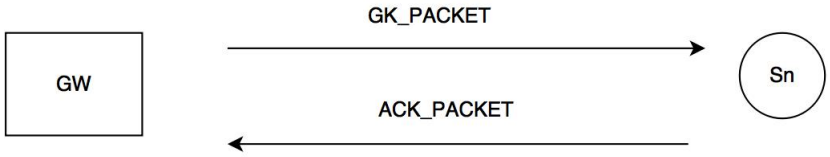


Figure 4.2: Packet exchanges in group key management

the current group key which is encrypted with the symmetric key shared between the GW and the node, along with a nonce. This is followed by the ACK_PACKET which contains the nonce previously received in the GK_PACKET encrypted with current group key, from the node to the GW. The ACK_PACKET confirms the reception of the group key by the node. The GW waits for the ACK_PACKET and if not received over a period of time, it retransmits the GK_PACKET twice before confirming the node is down.

4.2.3 Root discovery and multihop routing

The protocol behaves in a similar fashion to the RPL protocol, in establishing multihop routing and root discovery [31]. The GW acts as the sink and each node has a single path

to the GW resulting in a Designation Oriented Directed Acyclic Graph(DODAG) similar to that of RPL. For simplicity our implementation assumes a single root which is the GW. The position of each node with respect to the GW is given by the Rank, also in a fashion similar to the RPL. New nodes joining the network utilize Rank as a metric in deciding which node to be a neighbor with.

The GW sends the ROOT_PACKET periodically as a broadcast, in which it includes its Rank. When other nodes in the network receive the ROOT_PACKET they utilize the Rank obtained in the packet in determining their Rank and update the packet with their Rank and rebroadcast it. This is repeated until all the nodes in the network determine the GW's location. Along with the Rank the ROOT_PACKET also includes a nonce, both the Rank and the nonce are encrypted with group key thus protecting against replay attacks.

Multihop routing is facilitated by the GW, whenever a node has a packet destined for another it will keep forwarding the packet upward until it reaches the GW similar to the RPL protocol where in the packet is pushed to DODAG root, which decides on where to send the packet. The GW uses source routing in sending the packet to the required destination. As our implementation doesn't extend beyond a single DODAG, we disregard the possibility that a destination will not be found in the current DODAG. In the event of this happening in our implementation the GW will just drop the packet.

4.2.4 Active nodes

The GW has to keep track of the current nodes that are active, and revokes the nodes that are not active anymore. To achieve this the GW maintains a list of nodes that are part of the network and probes them periodically to determine if they are active. If all the nodes are active no further action is required, but if any of the nodes is not active then the GW has to remove it from the list and rekey the group key. To accomplish this we have utilized the LIST data structure which is part of Contiki. Using the LIST data structure we created a custom list called the neighbor list, which holds the HIT of every single node which established an association with the GW along with its corresponding symmetric key.

```
for(n=head(neighbor_list);n!=NULL;n=n.next)
{
    encrypt(ACTIVE,nonce,n.key);
    unicast_send(ACTIVE,n.HIT);
}
wait(30);
for(n=head(neighbor_list);n!=NULL;n=n.next)
{
   if(received(AKC,n))
          set(n,active);
  else
          {
          set(n,inactive);
          node_revoked=true;
          remove(n,neighbor_list);
}
if(node_revoked)
          rekey(group_key);
```

Figure 4.3: Pseudocode for probing and revoking inactive nodes

Every time a node joins the network it gets added to the list. The GW has to periodically probe the nodes on this list. To accomplish this we have created a separate process that periodically sends an ACTIVE_PACKET to every single node on the list. On receiving the ACTIVE_PACKET the node replies with an ACK_PACKET. ACTIVE_PACKETs are sent to the individual nodes hold a nonce encrypted with the symmetric key shared between the node and the GW, and the replied ACK_PACKET contains the nonce encrypted with group key.

Figure 4.3 shows the pseudocode used in accomplishing the node revocation. First each node is sent an ACTIVE_PACKET which is followed by a wait time for the nodes to respond, the wait time is calculated based on the size and distribution of the network and it is a dynamic parameter. Once the wait time is over, the GW checks if ACK_PACKET was received for each of the nodes in the list, if that is the case then nothing has to be done. If any of the nodes didn't reply then the nodes are removed from the list, followed by rekeying of the group key.

4.2.5 Preventing Denial of Service Attacks

The GW is the one responsible for sensing DoS attacks and making the decision on where to deploy the puzzle, along with its difficulty. In our implementation the GW maintains a fixed time window and monitors for the current packet flow and number of authentication failures over that window. The major aim of the GW is to maintain the packet flow and mitigate the damage done by the attacker by reducing his activity. The GW utilizes an iterative algorithm in deploying the puzzle. First the GW monitors the packet flow and failed authentications over a fixed window. If the values exceed a fixed threshold then the gateway deploys the puzzle. The threshold is determined by monitoring the network during peak load time along with data gathered from network activity including network management operations.

When deploying the puzzle the GW uses the smallest difficulty possible, and checks if the network reverts back to normal functioning in the next window. If threshold is breached

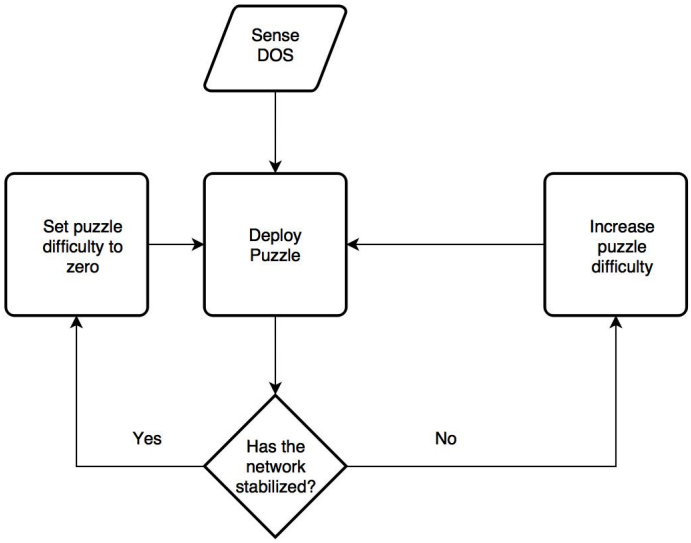


Figure 4.4: Steps taken by the GW in DOS prevention

again then the GW increments the puzzle difficulty, and this is repeated until the network stabilizes. Once the network is stabilized the GW removes the previously deployed puzzle. Figure 4.4 shows the sequences of steps the GW takes in deploying the puzzle and stabilizing the network.

4.3 Sensor Nodes

In the implementation each node is a Zolerita mote trying to locate the GW and join the network. The nodes are aware of their position on the network, address of the GW, and how to reach the GW. In our implementation the node wakes up and broadcasts a HELLO_PACKET over random intervals of time, until it receives a reply from the GW. The interval time is chosen randomly to reduce the possibility of network congestion. Now as discussed previously, if the node is not within one hop of the GW, then those nodes that are already part of the network relay the new node in reaching the GW. For this to be achieved the nodes are implemented with the ability to identify if an incoming broadcast is a HELLO_PACKET; if that is the case then the node forwards the packet to the GW in a multihop fashion.

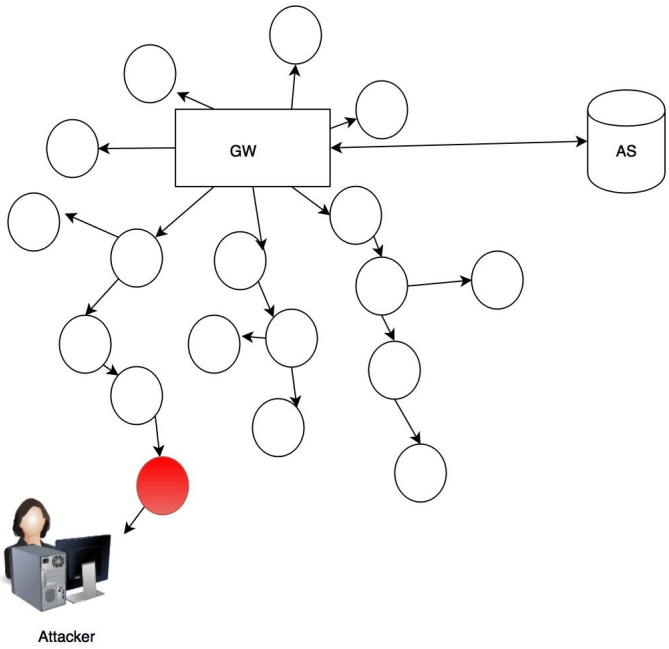


Figure 4.5: Node interactions in DOS prevention

Each node has a single process running, the purpose of which is to make sure the node is always part of the network. If a node loses its association due to power failure or any other problems, the process tries to reestablish the association. There has been some work done on the resumption of association by holding state of the existing association [23], but, in our work we have not tried to address this problem. Along with establishing and maintaining its association within the network, each node also takes part in GW discovery and DoS prevention.

4.3.1 GW discovery

All sensor nodes participate in multi-hop routing. The sensor nodes actively participate in ROOT_PACKET dissemination, thus allowing other nodes which are not within the reach of GW to determine the position of the GW. As discussed in Section 4.2.3 the sensor nodes forward the packet to the GW and the GW uses source routing in deciding on which path to take to the destination.

4.3.2 Sensor Nodes as the First Line of Defense

Individual nodes play a major role in DoS prevention in our implementation. When the network is under attack by a malicious attacker, the sensor node which serves as the port of entry for the attacker will act as a barricade by asking the attacker to solve a puzzle before forwarding his packets into the network. Although the decision on when to use puzzles is made by the GW, the sensor node is where the puzzle is deployed. The DoS prevention mechanism starts with the GW sensing a threat, followed by deployment of the puzzle which is at the periphery of the network and acts as a port of entry to the attacker.

The decision on which node to choose in deploying the puzzle is also made by the GW. For example in the Figure 4.5 the node in red is the one which the attacker reaches to get inside the network. Hence the GW will deploy the puzzle in this node. Going forward whenever a HELLO_PACKET is received by this node, it will reply back with a PUZZLE_PACKET, the PUZZLE_PACKET contains three things, a random number I, difficulty K, and expiration time for the puzzle. The value for I is determined from hashing of secret value S which is regenerated periodically and provided by the GW as per [29]. Now the value for I is determined as follows

$$I = Ltrunc(RHASH(S|INITIATOR - HIT|RESPONDER - HIT), 64)] \qquad (4.1)$$

The value for K is also provided by the GW based on the level of the attack the network is currently undergoing. The solution is verified as explained in Section 2.4.1.To understand the packet exchanges, let us assume two nodes one is the initiator(this is node which is trying to join the network), the other is the responder(which is preventing the DoS by initiating the puzzle). As seen in Figure 4.6 the first packet is the HELLO_PACKET from the initiator, which is followed by the PUZZLE_PACKET from the responder. Now the initiator solves the puzzle and includes the solution in the HELLO_PACKET which the responder receives and verifies the solution and forwards the packet to the GW if the solution is correct. Along with verifying if the solution is correct the responder also makes

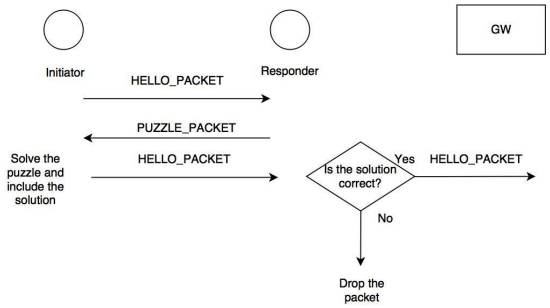


Figure 4.6: Packet exchanges in puzzle mechanism

sure that the solution arrived before the expiry time.

4.3.3 Sensor data

Once the node joins the network after the four way handshake, it will start gathering sensor data and send it to the GW periodically. In our implementation, the sensor data is sent via the DATA_PACKET in regular intervals of time. The sensor data is encrypted with group key and includes a monotonically incrementing sequence number to avoid replay attacks.

4.4 Attacker Model

We assume that the attacker is similar to a sensor node in most of its functionality, such as its radio, *except* for its computational power. The attacker is assumed to have arbitrary computational power and in our implementation we try to emulate the attacker with different computational powers to study the impact of the puzzle. The attacker can be from any part of the network. As we are concerned about DoS attacks, the attacker in our implementation keeps sending HELLO_PACKETs. If the attacker can directly reach the GW, the probability of it exhausting the GW's resources is essentially zero so we disregard this scenario. Our implementation is only concerned about an attacker who tries to exhaust the nodes that are on its way to the gateway. The attacker is equipped with the necessary functions to solve the puzzle and its efficiency is dependent upon its computational power

and the puzzle difficulty.

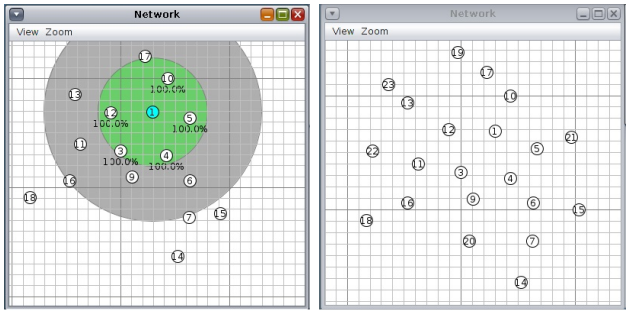
Chapter 5: Evaluation

The performance of our implementation was evaluated using Cooja simulator in the contiki operating system [32]. The evaluation involved measuring the throughput of the network under different scenarios. Some of the scenarios we considered are varying the number of nodes, the impact of the attacker with and without the puzzle, changing the number of hops, variation of puzzle difficulty, and variation in the attackers intensity. The network setup is quite similar to the one discussed in Section 3.1.1, and throughout this evaluation we assume to have a single attacker positioned at different places in the network. As stated in Section 4.4 we disregard the possibility of the attacker trying to exhaust the GW and focus on the intermediary nodes between the attacker and the GW. The attacker is assumed to have arbitrarily large computational power.

5.1 Experimental Setup

For the experiments we considered four different network setups. Each of the network setups varied in the number of nodes, and we performed similar experiments in all the four network setups. For different network setups, we increased the number of nodes by five in each network setup, starting with five nodes as minimum. We therefore have our topologies each with five, ten, fifteen and twenty nodes. The number of nodes doesn't include the GW and the attacker. Figure 5.1 shows the different network topologies. In each of the topology node with id 1 is the GW and the rest are sensor nodes. The attacker is not included in the picture. In each topology we tested the performance for six different scenarios, normal operating conditions, in the presence of attacker without deploying the puzzle, and in the presence of attacker with four varying puzzle difficulties. In each of these scenarios, we measure the throughput of sensor data originating from the nodes and

(a) 5 node network and 10 node network



(b) 15 node network and 20 node network

Figure 5.1: Different network topologies

successfully reaching the GW. The runtime for all the experiments was 10 minutes, hence the throughput measured is for 10 minutes.

5.1.1 Measuring performance

In measuring the performance of the network we consider two items: data throughput of the network and the percentage of total throughput that is data throughput – we call this as Goodput– under the different scenarios. In our experiments we consider data throughput as the total amount of data that originated from each node and reached the GW. We only consider the data specific to sensor monitoring and disregard the data that originates in establishing association and other maintenance activities when we calculate the data throughput. On the other hand the total throughput considers every single packet originating from the nodes and reaching the GW, and this includes the packets from the attacker. The runtime for all the experiments was 10 minutes and the throughput is measured in packets, so the throughput show represents the number of packets/10 minutes. The sensor

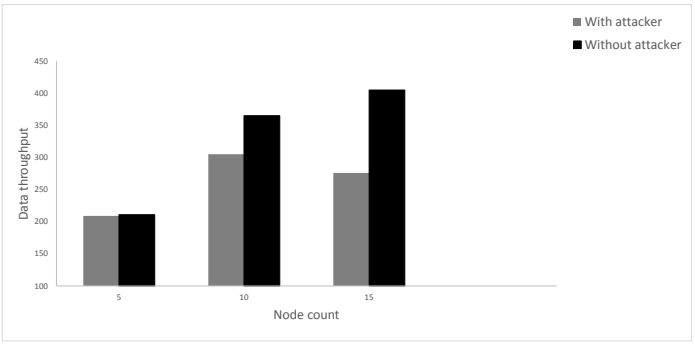
data is gathered periodically in all the sensor nodes and sent to the GW after association as discussed in section 4.3.3.

5.2 Experiment-I

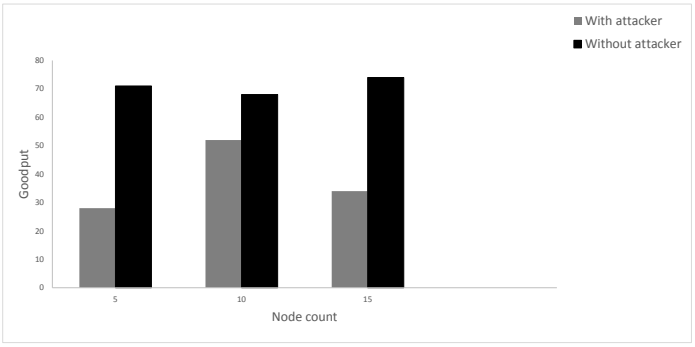
In this experiment we compare the performance of the network under the presence of the attacker and under normal operating conditions. We do not deploy puzzles in the presence of the attacker, we just measure the loss in throughput brought by the attacker in the absence of puzzle. There was a significant decrease in throughput in the presence of the attacker with the increase in node count. The Figure 5.2 shows the data throughput and Goodput of the different topologies both with and without the attacker. Of all the four networks, the one with five nodes did not have a significant change in the data throughput, but the Goodput reduced by 44% in the presence of attacker. The network with just five nodes was not complex enough for attacker to take advantage and bring in exhaustion, and the network was able to accommodate the attacker. For the other experiments, the networks with ten and fifteen nodes had a significant decrease in throughput with the presence of attacker. There was approximately 16% decrease in throughput in the network with ten nodes, and 32% decrease in the network with fifteen nodes. The Goodput also decreased significantly with 24% decrease for 10 node network and 54% decrease in 15 node network. This confirms our initial statement that as the network density increases the effect of the attacker also increases. We were not able to run the simulation for 20 node network, as the increase in network complexity caused the Cooja simulator to fail. Hence, we will be using 5, 10, and 15 node networks for other experiments.

5.3 Experiment-II

In this experiment we evaluate the performance of the different topologies in the presence of attacker and with the use of the puzzle. Specifically we increase the puzzle difficulty in each run and monitor the data throughput and Goodput. We started with a puzzle difficulty of



(a) Data throughput for different networks with and without attacker

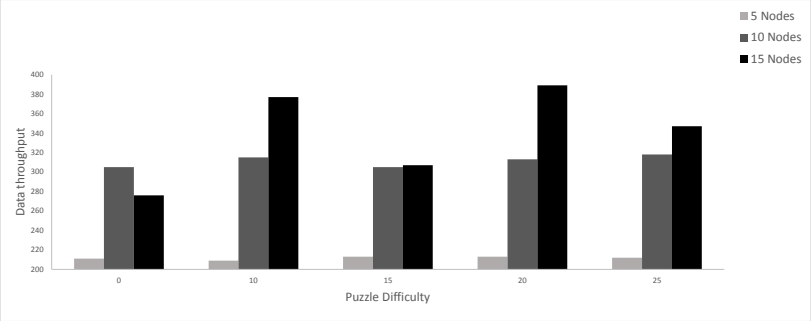


(b) Goodput for different networks with and without attacker

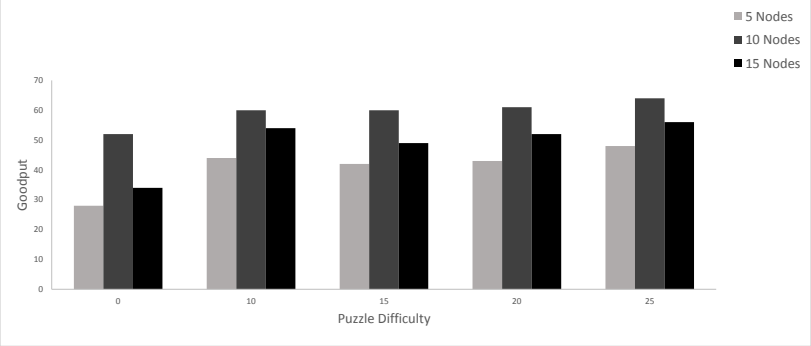
Figure 5.2: Performance of the networks with and without the attacker

zero, followed by difficulty of 10, 15, 20, and 25. The figure shows the data throughput and Goodput for different network topologies and with different difficulties. Here one thing to be noted is the presence of the puzzle for sure improved the Goodput and data throughput for the given experimental time.

In the network topology with five nodes, there was not much difference in the data throughput irrespective of the puzzle for the same reasons discussed in Experiment-I. As far as Goodput is concerned there was significant increase in Goodput with the usage of puzzle. The Goodput increased by 57% with the use of puzzles. Although the run with the puzzle difficulty of 25 had little higher throughput than the rest of the puzzle difficulties, mostly the Goodput was stable with the use of the puzzles. Next with the ten node network there was significant increase in throughput with the use of puzzles, the maximum data throughput



(a) Data throughput for different networks with different puzzle difficulties



(b) Goodput for different networks with different puzzle difficulties

Figure 5.3: Performance of the networks with different puzzle difficulties

was achieved with the puzzle difficulty of 25. While there is a drop in data throughput with the puzzle difficulty of 15, comparing with the Goodput for the same difficulty we can infer that the drop was not just among the data packets, but was equal among the network. The Goodput for the ten node network had 16% increase and the increase was stable across the puzzle difficulties with maximum Goodput achieved for the difficulty of 25.

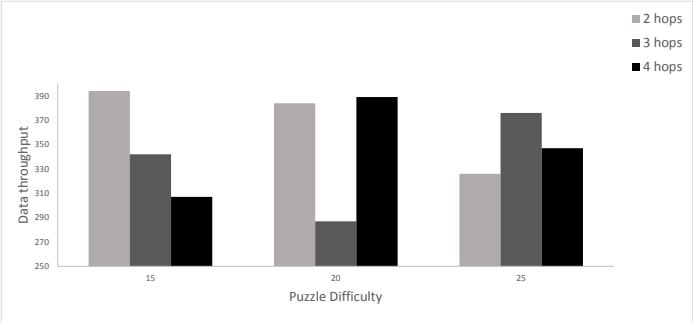
The network with fifteen nodes had a significant increase in data throughput with the use of puzzles with the maximum data throughput achieved for puzzle difficulty of 20. Of the three network setups we compared, this is the one with most increase in data throughput with the usage of puzzles. The maximum data throughput achieved was 40% greater

than the data throughput without the puzzle. The Goodput had an increase of 59%, and the increase was stable among different puzzle difficulties. Although there is difference in Goodput with the different puzzle difficulties, it is clear that the usage of puzzles plays a significant role in thwarting the attacker and bringing stability to the network.

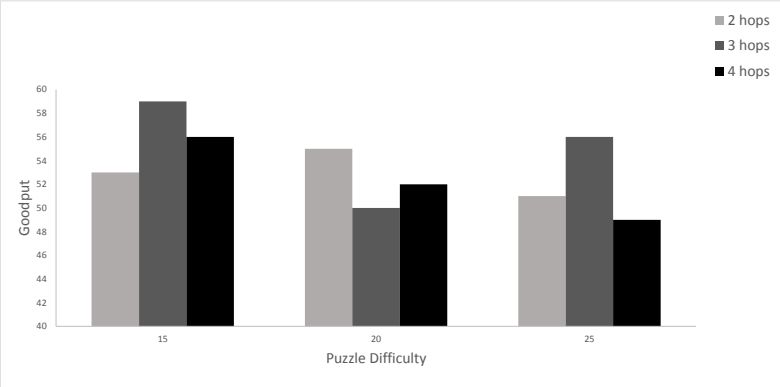
In terms of puzzle difficulty, our experiments showed that there is no single puzzle difficulty that would fit all the network topologies. Hence the best strategy is to try different puzzle difficulties and check which mitigates the attack better in a given scenario. The problem is this has to be done in run time from sensing the level of threat that is being faced from the attacker. We can equip the GW with a lookup table in order to smoothen the process and the lookup table can be learnt from initial runs. This is a potential area for future work.

5.4 Experiment-III

In this experiment we wanted to check the effect of the attacker's position in network. Position means the hop count from the GW. For the first two experiments, the attacker was located at a distance of 3 hops, in case of the five node network, and 4 hops in case of ten and fifteen node networks. We ran the experiment with the attacker in different positions under different puzzle difficulties for just the 15 node network. We ran the experiments with the attacker 2, 3, and 4 hops away from the GW and with puzzle difficulties of 15, 20, and 25 for each position. The results of the experiment are shown in the Figure 5.4, the puzzle difficulties had a different effect on the network based on the attacker's position. As discussed in Experiment-II, this confirms that there is no single puzzle difficulty that would match all the scenarios. Of the three setups for the one in which the attacker was 4 hops away, the puzzle difficulty of 20 had a better data throughput, while in network with attacker 3 hops away, the puzzle difficulty of 25 had a better data throughput and finally the network were in the attacker was two hops away the puzzle difficulty of 15 had a better data throughput. The uncertainty in the sensor network has a huge impact in the performance of the different puzzle difficulties. As far as the Goodput is concerned it was



(a) Data throughput for different networks with attacker in different positions



(b) Goodput for different networks with attacker in different positions

Figure 5.4: Performance of the networks with attacker in different positions

almost stable across the different puzzle difficulties in most of the networks.

5.5 Discussion

From the three experiments we concluded that the usage of puzzle does thwart the attacker's effect and the network scenarios play a huge role in the variation of data throughput and Goodput. For example, of the three networks the network with five nodes was able to accommodate all the packets originating from the attacker, without losing much of its data packets. Thus there was not a significant change in data throughput.

Similarly the puzzle difficulty also plays a large role in the network's performance. For

example, highly difficult puzzles takes more time for the attacker to solve and thus reduce the impact to a great extent than the puzzles with lesser difficulty. While this is true, at times just using puzzles with high difficulty will not solve the problem. Consider the puzzle difficulty of 15 for the fifteen node network, it has less data throughput and Goodput than the run with puzzle difficulty of 10. This has to do with the chosen time window and the threshold for deciding if there is imminent threat or not.

The time window and threshold on deciding if there is DoS threat plays a large role in the effectiveness of a particular puzzle difficulty. For example, let us assume that we have fixed time window of thirty seconds and a threshold of 20 failed authentications. The attacker will start flooding with packets periodically and at a point the number of failed authentications will exceed 20 and the GW would deploy the puzzle. Let us assume that the attacker takes ten seconds to solve the puzzle, hence it will be able to send only three packets in the next time window so the GW would revoke the puzzle causing him to continue the attack in the next available time window. Instead of ten seconds if it takes close to 25 seconds for him to solve, it would be sending only one packet in that time window and the next puzzle would take more than 90% of the next time window saving two time windows. But instead if it took 30 seconds to solve the puzzle, it will be done in the particular time window and start attacking in the next window. So the time window and threshold play a huge role on how effective a puzzle difficulty could be. The work on deciding what the optimal time window and threshold remains future work.

Chapter 6: Conclusion and Future Work

In this work we have showed the design and implementation of DrHIP protocol and studied its performance in mitigating Denial of Service attacks. Our work focused on holding the attacker at on the periphery of the network to mitigate the impact of multi-hop attacks. To accomplish this we have utilized the puzzle mechanism that is built in the Host Identity Protocol. We have evaluated the performance of our protocol in Cooja simulator under different scenarios. The scenarios involved variation in the network topology, variation in the puzzle difficulty, and variation in the attacker's position. Based on the evaluation we have confirmed that puzzle utilization increases Goodput by more than 55% in 5 node and 15 node network and by 16% in the 10 node network. While there was not significant change in data throughput for the 5 node network, with the use of puzzles, 10 and 15 node network achieved an increase of 4% and 40% respectively. We also identified that the puzzle's difficulty cannot be fixed for all types of network, and the GW has to learn from the network activity on which difficulty would be appropriate. We also underscore the importance of choosing appropriate time window and threshold in deciding if there is an imminent threat. Having analyzed the effectiveness of puzzles in thwarting DOS attacks the next step is to come up with effective methods in choosing optimal puzzle difficulty, effective time window and optimal threshold. We believe these to be areas with potential opportunity and leave it for future work.

Appendix A: Appendix

```
/**
 * \file
 *         Code for GW
 */

#include "contiki.h"
#include "net/rime.h"
#include <relic_core.h>
#include "random.h"
#include "event-post.h"
#include "aes.h"
#include <string.h>
#include "cfs/cfs.h"
#include "cfs/cfs-coffee.h"
#include "dev/battery-sensor.h"
#include "dev/leds.h"
#include "lib/list.h"
#include "lib/memb.h"
#include "sys/timer.h"
#define SIZE_SEED                       32
#define DEBUG DEBUG_NONE
#define PROCESS_CONF_NO_PROCESS_NAMES   1
#define MAX_NEIGHBORS   12
```

```
#define INIT_PACKET      1
#define REPLY_PACKET     2
#define GK_PACKET        3
#define ROOT_PACKET      4
#define HELLO_PACKET     5
#define BEACON_PACKET    6
#define ACTIVE_PACKET    7
#define ACK_PACKET       8
#define PUZZLE_PACKET    10
#define PUZZ_PACKET      11
#define DATA_PACKET 12


/*---------------------------------------------------*/
PROCESS(example_unicast_process, "Example_unicast");
PROCESS(beacon_process, "Beacon");
PROCESS(gk_process, "Gk");
PROCESS(timeout_process, "TM");
PROCESS(revoke_process, "RV");
PROCESS(root_process,"root");
PROCESS(state_process,"state");
PROCESS(puzzle_process,"puzzle");
PROCESS(data_process,"data");
AUTOSTART_PROCESSES(&example_unicast_process, \
&beacon_process,&root_process,&puzzle_process,&data_process);
/*---------------------------------------------------*/
```

```
typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t dest;
uint8_t bin[2 * FC_BYTES + 1];
int puzzle;
int len;
}key_packet;

typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t gw;
int len;
int seqnum;
unsigned char padding[41];
}root_packet;

typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t dest;
unsigned char gk[16];
unsigned char padding[29];
}gk_packet;
```

```
typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t dest;
int seqnum;
unsigned char padding[43];
}notify_packet;

typedef struct{
unsigned char iv[16];
int iv_flag;
unsigned char enc_text[40];
}data_packet;

struct record
{
char key[16];
int counter;
};
/*Neighbor Structure*/
struct neighbor
{
struct neighbor* next;
rimeaddr_t addr;
```

```
rimeaddr_t next_hop;
uint8_t key[16];
char is_active;
int  data_pack;
};

  //Memeory Allocation
MEMB(neighbor_memb, struct neighbor, MAX_NEIGHBORS);
//List for holding the neighbors
LIST(neighbor_list);
static int key_len=16;
static struct unicast_conn uc,ug;
static struct broadcast_conn bc;
static aes_context ctx[1];
static aes_context ctx2[1];
static rimeaddr_t waiting_for;
static char msg[] = "data";
static char reply_received = 'y';
static char group_key_set='n';
static char holding_state ='n';
static char group_ack='n';
static char neighbor_removed ='n';
static char gk_proc='n';
static ec_t public_key;
static bn_t private_key;
```

```
static uint8_t* group_key;
static char state_process_start ='n';
static char puzzle_mode = 'n';
static uint8_t* oldgk;
static rimeaddr_t node_addr;
static rimeaddr_t next_hop;
static int seq_num = 345;
static int unauth_count=0;
static rimeaddr_t hold_addr;
static rimeaddr_t old_addr;
static rimeaddr_t puzzle_addr;
static rimeaddr_t puzzle_to;
static int pac_count;
static int data_count;



//Used to print the generated symmetric key
static void print_key(unsigned char *key, int key_len){
 int i;
 for(i=0; i<key_len; i++){
        printf("%02x", key[i]);
        }
    printf("\n");
}
```

```
//Add a new neighbor to the existing list
static void add_neighbor(unsigned char* key,const \
 rimeaddr_t* from, const rimeaddr_t* next)
{
struct neighbor *n;
for(n = list_head(neighbor_list); \
n != NULL; n = list_item_next(n)) {
    if(rimeaddr_cmp(&n->addr, from)) {
            return;
                }
        }
if(n==NULL)
{
        n = memb_alloc(&neighbor_memb);
     if(n == NULL) {
        return;
        }
        rimeaddr_copy(&n->addr,from);
     rimeaddr_copy(&n->next_hop,next);
     memcpy(&n->key,key,key_len);
     n->is_active ='y';
     n->data_pack =0;
        list_add(neighbor_list,n);
        }
```

```
}



//Remove existing node
static void remove_neighbor(const rimeaddr_t* from)
{
  struct neighbor *n;
for(n = list_head(neighbor_list); \
n != NULL; n = list_item_next(n)) {
     if(rimeaddr_cmp(&n->addr, from)) {
              list_remove(neighbor_list,n);
                  }
         }
}


//get next node
static struct neighbor* get_next_node \
(const rimeaddr_t* from)
{
struct neighbor *n;
if (from->u8[0]==0)
  {
   n=list_head(neighbor_list);
   return n;
         }
```

```
else
{
for(n = list_head(neighbor_list); \
n != NULL; n = list_item_next(n)) {
    if(rimeaddr_cmp(&n->addr, from)) {
        if (list_item_next(n) != NULL)
                {
                n=list_item_next(n);
                return n;
                }
                        }
                                }
                                        }
                return NULL;
                                        }

//Get the symmetric key for a node
static unsigned char* get_key(const rimeaddr_t* node)
{
  struct neighbor *n;
for(n = list_head(neighbor_list); \
n != NULL; n = list_item_next(n)) {
    if(rimeaddr_cmp(&n->addr, node)) {
            return n->key;
                }
```

```
        }
return NULL;
}
static unsigned char* fetch_rand()
{
    unsigned char iv[16];
    unsigned char key1[16];
    unsigned char* enc;
    int counter;
    int fd, c, ln,i;
    struct record new_record;
    SENSORS_ACTIVATE(battery_sensor);
    for (i=0;i<16;i++)
    {
    uint16_t bateria = battery_sensor.value(0);
     iv[i]=(char)(bateria * 2.500 * 2) / 4096;
        }
    fd = cfs_open("A", CFS_WRITE| CFS_READ);
     if (fd == -1) {
                exit(0);
        }
      if(cfs_seek(fd,0, CFS_SEEK_SET) != 0) {
                cfs_close(fd);
                return NULL;
                  }
```

```
ln=cfs_read(fd,&new_record,sizeof(new_record));
if(ln!=0)
{
 memcpy(key1,new_record.key,sizeof(key1));
 counter= new_record.counter;
}
else
{
random_init(rimeaddr_node_addr.u8[0]);
counter = 432+rimeaddr_node_addr.u8[0];
for(i=0;i<16;i++)
key1[i]=random_rand() & 0xFF;
  }
xor_block(&key1,&iv);
aes_set_key(key1,16,ctx2);
enc=(unsigned char*)malloc(16);
aes_encrypt((const unsigned char*)&counter, \
    enc,ctx2);
memcpy(new_record.key,enc,16);
counter++;
new_record.counter=counter;
if(cfs_seek(fd,0, CFS_SEEK_SET) != 0) {
            cfs_close(fd);
            return NULL;
              }
```

```
        c = cfs_write(fd, &new_record, \
            sizeof(new_record));
         if (c != sizeof(new_record)) {
                return NULL;
                }
         if (fd != -1) {
                cfs_close(fd);
         }
        enc[16]='\0';
       return enc;
}
//returns a padded input
static unsigned char* pad_return(unsigned char* input)
{
size_t j;
if(strlen((const char*)input)%16==0)
         return input;
else
{
unsigned char* temp1;
j=strlen((char*)input)+(int) \
(16-(strlen((char*)input)%16));
temp1=(unsigned char*)malloc(j);
memset(temp1,0,j);
memcpy(temp1,input,strlen((char*)input));
```

```
temp1[j]='\0';
return temp1;
        }
return input;
}

//Returns an encyrpted text
static unsigned char* enc_key(unsigned char* input,\
 uint8_t* key)
{
    int j;
    uint8_t data_key[16];
    uint8_t temp[16];
    const unsigned char enc_temp[16];
      int rand_neighbor;
    char* enc;
    for(j=0;j<16;j++)
    temp[j]=0;
    memcpy(data_key,key,key_len);
    aes_set_key(data_key,key_len,ctx2);
    enc=(unsigned char*)malloc(16);
    aes_encrypt(input,enc,ctx2);
    return enc;
}
```

```
//Broadcasts the beacon active
static void send_beacon()
{
        notify_packet new;
        new.type=BEACON_PACKET;
        seq_num=seq_num+3;
        new.seqnum=seq_num;
        memcpy(&new.source,&rimeaddr_node_addr \
        ,sizeof(new.source));
        memset(&new.dest,0,sizeof(new.dest));
        packetbuf_copyfrom(&new,sizeof(new));
        broadcast_send(&bc);
        process_start(&revoke_process,NULL);
}

//Sets a particular node to active
static void set_active(const rimeaddr_t* node)
{
struct neighbor *n;
for(n = list_head(neighbor_list); \
  n != NULL; n = list_item_next(n)) {
if(rimeaddr_cmp(&n->addr, node)) {
        n->is_active='y';
}
        }
```

```
}


//Generates the group key
static void groupkey_generate()
{
struct neighbor *n;
uint8_t buffer[(16*(list_length(neighbor_list)))+2];
uint8_t temp[MD_LEN];
int i;
int l=0;
for(n = list_head(neighbor_list); \
n != NULL; n = list_item_next(n)) {
                memcpy(buffer+l,n->key,sizeof(n->key));
          l=l+16;
          }
buffer[(16*(list_length(neighbor_list)))+1]= \
random_rand() & 0xFF;
buffer[(16*(list_length(neighbor_list)))+2]= \
random_rand() & 0xFF;
md_map(temp,buffer,sizeof(buffer));
if(group_key_set == 'y')
    free(group_key);
group_key=(uint8_t*)malloc(16);
memcpy(group_key,temp,key_len);
}
```

```
//Sends the reset puzzle packet
static void reset_puzzle()
{
notify_packet new_notify;
new_notify.type = PUZZLE_PACKET;
memcpy(&new_notify.dest,&puzzle_addr, \
sizeof(new_notify.dest));
memcpy(&new_notify.source,&rimeaddr_node_addr, \
sizeof(new_notify.source));
new_notify.seqnum=0;
packetbuf_copyfrom(&new_notify,sizeof(new_notify));
printf("sending reset to%d.%d\n",puzzle_to.u8[0], \
puzzle_to.u8[1]);
unicast_send(&uc,&puzzle_to);
}

//revokes an inactive node
static void revoke_access()
{
struct neighbor *n;
for(n = list_head(neighbor_list); \
n != NULL; n = list_item_next(n)) {
if(n->is_active == 'n')
 {
```

```
        remove_neighbor(&n->addr);
            neighbor_removed='y';
            }
    }
    if(neighbor_removed =='y')
    {
        groupkey_generate();
        node_addr.u8[0]=0;
        node_addr.u8[1]=0;
        struct neighbor* temp=get_next_node(&node_addr);
        node_addr.u8[0]=temp->addr.u8[0];
        node_addr.u8[1]=temp->addr.u8[1];
        next_hop.u8[0]=temp->next_hop.u8[0];
        next_hop.u8[1]=temp->next_hop.u8[1];

        gk_proc='y';
        process_post_synch(&gk_process, \
            PROCESS_EVENT_CONTINUE,&msg);
            }
    }

//callback for unicast channel
static void recv_uc(struct unicast_conn *c, \
 const rimeaddr_t *from)
{
```

```
pac_count++;
ec_t q1;
int rc=0;
int l=0;
uint8_t key[16];
uint8_t bin[2 * FC_BYTES + 1];
key_packet recv_packet;
memcpy(&recv_packet,packetbuf_dataptr(), \
sizeof(recv_packet));
if(recv_packet.type == ACTIVE_PACKET)
 {
       notify_packet new;
       memcpy(&new,&recv_packet,sizeof(new));
      set_active(&new.source);
       }
 else if(recv_packet.type == DATA_PACKET)
 {
       data_count++;
       notify_packet new;
       memcpy(&new,&recv_packet,sizeof(new));
      set_data_pack(&new.source);
       }
 else if((recv_packet.type == HELLO_PACKET))
{
key_packet new_packet;
```

```
l=ec_size_bin(public_key,1);
ec_write_bin(bin,l,public_key,1);
new_packet.len=l;
memcpy(&new_packet.bin,bin,sizeof(bin));
memcpy(&new_packet.dest,&recv_packet.source, \
sizeof(new_packet.source));
memcpy(&new_packet.source,&rimeaddr_node_addr, \
sizeof(new_packet.dest));
new_packet.type= INIT_PACKET;
 packetbuf_copyfrom(&new_packet,sizeof(new_packet));
 if(recv_packet.source.u8[0] == 19)
 { notify_packet new;
   memcpy(&new,&recv_packet,sizeof(new));
   unauth_count++;
   if(puzzle_mode == 'y')
    {
       notify_packet new_notify;
       new_notify.type = PUZZLE_PACKET;
       rimeaddr_copy(&puzzle_addr,&recv_packet.source);
       rimeaddr_copy(&puzzle_to,from);
       memcpy(&new_notify.dest,&recv_packet.source, \
       sizeof(new_notify.dest));
       memcpy(&new_notify.source,&rimeaddr_node_addr, \
       sizeof(new_notify.source));
       new_notify.seqnum=15;
```

```
        packetbuf_copyfrom(&new_notify,sizeof(new_notify));
              unicast_send(&uc,from);
                }
        else if(new.seqnum == 2)
        {
        notify_packet new_notify;
        new_notify.type = PUZZLE_PACKET;
        rimeaddr_copy(&puzzle_addr,&recv_packet.source);
        rimeaddr_copy(&puzzle_to,from);
        memcpy(&new_notify.dest,&recv_packet.source, \
        sizeof(new_notify.dest));
        memcpy(&new_notify.source,&rimeaddr_node_addr, \
        sizeof(new_notify.source));
        new_notify.seqnum=0;
        packetbuf_copyfrom(&new_notify,sizeof(new_notify));
            unicast_send(&uc,from);
                }
                        }
 else if(recv_packet.source.u8[0] == 24)
{
        }
 else if(holding_state == 'n')
 {
 unicast_send(&uc,from);
 holding_state ='y';
```

```
    rimeaddr_copy(&hold_addr,from);
    if(state_process_start == 'n')
        process_start(&state_process,NULL);
    else
        process_post_synch(&state_process, \
PROCESS_EVENT_CONTINUE,&msg);
        }
                        }
  else
  {
   ec_read_bin(q1,recv_packet.bin,recv_packet.len);
    rc=cp_ecdh_key(key, key_len, private_key, q1);
   if(recv_packet.type == INIT_PACKET)
    {
    key_packet send_packet;
    l=ec_size_bin(public_key,1);
    ec_write_bin(bin,l,public_key,1);
    memcpy(&send_packet.bin,bin,sizeof(bin));
    send_packet.len=l;
    send_packet.type=REPLY_PACKET;
    memcpy(&send_packet.source,&rimeaddr_node_addr,\
        sizeof(send_packet.source));
    memcpy(&send_packet.dest,&recv_packet.source, \
        sizeof(send_packet.dest));
    packetbuf_copyfrom(&send_packet,sizeof(send_packet));
```

```
    unicast_send(&uc,from);
        }
    add_neighbor(key,&recv_packet.source,from);
    groupkey_generate();
    node_addr.u8[0]=0;
    node_addr.u8[1]=0;
    struct neighbor* temp=get_next_node(&node_addr);
    node_addr.u8[0]=temp->addr.u8[0];
    node_addr.u8[1]=temp->addr.u8[1];
    next_hop.u8[0]=temp->next_hop.u8[0];
    next_hop.u8[1]=temp->next_hop.u8[1];
    gk_proc='y';
    if (group_key_set =='y')
        process_post_synch(&gk_process, \
        PROCESS_EVENT_CONTINUE,&msg);
   else
        process_start(&gk_process,NULL);
                        }
                                }

static void recv_ug(struct unicast_conn *c, \
const rimeaddr_t *from)
{
  pac_count++;
   notify_packet new;
```

```
  memcpy(&new, packetbuf_dataptr(), sizeof(new));

  if((new.type == ACK_PACKET)&& \
  (rimeaddr_cmp(&node_addr,&new.source))) {
      group_ack='y';
      struct neighbor* temp=get_next_node(&node_addr);

  if (temp == NULL)
          holding_state ='n';
  else
  {
     node_addr.u8[0]=temp->addr.u8[0];
     node_addr.u8[1]=temp->addr.u8[1];
     next_hop.u8[0]=temp->next_hop.u8[0];
     next_hop.u8[1]=temp->next_hop.u8[1];
     gk_proc='y';
     process_post_synch(&gk_process,\
          PROCESS_EVENT_CONTINUE,&msg);
          }
                  }
                          }



static const struct unicast_callbacks \
```

```
 unicast_callbacks = {recv_uc};
static const struct unicast_callbacks \
 unicast_call_backs = {recv_ug};

//callback for broadcast channel
static void broadcast_recv(struct broadcast_conn *c, \
 const rimeaddr_t *from)
{
  pac_count++;
  key_packet new_packet;
  notify_packet new;
  memcpy(&new, packetbuf_dataptr(), sizeof(new));
  uint8_t bin[2 * FC_BYTES + 1];
  int l=0;
  l=ec_size_bin(public_key,1);
  ec_write_bin(bin,l,public_key,1);
  new_packet.len=l;
  memcpy(&new_packet.bin,bin,sizeof(bin));
  memcpy(&new_packet.dest,&new.source, \
  sizeof(new_packet.dest));
  memcpy(&new_packet.source,&rimeaddr_node_addr, \
  sizeof(new_packet.source));
  new_packet.type= INIT_PACKET;
  packetbuf_copyfrom(&new_packet,sizeof(new_packet));
if(new.type == DATA_PACKET)
```

```
data_count++;
else if((holding_state == 'n') && (new.type \
== HELLO_PACKET))
 { if(new.source.u8[0] == 14)
   {
     unauth_count++;
     if(puzzle_mode == 'y')
      {
        notify_packet new_notify;
        new_notify.type = PUZZLE_PACKET;
        memcpy(&new_notify.dest,&new.source, \
        sizeof(new_notify.dest));
        memcpy(&new_notify.source,&rimeaddr_node_addr, \
        sizeof(new_notify.source));
        rimeaddr_copy(&puzzle_addr,&new.source);
        new_notify.seqnum=15;
        packetbuf_copyfrom(&new_notify,sizeof(new_notify));
        unicast_send(&uc,from);
                }
        }
    else if(new.source.u8[0] == 2)
   {
    }
    else
    {
```

```
    unicast_send(&uc,from);
    holding_state ='y';
    rimeaddr_copy(&hold_addr,from);
    if(state_process_start == 'n')
        process_start(&state_process,NULL);
    else
        process_post_synch(&state_process, \
PROCESS_EVENT_CONTINUE,&msg);
        }
}
}

static const struct broadcast_callbacks \
broadcast_callbacks = {broadcast_recv};

//initialisation process
PROCESS_THREAD(example_unicast_process, ev, data)
{
    PROCESS_BEGIN();
    unicast_open(&uc, 146, &unicast_callbacks);
    broadcast_open(&bc,137,&broadcast_callbacks);
    static struct etimer et;
    uint8_t buf[32];
    int rand;
    int ln1=0;
```

```
    watchdog_stop();
    core_init();
    root_packet new;
    key_packet old;
    gk_packet old1;
    notify_packet old2;
   for(rand=0;rand<2;rand++)
   {
    unsigned char* enc=fetch_rand();
    memcpy(buf+ln1,enc,16);
    ln1=ln1+16;
        }
   rand_seed(buf, SIZE_SEED);
   ep_param_set(SECG_P160);
   bn_new(private_key);
   ec_new(public_key);
   cp_ecdh_gen(private_key,public_key);
   ep_print(public_key);
   etimer_set(&et, (CLOCK_SECOND*180));
   PROCESS_END();
}
//Process for Root advertisement
PROCESS_THREAD(root_process, ev, data)
{
    PROCESS_BEGIN();
```

```
 static struct etimer est,et;
 etimer_set(&est, (CLOCK_SECOND*30));
while(1)
{
 PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
 etimer_reset(&est);
 if(holding_state == 'n')
 {
 holding_state='y';
 root_packet send_packet;
 memcpy(&send_packet.source,&rimeaddr_node_addr, \
      sizeof(send_packet.source));
 memcpy(&send_packet.gw,&rimeaddr_node_addr, \
      sizeof(send_packet.gw));
 send_packet.len=0;
 send_packet.type=ROOT_PACKET;
 seq_num=seq_num+3;
 send_packet.seqnum=seq_num;
 packetbuf_copyfrom(&send_packet, \
      sizeof(send_packet));
 broadcast_send(&bc);
 holding_state='n';
      }
      }
      PROCESS_END();
```

```
}
//Process for sendinf beacon packets
PROCESS_THREAD(beacon_process, ev, data)
{
    PROCESS_BEGIN();
    static struct etimer est,et;
    etimer_set(&est, (CLOCK_SECOND*120));
   while(1)
   {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
    etimer_reset(&est);
    if((list_length(neighbor_list)!=0) && \
        (holding_state == 'n'))
    {
     holding_state='y';
    struct neighbor *n;
    for(n = list_head(neighbor_list); n != NULL; \
        n = list_item_next(n)) {
        n->is_active='n';
        }
    send_beacon();
        }
     }
    PROCESS_END();
```

```
}

//Process for sending group key
PROCESS_THREAD(gk_process, ev, data)
{
  //PROCESS_EXITHANDLER()
    PROCESS_BEGIN();
    group_key_set ='y';
    unicast_open(&ug, 148, &unicast_call_backs);
    process_start(&timeout_process,NULL);
   while(1)
   {
    gk_packet bgk_packet;
    char* enc_bgk=enc_key(group_key,get_key(&node_addr));
    memcpy(&bgk_packet.gk,enc_bgk,sizeof(bgk_packet.gk));
    bgk_packet.type=GK_PACKET;
    memcpy(&bgk_packet.source,&rimeaddr_node_addr, \
        sizeof(bgk_packet.source));
    memcpy(&bgk_packet.dest,&node_addr, \
        sizeof(bgk_packet.dest));
    packetbuf_copyfrom(&bgk_packet,sizeof(bgk_packet));
    if(gk_proc == 'y')
    {
    if(!rimeaddr_cmp(&node_addr, &next_hop))
    unicast_send(&ug,&next_hop);
```

```
      else
      unicast_send(&ug,&node_addr);
      gk_proc='n';
      group_ack='n';
       }
          PROCESS_WAIT_EVENT();
      }
      PROCESS_END();
}
//Process for periodic timeouts
PROCESS_THREAD(timeout_process, ev, data)
{
      PROCESS_BEGIN();
      static struct etimer est;
     etimer_set(&est, (CLOCK_SECOND*30));
     while(1)
     {
       PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
       etimer_reset(&est);
       if(group_ack =='n')
        {   gk_proc='y';
            process_post_synch(&gk_process,\
                      PROCESS_EVENT_CONTINUE,&msg);
                    }
      }
```

```
    PROCESS_END();
}
//Process for revoking nodes
PROCESS_THREAD(revoke_process, ev, data)
{
    PROCESS_BEGIN();
    static struct etimer est,et;
    etimer_set(&est, (CLOCK_SECOND*40));
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
    neighbor_removed='n';
    revoke_access();
    holding_state='n';
    PROCESS_END();
}

PROCESS_THREAD(state_process, ev, data)
{
  PROCESS_BEGIN();
  static struct etimer est;
  etimer_set(&est, (CLOCK_SECOND*30));
  state_process_start = 'y';
  while(1)
  {
    rimeaddr_copy(&old_addr,&hold_addr);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
```

```
    etimer_reset(&est);
    if(rimeaddr_cmp(&old_addr,&hold_addr))
    {
      if(holding_state == 'y')
      {
      holding_state = 'n';
        }
        }
    PROCESS_WAIT_EVENT();
    }
    PROCESS_END();
}



PROCESS_THREAD(puzzle_process, ev, data)
{
  PROCESS_BEGIN();
  static struct etimer est;
  etimer_set(&est, (CLOCK_SECOND*60));
   while(1)
  {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
    etimer_reset(&est);
   printf("current_unauth_count:%d\n",unauth_count);
    if(unauth_count > 5)
```

```
        {
          puzzle_mode = 'y';
          unauth_count =0;
          }
       else
        {
          if(puzzle_mode == 'y')
              reset_puzzle();
          puzzle_mode = 'n';
          unauth_count =0;

          }
  }
     PROCESS_END();
}

PROCESS_THREAD(data_process, ev, data)
{
   PROCESS_BEGIN();
   static struct etimer est;
   etimer_set(&est, (CLOCK_SECOND*60));
    while(1)
   {
     PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
     etimer_reset(&est);
```

```
    }
    PROCESS_END();
}

/*---------------------------------------------------------------------------*/


/**
 * \file
 *         Code for Sensor node DrHIP Protocol
 *
 */

#include "contiki.h"
#include "net/rime.h"
#include <relic_core.h>
#include "random.h"
#include "event-post.h"
#include "aes.h"
#include <string.h>
#include "cfs/cfs.h"
#include "cfs/cfs-coffee.h"
#include "dev/battery-sensor.h"
#include "dev/leds.h"
#include "lib/list.h"
```

```
#include "lib/memb.h"
#include "sys/timer.h"
#define SIZE_SEED                    32
#define DEBUG DEBUG_NONE
#define PROCESS_CONF_NO_PROCESS_NAMES    1
#define MAX_NEIGHBORS    8
#define INIT_PACKET      1
#define REPLY_PACKET     2
#define GK_PACKET        3
#define ROOT_PACKET      4
#define HELLO_PACKET     5
#define BEACON_PACKET    6
#define ACTIVE_PACKET    7
#define ACK_PACKET       8
#define NEIGHBOR_PACKET  9
#define PUZZLE_PACKET    10
#define PUZZ_PACKET      11
#define DATA_PACKET      12
/*---------------------------------------------*/
PROCESS(example_unicast_process, "Example unicast");
PROCESS(dos_process, "DOS Process");
AUTOSTART_PROCESSES(&example_unicast_process);
/*---------------------------------------------*/
typedef struct{
int type;
```

```
rimeaddr_t source;
rimeaddr_t dest;
uint8_t bin[2 * FC_BYTES + 1];
int puzzle;
int len;
}key_packet;

typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t dest;
unsigned char gk[16];
unsigned char padding[29];
}gk_packet;

typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t dest;
int seqnum;
unsigned char padding[43];
}notify_packet;



typedef struct{
```

```
int type;
rimeaddr_t source;
rimeaddr_t gw;
int len;
int seqnum;
unsigned char padding[41];
}root_packet;

typedef struct{
unsigned char iv[16];
int iv_flag;
unsigned char enc_text[40];
}data_packet;

struct record
{
char key[16];
int counter;
};
/*Neighbor Structure*/
struct neighbor
{
struct neighbor* next;
rimeaddr_t addr;
unsigned char key[16];
```

```
};

struct dest
{
struct dest* next;
rimeaddr_t send_to;
rimeaddr_t to;
};

//Memeory Allocation
MEMB(neighbor_memb, struct neighbor, MAX_NEIGHBORS);
MEMB(dest_memb, struct dest, MAX_NEIGHBORS);
//List for holding the neighbors
LIST(neighbor_list);
LIST(dest_list);
static int key_len=16;
static struct unicast_conn uc,ud,ug;
static struct broadcast_conn bc;
static aes_context ctx[1];
static aes_context ctx2[1];
static rimeaddr_t gw;
static rimeaddr_t gw_default;
static char msg[] = "data";
static int node_addr=0;
static char reply_received = 'n';
```

```
static char gk_received='n';
static char gw_set ='n';
static char puzzle_mode ='n';
static char dos_started ='n';
static int curr_path;
static int recv_puzzle;
static ec_t public_key;
static bn_t private_key;
static uint8_t* group_key;
static int seq_num=0;
static int recv_conn=0;
//Used to print the generated symmetric key
static void print_key(unsigned char *key, \
 int key_len){
        int i;
        for(i=0; i<key_len; i++){
                printf("%02x", key[i]);
        }
        printf("\n");
}

//Add a new neighbor to the existing list
static void add_neighbor(uint8_t* key, \
const rimeaddr_t* from)
{
```

```
  struct neighbor *n;
for(n = list_head(neighbor_list); n != NULL; \
n = list_item_next(n)) {
     if(rimeaddr_cmp(&n->addr, from)) {
               return;
                    }
          }
    if(n==NULL)
     {
          n = memb_alloc(&neighbor_memb);
          if(n == NULL) {
          return;
                    }
          rimeaddr_copy(&n->addr,from);
          memcpy(&n->key,key,key_len);
          list_add(neighbor_list,n);
          }

}

//Add a new neighbor to the existing list
static void add_dest(const rimeaddr_t* to, \
const rimeaddr_t* send_to)
{
  struct dest *n;
```

```
for(n = list_head(dest_list); n != NULL; \
n = list_item_next(n)) {
     if(rimeaddr_cmp(&n->to, to)) {
               return;
                    }
          }
    if(n==NULL)
     {
          n = memb_alloc(&dest_memb);
          if(n == NULL) {
          return;
                    }
          rimeaddr_copy(&n->to,to);
          rimeaddr_copy(&n->send_to,send_to);
          list_add(dest_list,n);
          }

}


//Get send_to for a node
static rimeaddr_t* get_send_to(const rimeaddr_t* node)
{
  struct dest *n;
for(n = list_head(dest_list); n != NULL; \
```

```
n = list_item_next(n)) {
    if(rimeaddr_cmp(&n->to, node)) {
                return &n->send_to;
                }
        }
return NULL;
}

//Get the symmetric key for a node
static unsigned char* get_key(const rimeaddr_t* node)
{
  struct neighbor *n;
for(n = list_head(neighbor_list); n != NULL; \
n = list_item_next(n)) {
    if(rimeaddr_cmp(&n->addr, node)) {
            return n->key;
                }
        }
return NULL;
}
//Call back for unicast channel
static void recv_uc(struct unicast_conn *c, \
const rimeaddr_t *from)
{
  ec_t q1;
```

```
  int rc=0;
  int l=0;
  uint8_t key[key_len];
  uint8_t bin[2 * FC_BYTES + 1];
  key_packet recv_packet;
  rimeaddr_t addr;
memcpy(&recv_packet,packetbuf_dataptr(),\
sizeof(recv_packet));
addr.u8[0]=recv_packet.dest.u8[0];
addr.u8[1]=recv_packet.dest.u8[1];
if(recv_packet.type == HELLO_PACKET)
{
if(puzzle_mode != 'y')
{
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
add_dest(&recv_packet.source,from);
unicast_send(&uc,&gw);
}
else
{
notify_packet new;
memcpy(&new,&recv_packet,sizeof(new));
if(new.seqnum == 2)
{
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
```

```
add_dest(&recv_packet.source,from);
unicast_send(&uc,&gw);
}
else
{
new.type = PUZZ_PACKET;
new.seqnum = recv_puzzle;
memcpy(&new.source,&rimeaddr_node_addr, \
sizeof(new.source));
memcpy(&new.dest,from,sizeof(new.dest));
packetbuf_copyfrom(&new,sizeof(new));
unicast_send(&uc,from);
        }
                }
}
else if(recv_packet.type == ACTIVE_PACKET)
{
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
add_dest(&recv_packet.source,from);
unicast_send(&uc,&gw);
}
else if(recv_packet.type == PUZZLE_PACKET)
{
notify_packet new;
memcpy(&new,&recv_packet,sizeof(new));
```

```
rimeaddr_t* to_addr;
to_addr=get_send_to(&new.dest);
if(rimeaddr_cmp(&addr,to_addr))
{
if(new.seqnum > 0)
{
puzzle_mode ='y';
recv_puzzle = new.seqnum;
}
else
{
puzzle_mode = 'n';
        }
}
else
{
unicast_send(&uc,get_send_to(&new.dest));
        }
}
else if(recv_packet.type == PUZZ_PACKET)
{
if(dos_started == 'n')
        process_start(&dos_process,NULL);
else
process_post_synch(&dos_process, \
```

```
PROCESS_EVENT_CONTINUE,&msg);
}
else if(!rimeaddr_cmp(&addr, &rimeaddr_node_addr)) {
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
unicast_send(&uc,get_send_to(&recv_packet.dest));
}
else
 {
    reply_received ='y';
  ec_read_bin(q1,recv_packet.bin,recv_packet.len);
  rc=cp_ecdh_key(key, key_len, private_key, q1);
  if(recv_packet.type == INIT_PACKET)
  {
    key_packet send_packet;
    l=ec_size_bin(public_key,1);
    ec_write_bin(bin,l,public_key,1);
    memcpy(&send_packet.bin,bin,sizeof(bin));
    send_packet.len=l;
    send_packet.type=REPLY_PACKET;
    memcpy(&send_packet.source,&rimeaddr_node_addr, \
        sizeof(send_packet.source));
    memcpy(&send_packet.dest,&recv_packet.source, \
        sizeof(send_packet.dest));
    packetbuf_copyfrom(&send_packet,sizeof(send_packet));
    unicast_send(&uc,from);
```

```
        }
      if(rc ==STS_OK)
      {
        add_neighbor(key,from);
        add_dest(&recv_packet.source,from);
        }
        }

}

//Used for padding the provided input
static unsigned char* pad_return(unsigned char* input)
{
size_t j;
if(strlen((const char*)input)%16==0)
        return input;
else
{
unsigned char* temp1;
j=strlen((char*)input)+(int) \
(16-(strlen((char*)input)%16));
temp1=(unsigned char*)malloc(j);
memset(temp1,0,j);
memcpy(temp1,input,strlen((char*)input));
temp1[j]='\0';
```

```
return temp1;
        }
return input;
}
//Random Byte generator
static unsigned char* fetch_rand()
{
unsigned char iv[16];
    unsigned char key1[16];
    unsigned char* enc;
    int counter;
    int fd, c, ln,i;
    struct record new_record;
    SENSORS_ACTIVATE(battery_sensor);
    for (i=0;i<16;i++)
    {
    uint16_t bateria = battery_sensor.value(0);
     iv[i]=(char)(bateria * 2.500 * 2) / 4096;
        }
    fd = cfs_open("A", CFS_WRITE| CFS_READ);
     if (fd == -1) {
                exit(0);
        }
      if(cfs_seek(fd,0, CFS_SEEK_SET) != 0) {
                cfs_close(fd);
```

```
            return NULL;
              }
ln=cfs_read(fd,&new_record,sizeof(new_record));
if(ln!=0)
{
 memcpy(key1,new_record.key,sizeof(key1));
 counter= new_record.counter;
}
else
{
random_init(rimeaddr_node_addr.u8[0]);
counter = 432+rimeaddr_node_addr.u8[0];
for(i=0;i<16;i++)
key1[i]=random_rand() & 0xFF;
  }
xor_block(&key1,&iv);
aes_set_key(key1,16,ctx2);
enc=(unsigned char*)malloc(16);
if(enc==NULL)
     printf("I am failing here\n");
aes_encrypt((const unsigned char*)&counter,enc,ctx2);
memcpy(new_record.key,enc,16);
counter++;
new_record.counter=counter;
if(cfs_seek(fd,0, CFS_SEEK_SET) != 0) {
```

```
                cfs_close(fd);
                return NULL;
                  }
        c = cfs_write(fd, &new_record, sizeof(new_record));
         if (c != sizeof(new_record)) {
                return NULL;
                }
         if (fd != -1) {
                cfs_close(fd);
         }
        enc[16]='\0';
       return enc;

}
//Callback for broadcast channel
static void broadcast_recv(struct broadcast_conn *c, \
const rimeaddr_t *from)
{
key_packet recv_packet;
 memcpy(&recv_packet,packetbuf_dataptr(),sizeof(recv_packet));
if(recv_packet.type == HELLO_PACKET)
{
if(puzzle_mode != 'y')
{
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
```

```
add_dest(&recv_packet.source,from);
if (gw_set == 'y')
unicast_send(&uc,&gw);
}
else
{
notify_packet new;
memcpy(&new,&recv_packet,sizeof(new));
if(new.seqnum == 2)
{
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
add_dest(&recv_packet.source,from);
unicast_send(&uc,&gw);
}
else
{
new.type = PUZZ_PACKET;
new.seqnum = recv_puzzle;
memcpy(&new.source,&rimeaddr_node_addr, \
sizeof(new.source));
memcpy(&new.dest,from,sizeof(new.dest));
packetbuf_copyfrom(&new,sizeof(new));
unicast_send(&uc,from);
        }
                }
```

```
}
else if(recv_packet.type == BEACON_PACKET)
{
    notify_packet new,old;
    memcpy(&old,&recv_packet,sizeof(old));
if(old.seqnum > seq_num+1)
  {
            memcpy(&old.source,&rimeaddr_node_addr, \
                sizeof(new.source));
            seq_num=old.seqnum+1;
            old.seqnum=seq_num;
            packetbuf_copyfrom(&old,sizeof(old));
            broadcast_send(&bc);
            new.type=ACTIVE_PACKET;
            memcpy(&new.source,&rimeaddr_node_addr, \
                        sizeof(new.source));
            memset(&new.dest,&gw,sizeof(new.dest));
            packetbuf_copyfrom(&new,sizeof(new));
            unicast_send(&uc, &gw);
        }
                }
else if((recv_packet.type== ROOT_PACKET) && \
(gk_received =='y'))
{
root_packet new;
```

```
memcpy(&new,&recv_packet,sizeof(new));
char set_gk = 'n';
if(gw_set == 'n')
{
  gw.u8[0]=from->u8[0];
  gw.u8[1]=from->u8[1];
  rimeaddr_copy(&gw_default,&new.gw);
  curr_path=new.len;
  gw_set ='y';
  set_gk = 'y';
  seq_num=new.seqnum+1;
}
else if(new.seqnum > seq_num+1)
{
  gw.u8[0]=from->u8[0];
  gw.u8[1]=from->u8[1];
  rimeaddr_copy(&gw_default,&new.gw);
  curr_path=new.len;
  gw_set ='y';
  set_gk = 'y';
  seq_num=new.seqnum+1;
}
    root_packet send_packet;
    send_packet.len=curr_path+1;
    send_packet.type=ROOT_PACKET;
```

```
    send_packet.seqnum=seq_num;
    memcpy(&send_packet.source,&rimeaddr_node_addr,\
        sizeof(send_packet.source));
    memcpy(&send_packet.gw,&gw_default,\
        sizeof(send_packet.gw));
    packetbuf_copyfrom(&send_packet,sizeof(send_packet));
    if(set_gk == 'y')
     broadcast_send(&bc);


                }
}
//Callback for gk unicast channel
static void recv_ug(struct unicast_conn *c, \
const rimeaddr_t *from)
{
gk_packet recv_packet;
memcpy(&recv_packet,packetbuf_dataptr(), \
sizeof(recv_packet));
if(!rimeaddr_cmp(&recv_packet.dest,\
&rimeaddr_node_addr)) {
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
unicast_send(&ug,get_send_to(&recv_packet.dest));
}
else if (recv_packet.type == GK_PACKET)
{
```

```
aes_context ctx3[1];
unsigned char* temp_key;
unsigned char data_key[16];
rimeaddr_t adr;
temp_key=get_key(from);
memcpy(&data_key,temp_key,sizeof(data_key));
aes_set_key(data_key,key_len,ctx3);
if(gk_received == 'y')
    free(group_key);
group_key=(uint8_t*)malloc(16);
if(aes_decrypt((unsigned char*)recv_packet.gk,\
group_key,ctx3) == EXIT_SUCCESS)
      {
      gk_received='y';
      notify_packet new;
      new.type=ACK_PACKET;
      memcpy(&new.source,&rimeaddr_node_addr, \
           sizeof(new.source));
      memcpy(&new.dest,&recv_packet.source, \
           sizeof(new.dest));
      packetbuf_copyfrom(&new,sizeof(new));
      unicast_send(&ug,from);
         }
else
       printf("failed to decrypt\n");
```

```
}
}

static const struct broadcast_callbacks \
broadcast_callbacks = {broadcast_recv};
static const struct unicast_callbacks \
unicast_callbacks = {recv_uc};
static const struct unicast_callbacks \
unicast_call = {recv_ug};

//Process for joining the network
PROCESS_THREAD(example_unicast_process, ev, data)
{
    PROCESS_BEGIN();
    broadcast_open(&bc,137,&broadcast_callbacks);
    unicast_open(&uc, 146, &unicast_callbacks);
    unicast_open(&ug, 148, &unicast_call);
    rimeaddr_t addr;
    static struct etimer et;
    uint8_t bin[2 * FC_BYTES + 1];
    uint8_t buf[32];
    int rand,rand2;
    int ln1=0,j=0;
    int l=0;
    node_addr=(int)rimeaddr_node_addr.u8[0];
```

```
  etimer_set(&et, CLOCK_SECOND);
  PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
  watchdog_stop();
  core_init();
  for(rand=0;rand<2;rand++)
  {
  unsigned char* enc=fetch_rand();
  memcpy(buf+ln1,enc,16);
  ln1=ln1+16;
   free(enc);
      }
 rand_seed(buf, SIZE_SEED);
 ep_param_set(SECG_P160);
 bn_new(private_key);
 ec_new(public_key);
 cp_ecdh_gen(private_key,public_key);
 while(1)
 {
 rand2=(unsigned int)random_rand()%4;
 if (rand2 ==0)
      rand2=1;
if(gk_received == 'y')
      etimer_set(&et, CLOCK_SECOND*5);
else if (reply_received =='y')
      etimer_set(&et, CLOCK_SECOND*90);
```

```
    else
          etimer_set(&et, CLOCK_SECOND*(25*rand2));

   if ( (reply_received != 'y') || (gk_received != 'y'))
    {     notify_packet new;
          new.type=HELLO_PACKET;
          memcpy(&new.source,&rimeaddr_node_addr, \
          sizeof(new.source));
          memset(&new.dest,0,sizeof(new.dest));
          packetbuf_copyfrom(&new,sizeof(new));
          broadcast_send(&bc);
          }
    else if((gk_received == 'y')&&(gw_set == 'y'))
          {
          notify_packet new;
          new.type=DATA_PACKET;
          memcpy(&new.source,&rimeaddr_node_addr,\
          sizeof(new.source));
          memset(&new.dest,&gw,sizeof(new.dest));
          packetbuf_copyfrom(&new,sizeof(new));
          unicast_send(&uc,&gw);
          }
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

}
```

```
        PROCESS_END();
}
//Process for Solving the puzzle
PROCESS_THREAD(dos_process, ev, data)
{
    PROCESS_BEGIN();
    dos_started = 'y';
    static struct etimer est,et;
    while(1)
    {
    etimer_set(&est, (CLOCK_SECOND*recv_puzzle));
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
    PROCESS_WAIT_EVENT();
    }
    PROCESS_END();

}
/*-----------------------------------------------------------*/



/**
 * \file
 *  Code for attacker
*/
#include "contiki.h"
```

```
#include "net/rime.h"
#include <relic_core.h>
#include "random.h"
#include "event-post.h"
#include "aes.h"
#include <string.h>
#include "cfs/cfs.h"
#include "cfs/cfs-coffee.h"
#include "dev/battery-sensor.h"
#include "dev/leds.h"
#include "lib/list.h"
#include "lib/memb.h"
#include "sys/timer.h"
#define SIZE_SEED                       32
#define DEBUG DEBUG_NONE
#define PROCESS_CONF_NO_PROCESS_NAMES   1
#define MAX_NEIGHBORS   8
#define INIT_PACKET     1
#define REPLY_PACKET    2
#define GK_PACKET       3
#define ROOT_PACKET     4
#define HELLO_PACKET    5
#define BEACON_PACKET   6
#define ACTIVE_PACKET   7
#define ACK_PACKET      8
```

```
#define NEIGHBOR_PACKET 9
#define PUZZLE_PACKET    10
#define PUZZ_PACKET      11
/*---------------------------------------------------*/
PROCESS(example_unicast_process, "Example unicast");
PROCESS(dos_process, "DOS Process");
AUTOSTART_PROCESSES(&example_unicast_process);
/*---------------------------------------------------*/
typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t dest;
uint8_t bin[2 * FC_BYTES + 1];
int puzzle;
int len;
}key_packet;

typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t dest;
unsigned char gk[16];
unsigned char padding[29];
}gk_packet;
```

```
typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t dest;
int seqnum;
unsigned char padding[43];
}notify_packet;


typedef struct{
int type;
rimeaddr_t source;
rimeaddr_t gw;
int len;
int seqnum;
unsigned char padding[41];
}root_packet;

typedef struct{
unsigned char iv[16];
int iv_flag;
unsigned char enc_text[40];
}data_packet;

struct record
```

```
{
char key[16];
int counter;
};
/*Neighbor Structure*/
struct neighbor
{
struct neighbor* next;
rimeaddr_t addr;
unsigned char key[16];
};

struct dest
{
struct dest* next;
rimeaddr_t send_to;
rimeaddr_t to;
};

//Memeory Allocation
MEMB(neighbor_memb, struct neighbor, MAX_NEIGHBORS);
MEMB(dest_memb, struct dest, MAX_NEIGHBORS);
LIST(neighbor_list);
LIST(dest_list);
static int key_len=16;
```

```
static struct unicast_conn uc,ud,ug;
static struct broadcast_conn bc;
static aes_context ctx[1];
static aes_context ctx2[1];
static rimeaddr_t gw;
static rimeaddr_t gw_default;
static char msg[] = "data";
static int node_addr=0;
static char reply_received = 'n';
static char gk_received='n';
static char gw_set ='n';
static char puzzle_mode ='n';
static char dos_started ='n';
static char puzz_received ='n';
static int curr_path;
static int recv_puzzle;
static ec_t public_key;
static bn_t private_key;
static uint8_t* group_key;
static int seq_num=0;
static int recv_conn=0;

//used to receive unicast callbacks
static void recv_uc(struct unicast_conn *c, \
const rimeaddr_t *from)
```

```
{
  ec_t q1;
  int rc=0;
  int l=0;
  uint8_t key[key_len];
  uint8_t bin[2 * FC_BYTES + 1];
  key_packet recv_packet;
  rimeaddr_t addr;
memcpy(&recv_packet,packetbuf_dataptr(),sizeof(recv_packet));
addr.u8[0]=recv_packet.dest.u8[0];
addr.u8[1]=recv_packet.dest.u8[1];
if(recv_packet.type == HELLO_PACKET)
{
if(puzzle_mode != 'y')
{
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
add_dest(&recv_packet.source,from);
unicast_send(&uc,&gw);
}
else
{
notify_packet new;
memcpy(&new,&recv_packet,sizeof(new));
if(new.seqnum == 2)
{
```

```
packetbuf_copyfrom(&recv_packet, sizeof(recv_packet));
add_dest(&recv_packet.source, from);
unicast_send(&uc,&gw);
}
else
{
new.type = PUZZ_PACKET;
new.seqnum = recv_puzzle;
memcpy(&new.source,&rimeaddr_node_addr, sizeof(new.source));
memcpy(&new.dest, from, sizeof(new.dest));
packetbuf_copyfrom(&new, sizeof(new));
unicast_send(&uc, from);
        }
                }
}
else if(recv_packet.type == ACTIVE_PACKET)
{
packetbuf_copyfrom(&recv_packet, sizeof(recv_packet));
add_dest(&recv_packet.source, from);
unicast_send(&uc,&gw);
}
else if(recv_packet.type == PUZZLE_PACKET)
{
notify_packet new;
memcpy(&new,&recv_packet, sizeof(new));
```

```
rimeaddr_t* to_addr;
to_addr=get_send_to(&new.dest);
if(rimeaddr_cmp(&addr,to_addr))
{
puzzle_mode ='y';
recv_puzzle = new.seqnum;
}
else
unicast_send(&uc,get_send_to(&new.dest));
}
else if(recv_packet.type == PUZZ_PACKET)
{
puzz_received='y';
notify_packet new;
memcpy(&new,&recv_packet,sizeof(new));
recv_puzzle = new.seqnum;
printf("starting timer process\n");
process_start(&dos_process,NULL);
 }
else if(!rimeaddr_cmp(&addr, &rimeaddr_node_addr)) {
packetbuf_copyfrom(&recv_packet,sizeof(recv_packet));
unicast_send(&uc,get_send_to(&recv_packet.dest));
}
```

```
}


//random bytes generator
static unsigned char* fetch_rand()
{
unsigned char iv[16];
    unsigned char key1[16];
    unsigned char* enc;
    int counter;
    int fd, c, ln,i;
    struct record new_record;
    SENSORS_ACTIVATE(battery_sensor);
    for (i=0;i<16;i++)
    {
    uint16_t bateria = battery_sensor.value(0);
     iv[i]=(char)(bateria * 2.500 * 2) / 4096;
        }
    fd = cfs_open("A", CFS_WRITE| CFS_READ);
     if (fd == -1) {
                exit(0);
        }
      if(cfs_seek(fd,0, CFS_SEEK_SET) != 0) {
                cfs_close(fd);
                return NULL;
                  }
```

```
ln=cfs_read(fd,&new_record,sizeof(new_record));
if(ln!=0)
{
 memcpy(key1,new_record.key,sizeof(key1));
 counter= new_record.counter;
}
else
{
random_init(rimeaddr_node_addr.u8[0]);
counter = 432+rimeaddr_node_addr.u8[0];
for(i=0;i<16;i++)
key1[i]=random_rand() & 0xFF;
  }
xor_block(&key1,&iv);
aes_set_key(key1,16,ctx2);
enc=(unsigned char*)malloc(16);
if(enc==NULL)
            printf("I am failing here\n");
aes_encrypt((const unsigned char*)&counter,enc,ctx2);
memcpy(new_record.key,enc,16);
counter++;
new_record.counter=counter;
if(cfs_seek(fd,0, CFS_SEEK_SET) != 0) {
            cfs_close(fd);
            return NULL;
```

```
                }
        c = cfs_write(fd, &new_record, sizeof(new_record));
         if (c != sizeof(new_record)) {
                return NULL;
                }
         if (fd != -1) {
                cfs_close(fd);
         }
        enc[16]='\0';
       return enc;

}
//broadcast call back
static void broadcast_recv(struct broadcast_conn *c, \
const rimeaddr_t *from)
{
key_packet recv_packet;
 memcpy(&recv_packet,packetbuf_dataptr(), \
 sizeof(recv_packet));
if(recv_packet.type == BEACON_PACKET)
{
     notify_packet new,old;
     memcpy(&old,&recv_packet,sizeof(old));
if(old.seqnum > seq_num+1)
  {
```

```
                memcpy(&old.source,&rimeaddr_node_addr, \
                        sizeof(new.source));
                seq_num=old.seqnum+1;
                old.seqnum=seq_num;
                packetbuf_copyfrom(&old,sizeof(old));
                broadcast_send(&bc);
                new.type=ACTIVE_PACKET;
                memcpy(&new.source,&rimeaddr_node_addr, \
                                sizeof(new.source));
                memset(&new.dest,&gw,sizeof(new.dest));
                packetbuf_copyfrom(&new,sizeof(new));
                unicast_send(&uc, &gw);
        }
                }
else if((recv_packet.type== ROOT_PACKET) && \
(gk_received =='y'))
{
root_packet new;
memcpy(&new,&recv_packet,sizeof(new));
char set_gk = 'n';
if(gw_set == 'n')
{
  gw.u8[0]=from->u8[0];
  gw.u8[1]=from->u8[1];
  rimeaddr_copy(&gw_default,&new.gw);
```

```
    curr_path=new.len;
    gw_set ='y';
    set_gk = 'y';
    seq_num=new.seqnum+1;
}
else if(new.seqnum > seq_num+1)
{
 gw.u8[0]=from->u8[0];
    gw.u8[1]=from->u8[1];
     rimeaddr_copy(&gw_default,&new.gw);
    curr_path=new.len;
    gw_set ='y';
    set_gk = 'y';
    seq_num=new.seqnum+1;
}
      root_packet send_packet;
      send_packet.len=curr_path+1;
      send_packet.type=ROOT_PACKET;
      send_packet.seqnum=seq_num;
      memcpy(&send_packet.source,&rimeaddr_node_addr, \
           sizeof(send_packet.source));
      memcpy(&send_packet.gw,&gw_default,sizeof(send_packet.gw));
      packetbuf_copyfrom(&send_packet,sizeof(send_packet));
      if(set_gk == 'y')
       broadcast_send(&bc);
```

```
        }

}

static const struct broadcast_callbacks \
 broadcast_callbacks = {broadcast_recv};
static const struct unicast_callbacks \
unicast_callbacks = {recv_uc};

//initialization process
PROCESS_THREAD(example_unicast_process, ev, data)
{
    PROCESS_BEGIN();
    broadcast_open(&bc,137,&broadcast_callbacks);
    unicast_open(&uc, 146, &unicast_callbacks);
    rimeaddr_t addr;
    static struct etimer et;
    uint8_t bin[2 * FC_BYTES + 1];
    uint8_t buf[32];
    int rand,rand2;
    int ln1=0,j=0;
    int l=0;
    node_addr=(int)rimeaddr_node_addr.u8[0];
    etimer_set(&et, CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
```

```
 watchdog_stop();
 core_init();
 for(rand=0;rand<2;rand++)
 {
 unsigned char* enc=fetch_rand();
 memcpy(buf+ln1,enc,16);
 ln1=ln1+16;
  free(enc);
      }
rand_seed(buf, SIZE_SEED);
ep_param_set(SECG_P160);
bn_new(private_key);
ec_new(public_key);
cp_ecdh_gen(private_key,public_key);
ep_print(public_key);
etimer_set(&et, CLOCK_SECOND*1);
while(1)
{
      notify_packet new;
      new.type=HELLO_PACKET;
      new.seqnum=0;
      memcpy(&new.source,&rimeaddr_node_addr, \
      sizeof(new.source));
      memset(&new.dest,&gw,sizeof(new.dest));
      packetbuf_copyfrom(&new,sizeof(new));
```

```
        if(dos_started =='n')
        {
              broadcast_send(&bc);
        }
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
        etimer_reset(&et);
}
        PROCESS_END();
}

//puzzle solving process
PROCESS_THREAD(dos_process, ev, data)
{
    PROCESS_BEGIN();
    static struct etimer est;
    dos_started='y';
    etimer_set(&est, (CLOCK_SECOND*recv_puzzle));
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&est));
    notify_packet new;
    new.type=HELLO_PACKET;
    new.seqnum=2;
    memcpy(&new.source,&rimeaddr_node_addr, \
        sizeof(new.source));
    memset(&new.dest,&gw,sizeof(new.dest));
    packetbuf_copyfrom(&new,sizeof(new));
```

```
    broadcast_send(&bc);
    dos_started='n';

    PROCESS_END();

}
/*---------------------------------------------------------------------------*/
```

Bibliography

[1] H.-I. Yang, "wireless sensor network: the challenges of design and programmability."

[2] M. Tubaishat and S. Madria, "Sensor networks: an overview," *Potentials, IEEE*, vol. 22, no. 2, pp. 20–23, 2003.

[3] N. Sastry and D. Wagner, "Security considerations for ieee 802.15. 4 networks," in *Proceedings of the 3rd ACM workshop on Wireless security*. ACM, 2004, pp. 32–42.

[4] R. Moskowitz, "Hip diet exchange (dex)," *draft-moskowitz-hip-dex-00 (WiP), IETF*, 2012.

[5] Wikepedia, "Wireless Sensor Network," https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes.

[6] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," Tech. Rep., 2014.

[7] Wikepedia, "Advanced Encryption Standard," https://en.wikipedia.org/wiki/Senso_node.

[8] D. G. Padmavathi, M. Shanmugapriya *et al.*, "A survey of attacks, security mechanisms and challenges in wireless sensor networks," *arXiv preprint arXiv:0909.0576*, 2009.

[9] H. Alzaid, E. Foo, and J. G. Nieto, "Secure data aggregation in wireless sensor network: a survey," in *Proceedings of the sixth Australasian conference on Information security-Volume 81*. Australian Computer Society, Inc., 2008, pp. 93–105.

[10] A. Liu and P. Ning, "Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks," in *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on.* IEEE, 2008, pp. 245–256.

[11] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab, "Nanoecc: Testing the limits of elliptic curve cryptography in sensor networks," in *Wireless sensor networks.* Springer, 2008, pp. 305–320.

[12] D. Liu, P. Ning, and R. Li, "Establishing pairwise keys in distributed sensor networks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 1, pp. 41–77, 2005.

[13] S. Hussain, F. Kausar, and A. Masood, "An efficient key distribution scheme for heterogeneous sensor networks," in *Proceedings of the 2007 international conference on Wireless communications and mobile computing.* ACM, 2007, pp. 388–392.

[14] S. Raza, S. Duquennoy, T. Chung, T. Voigt, U. Roedig *et al.*, "Securing communication in 6lowpan with compressed ipsec," in *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on.* IEEE, 2011, pp. 1–8.

[15] Y. Xiao, V. K. Rayi, B. Sun, X. Du, F. Hu, and M. Galloway, "A survey of key management schemes in wireless sensor networks," *Computer communications*, vol. 30, no. 11, pp. 2314–2341, 2007.

[16] C. Karlof and D. Wagner, "Secure routing in wireless sensor networks: Attacks and countermeasures," *Ad hoc networks*, vol. 1, no. 2, pp. 293–315, 2003.

[17] A. D. Wood and J. A. Stankovic, "A taxonomy for denial-of-service attacks in wireless sensor networks," *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*, pp. 739–763, 2004.

[18] A. Agah and S. K. Das, "Preventing dos attacks in wireless sensor networks: A repeated game theory approach." *IJ Network Security*, vol. 5, no. 2, pp. 145–153, 2007.

[19] D. Juneja and N. Arora, "An ant based framework for preventing ddos attack in wireless sensor networks," *arXiv preprint arXiv:1007.0413*, 2010.

[20] P. Nie, J. Vähä-Herttua, T. Aura, and A. Gurtov, "Performance analysis of hip diet exchange for wsn security establishment," in *Proceedings of the 7th ACM symposium on QoS and security for wireless and mobile networks*. ACM, 2011, pp. 51–56.

[21] A. Campbell, J. Omez, S. A. Kim, A. G. Valkó, C.-Y. Wan, and Z. R. Turányi, "Design, implementation, and evaluation of cellular ip," *Personal Communications, IEEE*, vol. 7, no. 4, pp. 42–49, 2000.

[22] D. Kuptsov, B. Nechaev, and A. Gurtov, "Securing medical sensor network with hip," in *Wireless Mobile Communication and Healthcare*. Springer, 2012, pp. 150–157.

[23] R. Hummen, H. Wirtz, J. H. Ziegeldorf, J. Hiller, and K. Wehrle, "Tailoring end-to-end ip security protocols to the internet of things," in *Network Protocols (ICNP), 2013 21st IEEE International Conference on*. IEEE, 2013, pp. 1–10.

[24] H. Chan, A. Perrig, and D. Song, "Key distribution techniques for sensor networks," in *Wireless sensor networks*. Springer, 2004, pp. 277–303.

[25] M. Raghini, N. U. Maheswari, and R. Venkatesh, "Overview on key distribution primitives in wireless sensor network," *Journal of Computer Science*, vol. 9, no. 5, p. 543, 2013.

[26] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[27] Wikepedia, "Advanced Encryption Standard," https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[28] J. Laganier and F. Dupont, "An ipv6 prefix for overlay routable cryptographic hash identifiers version 2 (orchidv2)," 2014.

[29] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson, "Host identity protocol," Tech. Rep., 2008.

[30] H. Krawczyk, "Sigma: The sign-and-macapproach to authenticated diffie-hellman and its use in the ike protocols," in *Advances in Cryptology-CRYPTO 2003*. Springer, 2003, pp. 400–425.

[31] T. Winter, P. Thubert, T. Clausen, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, and J. Vasseur, "Rpl: Ipv6 routing protocol for low power and lossy networks, rfc 6550," *IETF ROLL WG, Tech. Rep*, 2012.

[32] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 2004, pp. 455–462.

[33] D. F. Aranha and C. P. L. Gouvêa, "RELIC is an Efficient LIbrary for Cryptography," https://github.com/relic-toolkit/relic.

[34] B. Gladman, "Byte Oriented AES (Low Resource Version)," https://github.com/BrianGladman/AES.

Curriculum Vitae

Panneer Selvam Santhalingam graduated from Anna University, Chennai, India, in 2010. He received his Bachelor of Engineering from there. He was employed as an Application Developer for three years at iNautix and received his Master of Science in Information Security and Assurance from George Mason University in 2016.