Università della Calabria

DIPARTIMENTO DI INGEGNERIA INFORMATICA MODELLISTICA ELETTRONICA E SISTEMISTICA

Corso di Laurea Magistrale in Ingegneria Informatica

Relazione Progetto Network Security

Candidati:

Luca Cataldi 207148

Francesco Musmanno 216634

Relatore:

Prof. Floriano De Rango

Anno Accademico 2020-2021

Indice

1 VANET Network **3**
1.1 Introduzione . . . 3
1.2 Modalità di comunicazione . . . 4
1.3 Rischi di Sicurezza . . . 5

2 AODV Protocol - RFC 3561 **7**
2.1 Introduzione . . . 7
2.2 Funzionamento del protocollo . . . 7
2.2.1 Route Requests RREQs . . . 8
2.2.2 Route Replies RREPs . . . 9
2.2.3 Route Errors RRERs . . . 9
2.3 Vantaggi . . . 9
2.4 Svantaggi . . . 10

3 SAODV Protocol **11**
3.1 Introduzione . . . 11
3.2 Funzionamento del protocollo . . . 11
3.3 Estensione pacchetti RREQ e RREQ . . . 13

4 Ambiente di simulazione OMNeT++ **14**
4.1 Tool aggiuntivi . . . 14
4.1.1 Instant Veins . . . 14
4.1.2 Inet Framework . . . 14
4.1.3 Veins . . . 15
4.1.4 Sumo . . . 16
4.1.5 Crypto++ . . . 17
4.1.6 OpenStreetMap . . . 17
4.2 Descrizione OMNeT++ e dei suoi elementi principali . . . 18
4.3 Contesto di simulazione . . . 20
4.4 Components: Modules and Channels . . . 20
4.5 VANET Network su OMNeT++ . . . 21
4.5.1 Integrated Visualizer . . . 22
4.5.2 Configurator: IPv4NetworkConfigurator . . . 22
4.5.3 RadioMedium . . . 23
4.5.4 ScenarioManager . . . 23
4.5.5 AodvRouter . . . 24

5 Sviluppo Idea Progettuale **25**
5.1 Generazione mappa 25
5.1.1 downtown.net.xml 25
5.1.2 downtown.rou.xml 26
5.1.3 downtown.sumo.cfg 26
5.1.4 downtown.launchd.xml 27
5.2 File di configurazione 27
5.2.1 omnetpp.ini 27
5.2.2 RoutingScenario.ned 29
5.2.3 VeinsInetRoutingCar.ned 30
5.3 Configurazione protocollo AODV 31
5.3.1 AodvBase 31
5.3.2 Inizializzazione Crypto++ per SecureAodv 31
5.3.3 SecureAodv 33
5.3.4 AodvControlPackets 41
5.3.5 AodvAttacker 44
5.3.6 AodvAttackerRSA 47

6 Simulazioni **50**
6.1 Scenario 1 - Funzionamento Aodv Standard 51
6.1.1 Risultati empirici nodi 51
6.1.2 Risultati empirici pacchetti 52
6.2 Scenario 2 - Funzionamento Aodv con Attaccante 53
6.2.1 Risultati empirici nodi 53
6.2.2 Risultati empirici pacchetti 55
6.3 Scenario 3 - Funzionamento Secure Aodv Standard 56
6.3.1 Risultati empirici nodi 56
6.3.2 Risultati empirici pacchetti 57
6.4 Scenario 4 - Funzionamento Secure Aodv con Attaccante 58
6.4.1 Risultati empirici nodi 58
6.4.2 Risultati empirici pacchetti 60
6.5 Osservazioni Finali 61

7 Codice sorgente progetto **62**

Capitolo 1

VANET Network

1.1 Introduzione

Per introdurre il concetto di *Vehicular Ad-hoc Networks* (VANETs), si tratta innanzitutto delle MANET. In telecomunicazioni, una rete *Mobile Ad-hoc Network* è un sistema autonomo di *host* mobili connessi tramite connessioni *wireless* di tipo *ad-hoc*. Una rete Ad-hoc implica la mancanza di un elemento centrale che organizzi la struttura della rete stessa, quindi è un sistema autonomo di *router* mobili e dei loro host associati, che si organizzano arbitrariamente. Una particolare tipologia di MANETs sono le VANETs, formate da nodi che non sono altro che veicoli dotati di dispositivi con tecnologie di comunicazione wireless a medio raggio che si scambiano le informazioni e si spostano su dei "percorsi" obbligati definiti dalla conformazione dell'infrastruttura stradale sulla quale si trovano. L'obiettivo che ha portato allo sviluppo sempre maggiore di questo tipo di rete è quello di rendere più sicuri gli spostamenti veicolari sulle strade di tutto il mondo. I principi che queste reti vogliono implementare e rispettare sono:

- I movimenti dei veicoli risultano strutturati e vincolati dalla rete viaria;
- I veicoli conoscono la propria geo-localizzazione, (ad esempio via GPS);
- La comunicazione tra veicoli si sviluppa principalmente senza infrastrutture terze, ogni veicolo può trasmettere, riceve ed inoltrare messaggi autonomamente;
- Queste comunicazioni presentano contenuti informativi come avvisi di incidente, allarmi stradali e informazioni sul traffico.

Le VANET fanno tesoro delle capacità di auto-coordinamento e consapevolezza collaborativa delle società evolute, impedendo l'insorgere di criticità evitabili (nessuno imboccherebbe una strada interrotta, se potesse saperlo in anticipo). Di conseguenza, la migliore efficienza della mobilita si tramuta in risparmio energetico e di tempo, ossia una migliore qualità della vita. Purtroppo, però, la VANET può avere molti punti deboli sfruttabili dai cosiddetti attaccanti per limitarne le funzionalità e questo è il motivo principale per cui ancora oggi non si è giunti ad una vera e propria applicazione reale di questa tecnologia di assoluta importanza.

1.2 Modalità di comunicazione

All'interno di questo paragrafo vengono evidenziate le modalità di comunicazione delle reti VANET:

- *Beaconing*: ogni nodo trasmette ciclicamente (ogni 0,1-1 sec) alcune informazioni destinate ai nodi limitrofi, comunicando ad esempio la propria identità, posizione, direzione, velocità e informazioni aggregate relative a nodi più distanti. All'aumentare della densità dei veicoli, aumenta l'impegno del mezzo trasmissivo, ostacolando la propagazione dei messaggi di emergenza. La potenza di tutti va regolata allo scopo di garantire ad ogni mezzo la percezione di tutti i suoi vicini.

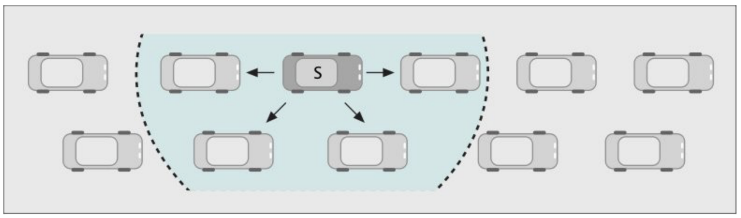

Figura 1.1: Comunicazione *Beaconing*

- *Geobroadcast*: impiegato, ad esempio, per notificare i veicoli in arrivo di eventi improvvisi e condizioni anomale. Il mittente trasmette in *broadcast* il messaggio a tutti i suoi vicini, che lo ritrasmettono solo entro i limiti dell'area designata. Importante notare che la priorità dei messaggi è maggiore di quella dei *Beacons*, il mezzo trasmissivo deve mantenere libera una capacità sufficiente a garantire la propagazione del *Geobroadcast*.

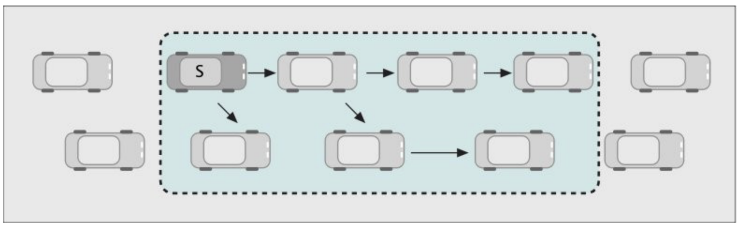

Figura 1.2: Comunicazione *Geobroadcast*

- *Routing Unicast*: in questo caso la rete *ad-hoc* è utilizzata per il trasporto punto-punto di un messaggio, anziché per la sua distribuzione.

- *Information dissemination*: ha lo scopo di mantenere viva una informazione per prolungati periodi di tempo, rendendola disponibile anche ai nodi che arrivano sulla rete successivamente, i messaggi vengono memorizzati e reinoltrati in successione, per evitare il sovraccarico, i messaggi vengono prioritizzati (o scartati) sia in base al contesto, che li rende più o meno rilevanti, sia in base al rilevamento del volume di traffico.

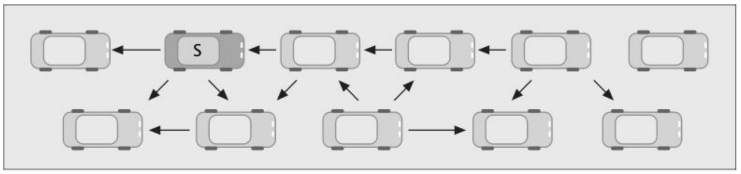

Figura 1.3: Comunicazione *Information Dissemination*

- *Information aggregation*: questa volta i messaggi non sono semplicemente inoltrati, ma elaborati ed aggregati da ogni veicolo che li riceve. Questo approccio mira a ridurre il sovraccarico, come ad esempio, nel caso in cui lo stesso evento è annunciato da più veicoli.

Segue una rappresentazione dell'architettura di elaborazione, archiviazione e comunicazione dei nodi, idonea ad assolvere i requisiti di inoltro di pacchetto e di comunicazione:

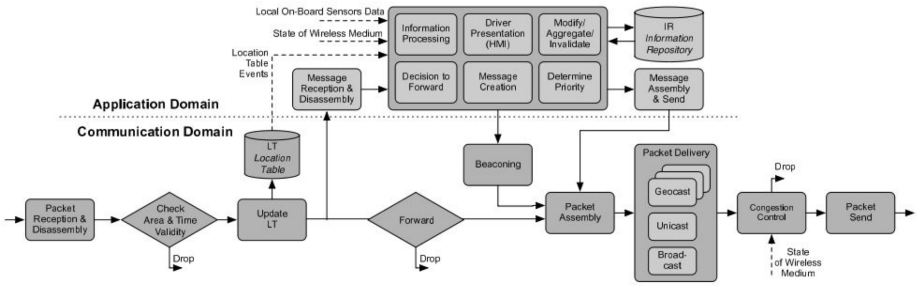

Figura 1.4: Architettura dei nodi

1.3 Rischi di Sicurezza

Le VANETs si prestano ad una serie molto ampia di abusi:

- sabotaggio alla viabilità mediante falsi avvisi di emergenza, incidente e congestione;
- simulazione di mezzi di emergenza per sorpassare altri veicoli;
- violazione della privacy mediante intercettazione, localizzazione e tracciamenti dei veicoli;

- attacchi all'integrità e alla riservatezza, da parte dei nodi di instradamenti intermedi;
- attacco di tipo DOS , quindi interferenza ed impedimento alla comunicazione;
- attacchi di replica da parte di osservatori passivi.

Una possibile soluzione consiste nell'utilizzare diverse *Certification Authority* (CA) firmano i certificati dei veicoli su base regionale. Ogni veicolo possiede un'identità e lungo termine ed una serie di pseudonimi.

- ad ogni identità è associata una coppia di chiave pubblica e chiave privata ed un certificato firmato dalla CA;
- solo la CA conosce la corrispondenza tra pseudonimi ed identità reale;
- l'identità e le chiavi a lungo termine sono usate solo per comunicare con le CA;
- le comunicazioni tra veicoli avvengono via pseudonimi cambiati periodicamente.

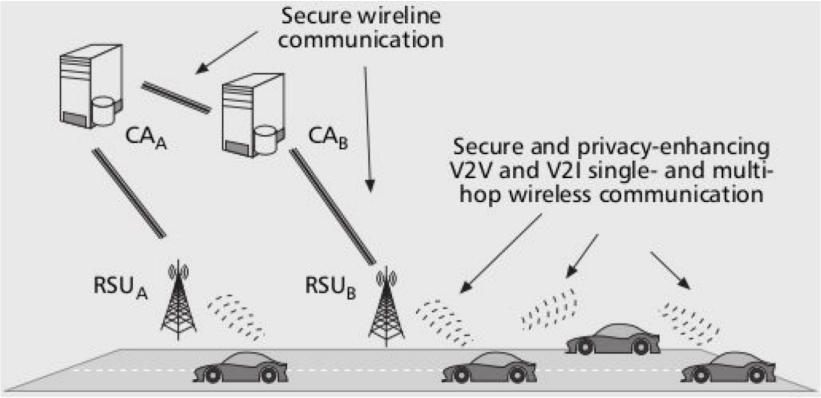

Figura 1.5: Livelli di comunicazione

Capitolo 2

AODV Protocol - RFC 3561

2.1 Introduzione

L'algoritmo *Ad hoc On-Demand Distance Vector* (AODV) permette il routing dinamico, autonomo e *multihop* tra nodi mobili partecipanti che desiderano stabilire e mantenere una rete ad hoc. AODV permette ai nodi mobili di ottenere rapidamente le *routes* per nuove destinazioni e non richiede ai nodi di mantenere rotte verso destinazioni che non sono in comunicazione attiva. AODV permette ai nodi mobili di rispondere alle interruzioni dei collegamenti e ai cambiamenti nella topologia della rete in modo tempestivo. Il funzionamento di AODV è privo di loop e, evitando il problema di Bellman-Ford "*counting to infinity*", offre una rapida convergenza quando la topologia della rete ad hoc cambia (tipicamente, quando un nodo si sposta nella rete). Quando i collegamenti si rompono, AODV fa sì che l'insieme dei nodi interessati venga notificato in modo che siano in grado di invalidare i percorsi che utilizzano il collegamento perso. Una caratteristica distintiva di AODV è il suo uso di un numero di sequenza di destinazione per ogni *entry* di rotta. Il numero di sequenza della destinazione viene creato dalla destinazione per essere incluso insieme a qualsiasi informazione di rotta che invia ai nodi richiedenti. L'uso del numero di sequenza della destinazione assicura l'assenza di loop ed è semplice da programmare. Data la scelta tra due rotte verso una destinazione, un nodo richiedente è tenuto a selezionare quello con il numero di sequenza maggiore.

2.2 Funzionamento del protocollo

Il protocollo di *routing* AODV è progettato per reti mobili ad hoc con popolazioni da decine a migliaia di nodi mobili. AODV può gestire tassi di mobilità bassi, moderati e relativamente alti, così come una una varietà di livelli di traffico dati. AODV è progettato per l'uso in reti dove tutti i nodi possono fidarsi l'uno dell'altro, sia attraverso l'uso di chiavi preconfigurate, o perché si sa che non ci sono nodi malintenzionati. AODV è stato progettato per ridurre la diffusione del traffico di controllo ed eliminare l'*overhead* sul traffico di dati al fine di migliorare la scalabilità e le prestazioni.

Le tipologie di messaggi che il protocollo utilizza sono:

- *Route Requests* (RREQs);
- *Route Replies* (RREPs);
- *Route Errors* (RERRs).

Questi messaggi vengono scambiati utilizzando il protocollo UDP e processando l'*header* IP. Il nodo richiedente utilizzerà il proprio indirizzo IP come "*originator* IP" per il messaggio, se il messaggio è di tipo *broadcast*, viene usato l'indirizzo IP *broadcast* (255.255.255.255).

2.2.1 Route Requests RREQs

Nel momento in cui un nodo necessita di una *route* per determinare una nuova destinazione tale nodo invia una richiesta in *broadcast* (RREQ) nel tentativo di raggiungere una nuova destinazione. Il pacchetto RREQ che viene inviato è strutturato nel modo seguente:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |J|R|G|D|U|   Reserved          |   Hop Count   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            RREQ ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination IP Address                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Destination Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Originator IP Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Originator Sequence Number                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 2.1: Struttura pacchetto RREQ

All'interno del messaggio RREQ viene specificato il *request ID*, il quale viene incrementato ogni volta che il nodo sorgente genera nuova RREQ, ed insieme all'indirizzo IP presente all'interno del pacchetto permette di identificare univocamente il messaggio nella rete. Nel caso in cui un nodo abbia già ricevuto un pacchetto avente la stessa coppia *request ID - indirizzo IP*, il pacchetto viene scartato, altrimenti ne viene effettuato il *forwarding* sempre in *broadcast*. Questo processo termina nel momento in cui il pacchetto RREQ raggiunge il nodo destinazione, oppure quando un nodo intermedio ha all'interno della sua tabella di *routing* una *entry* valida, relativa al nodo destinazione. Una *entry* della *routing table* si definisce valida se dispone di un *sequence number* associato grande almeno quello contenuto all'interno della RREQ. Nel momento in cui il processo appena descritto termina, il nodo destinatario invia al nodo richiedente un messaggio di tipo RREP, descritto nel paragrafo seguente.

2.2.2 Route Replies RREPs

Ogni nodo che riceve le richieste, memorizza le informazioni per permettere che il messaggio di risposta RREP, in caso di riscontro positivo, arrivi correttamente al nodo richiedente, passando per i nodi intermedi che possono contattare il nodo richiedente o se possibile contattando direttamente il nodo che ha effettuato la richiesta nel caso in cui non ci sia necessità di passare per nodi intermedi. La struttura del pacchetto RREP è la seguente:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |R|A|    Reserved     |Prefix Sz|   Hop Count   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination IP address                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Destination Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Originator IP address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Lifetime                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 2.2: Struttura pacchetto RREP

2.2.3 Route Errors RRERs

Tutti i nodi monitorano lo stato dei collegamenti degli *hops* nelle *route* attive, nel momento in cui si verifica la rottura di un collegamento su una delle *route* viene utilizzato il messaggio di tipo RERR per notificare gli altri nodi della rottura del collegamento e in cui viene specificato quelle destinazioni che non sono più raggiungibili usando quella specifica *route*. Per sapere quali nodi notificare ogni nodo mantiene una lista, denominata "*precursor list*" contenente gli indirizzi IP di tutti i vicini che utilizzavano o avrebbero potuto utilizzare quella *route* come *next hop* per raggiungere determinati nodi. Gli indirizzi IP vengono salvati in questa lista quando viene generato un messaggio RREP. La struttura del pacchetto RRER è definita nell'immagine seguente:

2.3 Vantaggi

Tra i principali vantaggi che il protocollo AODV offre si hanno:

1. Non necessita di nessun sistema di controllo centralizzato per gestire il processo di *routing*, essendo un protocollo reattivo tende a ridurre l'*overhead* per i messaggi di controllo al costo di aumentare la latenza per trovare le *route*.

2. I messaggi di HELLO hanno una distanza limitata pertanto non generano un *overhead* rilevante. È un protocollo molto attento anche ai consumi di energia, infatti il nodo di destinazione risponde solo alla prima richiesta che

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |N|          Reserved           |   DestCount   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Unreachable Destination IP Address (1)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Unreachable Destination Sequence Number (1)           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
|  Additional Unreachable Destination IP Addresses (if needed)  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Additional Unreachable Destination Sequence Numbers (if needed)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 2.3: Struttura pacchetto RRER

riceve, le altre richieste vengono ignorate e dunque viene anche mantenuta al più solo una *entry* per ogni destinazione.

3. Nel momento in cui un nodo debba scegliere tra due *route*, viene scelta quella con il numero di sequenza di destinazione più grande.

2.4 Svantaggi

Sfortunatamente il protocollo AODV porta con sè anche degli svantaggi, in quanto essendo applicato in reti dove i nodi dispongono di mobilità è difficile determinare quando una *route* non è più valida e il protocollo di per sè non raccoglie molte informazioni riguardanti il *routing* in quanto limitate al solo nodo sorgente che invia il pacchetto. Ciò comporta inevitabilmente un maggiore utilizzo delle richieste effettuate in *broadcast* per effettuare l'*update* delle *routing tables*, che porta alla presenza di trasmissioni ridondanti che possono causare il cosiddetto *broadcast storm*. La rete non deve essere molto grande, poiché con reti molto grandi le performance di AODV degradano in particolare per l'aumento della lunghezza media dei *path* della rete. Infatti, la presenza di *path* più lunghi rendono, in media, la rete più vulnerabile alla rottura dei link tra nodi e comporta un *overhead* non indifferente. Il protocollo AODV inoltre viene generalmente utilizzato nelle reti ritenute "*trusted*" in quanto è basato sull'assunzione che tutti i nodi della rete cooperano tra di loro. Qualora non ci fosse cooperazione tra i nodi non potrebbero essere generate le tabelle di *routing* e i pacchetti non verrebbero inoltrati correttamente a destinazione. Poiché il protocollo AODV ritiene tutti i nodi della rete fidati, non è stata implementata nessuna politica di sicurezza, infatti esiste una versione del protocollo che implementa politiche di sicurezza che prende il nome di *Secure AODV* (sAODV) che verrà trattato nel capitolo seguente.

Capitolo 3

SAODV Protocol

3.1 Introduzione

I protocolli *Manet* sono stati progettati senza avere in mente la sicurezza, nella maggior parte delle loro specifiche si presume che tutti i nodi della rete siano fidati. La questione della sicurezza è stata rimandata a future implementazioni. Tuttavia, rendere sicure le trasmissioni di rete senza rendere sicuri i protocolli di *routing* non è sufficiente. Inoltre, adattando sistemi (come IPSec) la sicurezza non viene necessariamente raggiunta. Nelle reti *manet* con esigenze di sicurezza, ci devono essere due sistemi di sicurezza: uno per proteggere la trasmissione dei dati e uno per rendere sicuro il protocollo di *routing*. Pertanto nel corso degli anni sono state sviluppate alcune metodologie per aggiungere sicurezza ai vari protocolli di *routing* e, in particolare al protocollo AODV.

3.2 Funzionamento del protocollo

Secure AODV è un'estensione del protocollo AODV principalmente utilizzando nuovi messaggi di estensione. In questi messaggi di estensione c'è una firma creata dal *digesting* del pacchetto AODV usando la chiave privata del mittente originale del messaggio di Routing (non dei nodi intermedi che lo inoltrano soltanto). Per quanto riguarda i messaggi RREQ e RREP ci sono due alternative: La prima in cui solo le destinazioni finali sono autorizzate a rispondere a un RREQ, e la seconda in cui non c'è tale limitazione. Nella prima, quando viene inviata una RREQ, il mittente firma il messaggio. I nodi intermedi verificano la firma prima di creare o aggiornare una rotta inversa verso quell'*host*. E solo se la firma è corretta essi memorizzano la *reverse route*. Il nodo di destinazione finale firma la RREP con la sua chiave privata. I nodi intermedi e finali, ancora una volta verificano la la firma prima di creare o aggiornare una rotta verso quell'*host*, anche memorizzando la firma con la *entry* della *route*. Nel secondo, quando viene inviata una RREQ, il mittente firma il messaggio. I nodi intermedi verificano la firma prima di creare o aggiornare una rotta inversa verso quell'host. E, di nuovo, solo se la firma è corretta memorizzano il percorso inverso. Ma la differenza è che il messaggio RREQ ha anche una seconda firma che viene sempre memorizzata con il percorso inverso. Questa seconda firma deve essere aggiunta nelle RREP di *gratuitous* di quella RREQ e nelle RREP regolari

alle future RREQ a cui il nodo potrebbe rispondere come nodo intermedio. Un nodo intermedio che vuole rispondere ad una RREQ ha bisogno non solo della rotta corretta, ma anche la firma corrispondente a quella rotta per aggiungerla nella RREP e i campi "*Lifetime*" e "*Originator IP address*" che lavorano con quella firma. Se li ha, genera la RREP, (aggiungendo la firma memorizzata, il tempo di vita e l'indirizzo IP dell'originatore) firma il tempo di vita effettivo e l'indirizzo IP di origine effettivo e invia la RREP. Tutti i nodi che ricevono la RREP e che aggiornano la *route* memorizzano la firma, il *lifetime* e l'indirizzo IP di origine con quella rotta. Se un nodo vuole essere in grado di rispondere come nodo intermedio per una rotta ad un nodo che è stato aggiunto a causa di una RREQ o di una RREP, esso deve memorizzare l'indirizzo IP "RREQ Destination" o "RREP Originator", la durata e la firma. E usarli come "*Signature*", "*Old Lifetime*", e "*Old Originator IP address*" nel messaggio RREP-DSE. I messaggi Hello sono messaggi RREP, quindi sono firmati allo stesso modo. Le estensioni *Hello Interval* non sono firmate. Non c'è nessun attacco da cambiare l'estensione *Hello Interval*. In realtà, se l'estensione dell'intervallo di Hello fosse aggiunta nella firma, i nodi che ricevono un messaggio di Hello da un nodo 'D' non sarebbero in grado di rispondere come nodo intermedio quando un nodo 'S' emetterebbe una RREQ per 'D', perché non avrebbero una firma valida per la RREP senza l'estensione dell'intervallo. I messaggi di estensione che includono una seconda firma includono anche i campi RREP (flags e dimensione del prefisso) che non sono derivabili dalla RREQ ma non azzerati quando si calcola la firma. I messaggi RREP-ACK possono essere autenticati usando una firma digitale, che potrebbe essere verificata da chiunque li riceva. Ogni nodo, che genera o inoltra un messaggio RERR, usa la firma digitale firma digitale per firmare l'intero messaggio e ogni nodo che riceve verifica la firma. Il numero di hop di tutti questi messaggi è autenticato utilizzando una catena di *hash*.

3.3 Estensione pacchetti RREQ e RREQ

La versione del protocollo SAODV adottata all'interno del progetto prevede l'utilizzo di estensioni per i pacchetti RREQ e RREP a singola firma, di seguito è riportata la struttura dell'estensione di entrambi i pacchetti:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     | Hash Function | Max Hop Count |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Top Hash                            |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Sign Method  |H|         Reserved            |  Padd Length  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Public Key                           |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Padding (optional)                        |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Signature                           |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Hash                              |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 3.1: RREQ Single Signature Extension

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     | Hash Function | Max Hop Count |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Top Hash                            |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Sign Method  |H|         Reserved            |  Padd Length  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Public Key                           |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Padding (optional)                        |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Signature                           |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Hash                              |
 ...                                                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 3.2: RREP Single Signature Extension

Capitolo 4

Ambiente di simulazione OMNeT++

4.1 Tool aggiuntivi

All'interno di questo paragrafo sono illustrati tutti i vari tool necessari al funzionamento della simulazione, opportunamente connessi e integrati.

4.1.1 Instant Veins

Instant Veins è una macchina virtuale gratuita basata su sistema operativo Linux che consente di eseguire simulazioni utilizzando OMNeT++ in un ambiente già configurato all'interno del quale sono presenti tutti i *tool* di cui si ha necessità. Inoltre è particolarmente ottimizzata e consente di ridurre drasticamente le tempistiche per l'avvio della simulazione. Viene distribuito come un dispositivo virtuale a file singolo, pronto per essere importato con un clic in software come Oracle VM VirtualBox, VMware Workstation Player o qualsiasi software che supporti l'*Open Virtualization Format* (OVF).

4.1.2 Inet Framework

INET Framework è una libreria di modelli open source per l'ambiente di simulazione OMNet ++. Fornisce protocolli, agenti e altri modelli per ricercatori e studenti che lavorano con reti di comunicazione. INET è particolarmente utile quando si progettano e convalidano nuovi protocolli. INET contiene modelli per lo *stack* Internet (TCP, UDP, IPv4, IPv6, OSPF, BGP, ecc.), protocolli di livello di collegamento cablato e wireless (Ethernet, PPP, IEEE 802.11, ecc.), supporto per la mobilità, protocolli MANET, DiffServ, MPLS con segnalazione LDP e RSVP-TE, diversi modelli di applicazioni e molti altri protocolli e componenti. Diversi altri *framework* di simulazione prendono INET come base e lo estendono in direzioni specifiche, a seconda del contesto che si desidera simulare. Gli agenti ed i protocolli di rete sono rappresentati da componenti che possono essere combinati liberamente per formare *host*, *router*, *switch* e altri dispositivi di rete. Nuovi componenti possono essere programmati dall'utente ei componenti esistenti sono stati scritti in modo che siano facili da capire e modificare. INET

beneficia dell'infrastruttura fornita da OMNeT++. Oltre a utilizzare i servizi forniti dal kernel e dalla libreria di simulazione OMNeT++ (modello dei componenti, parametrizzazione, registrazione dei risultati, ecc.), Ciò significa anche che i modelli possono essere sviluppati, assemblati, parametrizzati, eseguiti e i loro risultati valutati comodamente da dall'IDE di OMNeT++ Simulation o dalla riga di comando.

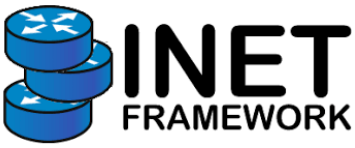

Figura 4.1: Inet Framework

4.1.3 Veins

Veins è un *framework* di simulazione open source per la comunicazione *interveicolare* (IVC) composto da un simulatore di rete basato su eventi (OMNeT++) e un simulatore di traffico stradale (SUMO), che utilizza la cosimulazione. Sostanzialmente è una libreria di modelli open-source per (e un *toolbox* intorno a) OMNeT++, che supporta i ricercatori che conducono simulazioni che coinvolgono veicoli stradali comunicanti, sia come l'obiettivo principale di uno studio che come componente. Veins include già uno stack completo di modelli di simulazione per studiare auto e infrastrutture che comunicano tramite tecnologie basate su IEEE 802.11 in simulazioni di reti ad hoc veicolari (VANET) e sistemi di trasporto intelligenti (ITS). Grazie alla sua modularità, però, può essere usato altrettanto bene come base per modellare altri nodi mobili (come biciclette o pedoni) e tecnologie di comunicazione (dalla banda larga mobile alla luce visibile). Servendo come base per centinaia di pubblicazioni e corsi universitari dal suo inizio nel 2006, oggi Veins è uno degli strumenti più maturi e affermati in questo campo.

Figura 4.2: Veins

4.1.4 Sumo

SUMO (Simulation of Urban MObility), è un pacchetto di simulazione del traffico multimodale open source, altamente portatile, microscopico e continuo, progettato per gestire grandi reti. SUMO permette la modellazione di sistemi di traffico intermodale che includono veicoli stradali, trasporto pubblico e pedoni. Incluso in SUMO c'è una ricchezza di strumenti di supporto che gestiscono compiti come la ricerca del percorso, la visualizzazione, l'importazione della rete e il calcolo delle emissioni. SUMO può essere migliorato con modelli personalizzati e fornisce varie API per controllare a distanza la simulazione. Tra le varie funzionalità che SUMO offre si hanno:

- Guida automatizzata: integrare i veicoli automatizzati nelle simulazioni di traffico ed equipaggiare i veicoli con un dispositivo di transizione del controllo (ToC).
- Comunicazione del veicolo: implementare e valutare le tecnologie di comunicazione C2X accoppiandole a un simulatore di reti di comunicazione (OMNeT++ o ns-3).
- Gestione del traffico: rilevatori video modello e loop di induzione per gestire il traffico in modo interattivo controllando i limiti di velocità, i semafori e il comportamento dei veicoli.
- Simulazione microscopica: simulare il movimento di ogni singolo oggetto modellando esplicitamente tutti i veicoli, i pedoni e il trasporto pubblico.
- Traffico multimodale: combina diversi modi di trasporto e simula auto, autobus, treni, biciclette, pedoni, trasporti pubblici e altro ancora.
- Interazione online: controlla il comportamento di tutti gli oggetti della simulazione durante una simulazione dal vivo con l'interfaccia di controllo del traffico (TraCI).
- Network import: importazione di reti stradali da formati di rete comuni come OpenStreetMap, VISUM, VISSIM, NavTeq, MATsim e OpenDRIVE.
- Generazione della domanda: utilizzare i conteggi del traffico su strade e incroci, le matrici origine-destinazione o i modelli di popolazione virtuale per generare profili di domanda realistici
- Semafori: modifica visivamente gli orari dei semafori con netedit, importa gli orari da fonti di dati esterne o genera automaticamente gli orari.
- Prestazioni: dimensione di rete illimitata, quantità illimitata di veicoli simulati e un tempo di simulazione illimitato.
- Portabilità: utilizzabile su varietà di piattaforme (Windows, Linux o macOS) poiché è implementato in C++ e Python e usa librerie portabili.

Figura 4.3: Sumo

4.1.5 Crypto++

Crypto++ (nota anche come Cryptopp) è una libreria C++ che offre algoritmi crittografici e protocolli, viene originariamente implementata da tale *Wei Dai*. Questa libreria ha subito preso piede, trovando largo uso nel mondo accademico, ma anche in progetti commerciali e *open source*. La prima versione della libreria risale al 1995. Crypto++ fornisce implementazioni complete degli algoritmi crittografici più comuni e non. Si inserisce un'immagine che spiega, più nel dettaglio, quali sono gli algoritmi che questa libreria offre:

Algorithm	Name
authenticated encryption schemes	GCM, CCM, EAX, ChaCha20Poly1305, XChaCha20Poly1305
high speed stream ciphers	ChaCha (8/12/20), ChaCha (IETF) HC (128/256), Panama, Rabbit (128/256), Sosemanuk, Salsa20 (8/12/20), XChaCha (8/12/20), XSalsa20
AES and AES candidates	AES (Rijndael), RC6, MARS, Twofish, Serpent, CAST-256
other block ciphers	ARIA, Blowfish, Camellia, CHAM, HIGHT, IDEA, Kalyna (128/256/512), LEA, SEED, RC5, SHACAL-2, SIMECK, SIMON (64/128), Skipjack, SPECK (64/128), Simeck, SM4,Threefish (256/512/1024), Triple-DES (DES-EDE2 and DES-EDE3), TEA, XTEA
block cipher modes of operation	ECB, CBC, CBC ciphertext stealing (CTS), CFB, OFB, counter mode (CTR), XTS
message authentication codes	BLAKE2b, BLAKE2s, CMAC, CBC-MAC, DMAC, GMAC (GCM), HMAC, Poly1305, SipHash, Two-Track-MAC, VMAC
hash functions	BLAKE2b, BLAKE2s, Keccack (F1600), SHA-1, SHA-2, SHA-3, SHAKE (128/256), SipHash, Tiger, RIPEMD (128/160/256/320), SM3, WHIRLPOOL
public-key cryptography	RSA, DSA, Deterministic DSA (RFC 6979), ElGamal, Nyberg-Rueppel (NR), Rabin-Williams (RW), EC-based German Digital Signature (ECGDSA), LUC, LUCELG, DLIES (variants of DHAES), ESIGN
padding schemes for public-key systems	PKCS#1 v2.0, OAEP, PSS, PSSR, IEEE P1363 EMSA2 and EMSA5
key agreement schemes	Diffie-Hellman (DH), Unified Diffie-Hellman (DH2), Menezes-Qu-Vanstone (MQV), Hashed MQV (HMQV), Fully Hashed MQV (FHMQV), LUCDIF, XTR-DH
elliptic curve cryptography	ECDSA, Deterministic ECDSA (RFC 6979), ed25519, ECGDSA, ECNR, ECIES, x25519, ECDH, ECMQV
insecure or obsolescent algorithms retained for backwards compatibility and historical value	MD2, MD4, MD5, Panama Hash, DES, ARC4, SEAL 3.0, WAKE-OFB, DESX (DES-XEX3), RC2, SAFER, 3-WAY, GOST, SHARK, CAST-128, Square

Figura 4.4: Algoritmi implementati da Crypto++

4.1.6 OpenStreetMap

OpenStreetMap (OSM) è un progetto collaborativo finalizzato a creare mappe del mondo a contenuto libero. Il progetto punta ad una raccolta mondiale di dati geografici, con scopo principale la creazione di mappe e cartografie. La caratteristica fondamentale dei dati geografici presenti in OSM è che vengono distribuiti con una licenza libera, la Open Database License: è cioè possibile utilizzarli liberamente per qualsiasi scopo, anche commerciale, con il solo vincolo di citare la fonte e usare la stessa licenza per eventuali lavori derivati dai dati di OSM. Tutti possono contribuire arricchendo o correggendo i dati. Una alternativa proprietaria che riprendeva le pratiche di crowdsourcing introdotte dalla comunità di OpenStreetMap era Google Map Maker, nel quale i contributi dell'utente potevano essere inseriti su Google Maps a fronte di validazione, impedendone il riutilizzo da parte di terzi. Le mappe sono create usando come riferimento i dati registrati da dispositivi GPS portatili, fotografie aeree ed altre

fonti libere. Sia le immagini renderizzate che i dati vettoriali, oltre che lo stesso database di geodati sono diffusi sotto licenza Open Database License. OpenStreetMap è stato ispirato da siti come Wikipedia: la pagina in cui la mappa è consultabile espone in evidenza un'etichetta "Modifica" per procedere con la modifica dei dati ed il progetto è accompagnato da un archivio storico delle modifiche (cronologia e log). Gli utenti registrati possono caricare nei database del progetto tracce GPS e modificare i dati vettoriali usando gli editor forniti.

Figura 4.5: OpenStreetMap

4.2 Descrizione OMNeT++ e dei suoi elementi principali

Al fine di realizzare uno studio accurato sulle reti VANET, si è resa necessaria la scelta del tool da impiegare per procedere all'implementazione della simulazione, questa è ricaduta su OMNET++ (*Objective Modular Network TestBed*) in C++. È un ambiente *Object-Oriented* per simulazioni di eventi discreti, quali ad esempio:

- reti di comunizazioni *wired*/*wireless*;
- modellazione di protocolli;
- modellazione di sistemi multiprocessore ed altri sistemi distribuiti;
- più in generale, è un ottimo ambiente per la modellazione di qualunque sistema in cui sia applicabile l'approccio ad eventi discreti e che possa essere concretizzato in entita che comunicano attraverso lo scambio di messaggi.

Si intuisce subito l'effettiva utilità di questo tool di simulazione, visto che si lavora con le reti VANET, questo perché OMNET++ fornisce l'infrastruttura ed una serie di tool utili alla realizzazione di simulazioni. Una delle potenzialità di questo sistema è sicuramente data dall'utilizzo un'architettura a componenti per i modelli di simulazione, cosa che rende questo tool performante e responsivo alle specifiche degli utenti. I modelli di simulazione vengono realizzati tramite l'impiego di componenti riusabili chiamati moduli. I vari moduli possono essere connessi tra loro attraverso i *gates* e combinati insieme per poter formare dei moduli composti (*compound modules*) da poter poi utilizzare, è importante sottolineare come non sia imposto un limite sulla profondità di *nesting* dei vari moduli. Come già detto, la comunicazione tra i vari moduli avviene attraverso lo scambio di messaggi (*message passing*) che possono contenere delle strutture dati arbitrarie (ma hanno anche alcune informazioni predefinite come, ad esempio,

i *timestamp*). I vari messaggi possono viaggiare attraverso percorsi predefiniti di *gates* e *connections*, oppure possono essere inviati direttamente alla destinazione desiderata (situazione classica delle comunicazioni *wireless*). Ogni modulo inoltre può poi essere "personalizzato", tramite la definizione di alcuni *parameters* che ne rendono parametrico e volendo anche individuale il comportamento. Per quanto riguarda gli elementi principali, occorre fare una opportuna distinzione. I moduli che racchiudono la logica vera e propria vengono chiamati simple modules; questi vengono scritti in C++ attraverso il potenziale utilizzo di librerie di supporto che sono messe a disposizione da OMNeT++. I vari *simple modules* possono essere raggruppati in *compound modules*, senza avere, come detto, alcun limite di gerarchia. L'intero modello che conterrà poi i vari moduli (simple e compound) verranno racchiusi in un ulteriore componente che prende il nome di network che, di fatto, risulta essere esso stesso un *compound module*.

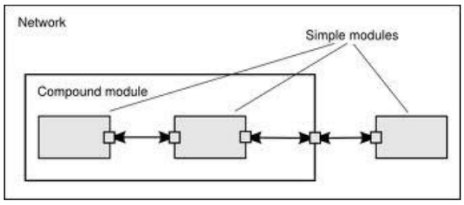

Figura 4.6: Architettura Compound module

In figura è riassunta la situazione sopra descritta, si può notare la presenza dei *simple modules* (rappresentati tramite rettangoli grigi), il *compound module* (rappresentato dal rettangolo bianco, che contiene molteplici *simple modules*) ed inoltre, si nota la presenza della network, che di fatto contiene tutto il resto. Da un'analisi più attenta, però, è anche possibile notare dei moduli più piccoli sui bordi dei *simple modules* e delle frecce bidirezionali che li uniscono. Quei moduli sono i *gates* (sia d'uscita che di entrata) e le frecce sono le connections, questi elementi sono essenziali per la riuscita di una buona comunicazione tra i vari moduli. Piu nel dettaglio, come detto in precedenza, i moduli comunicano attraverso messaggi, i quali possono attraversare vari *gates* oppure essere inviati direttamente al modulo destinazione. I *gates*, quindi, sono le interfacce di input e di output dei moduli: i messaggi sono spediti attraverso *output gates* e arrivano agli input gates. Un input gate e un output gate possono essere collegati da una *connection* (o link). A causa della natura gerarchica del modello, i messaggi tipicamente viaggiano attraverso una catena di connections partendo da e arrivando a *simple modules*. Alle connections possono essere assegnati dei parametri come:

- Propagation delay;
- Data rate;
- Bit error rate.

I moduli possono avere dei *parameters*, usati principalmente per passare dati di configurazione ai *simple modules* e per aiutare a definire la topologia del modello di simulazione. I *parameters* possono essere stringhe, valori numerici o booleani. Oltre all'essere usati come costanti, i *parameters* possono essere anche sorgenti di numeri random secondo una data distribuzione.

4.3 Contesto di simulazione

Al fine di realizzare una simulazione di uno specifico contesto, OMNet++ richiede la specifica di 3 particolari tipi di file:

- File NED (NEtwork Description File) con estensione '.ned' che vengono utilizzati per la descrizione della rete da simulare; in questo file vengono specificati l'insieme dei moduli necessari ed i vari collegamenti (che possono essere *wired* o *wireless*). Per la descrizione di ciascun modulo si fa uso dei *parameters*.

- Message definitions con estensione '.msg' che servono appunto per la definizione di messaggi (che ereditano da message oppure da packet) descrivendone i vari attributi previsti. Un particolare tool di OMNet++ (opp_msgc), si impegnerà poi nella traduzione in una corrispondente classe C++.

- Sorgenti dei simple module che non sono altro che dei file C++ (in particolare si generano due file, uno con estensione '.h' ed uno con estensione '.cc'). Di fatto descrivono il comportamento di ciascun simple module (che deve ovviamente fare riferimento ad esso).

Il sistema di simulazione inoltre fornisce i seguenti componenti:

1. Simulation kernel: contiene il codice che gestisce la simulazione e la sua class library;

2. User interfaces: usate durante l'esecuzione della simulazione per facilitare debugging, dimostrazioni ed esecuzioni batch.

Ultimo, ma non per importanza è il file omnetpp.ini, fondamentale per la simulazione. È un file di configurazione, di fatto contiene tutti i *settings* che controllano come venga eseguita la simulazione, i valori dei vari *parameters* prima definiti, etc. Il file di configurazione può avere al suo interno anche più configurazioni di simulazione, dando la possibilità all'utente di scegliere quale eseguire.

4.4 Components: Modules and Channels

In OMNet++ i *modules* (*simple* o *compound*) ed i *channels* sono chiamati *components*. I components sono rappresentati dalla classe C++ cComponent. La classe astratta per i moduli cModule e la classe astratta per i channel cChannel derivano entrambe da cComponent.

cModule ha due sottoclassi: cSimpleModule e cCompoundModule. L'utente definisce i propri simple modules derivando da cSimpleModule. cChannel invece si deriva in tre sottoclassi: cIdealChannel, cDelayChannel e cDatarateChannel; l'utente può creare i propri channel derivando da cChannel o da una delle sue sottoclassi. Simple modules (e channels) possono essere programmati ridefinendo dei metodi delle classi padre. In particolare cComponent offre i seguenti metodi:

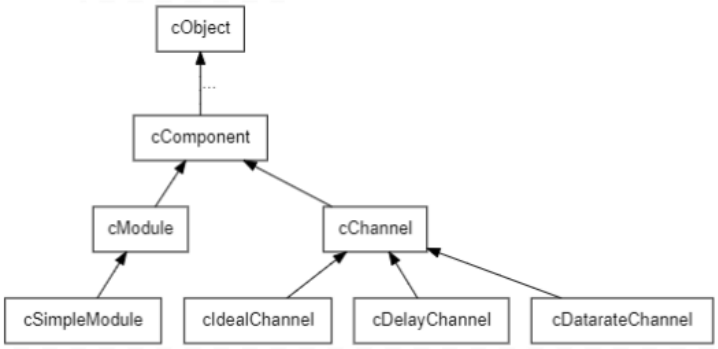

Figura 4.7: Architettura cComponents

- initialize(). Questo metodo viene richiamato dopo che OMNet++ ha costruito la network (istanziando i vari moduli e connettendoli secondo le definizioni NED) e serve ad eseguire codice di inizializzazione.

- finish(). Viene richiamato quando la simulazione termina con successo, utile per il salvataggio di risultati e statistiche.

Per programmare il comportamento di un simple module è necessario ridefinire il metodo handleMessage(cMessage *msg) il quale viene invocato ogni qualvolta il modulo riceve un messaggio; il simulation time non trascorre durante l'esecuzione del metodo ma solo tra le invocazioni. Le funzioni che vengono principalmente usate all'interno di handleMessage sono:

- send(), per inviare un messaggio ad un altro modulo (attraverso un proprio gate di out o input);

- scheduleAt(), per implementare timers (il modulo manda un messaggio a se stesso!);

- cancelEvent(), per eliminare un evento schedulato con scheduleAt.

4.5 VANET Network su OMNeT++

Al fine di realizzare una simulazione riguardante una rete VANET, si è provveduto a realizzare una network opportunamente definita. Alla base della simulazione è utilizzato il framework INET, che fornisce una serie di moduli. Da sottolineare che, a scopo didattico, è stato previsto di mostrare innanzitutto l'andamento in situazione "normale" per quanto riguarda i vari host, concentrandosi sul protocollo AODV, per poi procedere allo studio della rete soggetta ad attacchi. In questa sezione si procederà a descrivere i componenti presenti all'interno della rete, spiegandone anche l'utilizzo all'interno della stessa.

4.5.1 Integrated Visualizer

INET fornisce diverse opzioni per visualizzare i nodi di rete, sia sulla spazio 2D che in 3D. Oltre all'estetica, la personalizzazione dell'aspetto dei nodi può essere utile, ad esempio, per evidenziare determinati nodi o per distinguere i nodi in base alla posizione o alla funzione.

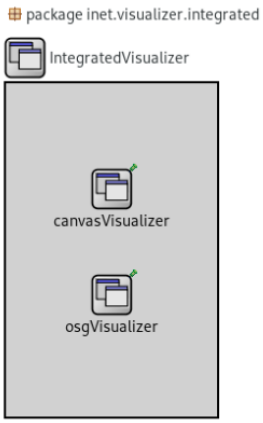

Figura 4.8: Integrated Visualizer

Il progetto in esame prevede la visualizzazione della rete in 2D. Questa modalità di visualizzazione è utile, specialmente, per quanto riguarda lo scambio dei pacchetti di rete.

4.5.2 Configurator: IPv4NetworkConfigurator

Nelle simulazioni INET, i moduli *configurator* sono comunemente usati per assegnare indirizzi IP ai nodi di rete e per impostare le loro tabelle di instradamento. Ci sono vari moduli *configurator* in INET. Il componente relativo a questo ambito che risulta essere il più generico e ricco di funzionalità è *IPv4NetworkConfigurator*. Supporta la configurazione di rete automatica e manuale e le loro combinazioni. Per impostazione predefinita, la configurazione è completamente automatica. Quando la configurazione automatica non produce i risultati desiderati, l'utente può specificare la configurazione manuale per parti (o tutta) della rete e il resto verrà configurato automaticamente. Le varie funzionalità del configuratore possono essere attivate e disattivate tramite parametri. I dettagli della configurazione, come indirizzi IP e percorsi, possono essere specificati in un file XML.

4.5.3 RadioMedium

Il modulo medium descrive il supporto fisico condiviso in cui avviene la comunicazione. Tiene traccia di radio, sorgenti di rumore, trasmissioni in corso, rumore di fondo e altri rumori in corso. Il medium calcola quando, dove e come le trasmissioni e i rumori arrivano ai ricevitori. Fornisce inoltre in modo efficiente l'insieme di trasmissioni e rumori interferenti per i ricevitori.

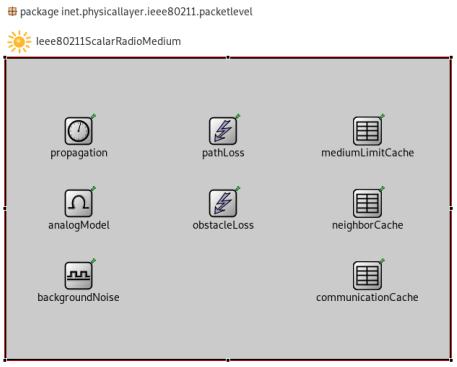

Figura 4.9: Radio Medium

4.5.4 ScenarioManager

ScenarioManager serve per impostare e controllare gli esperimenti di simulazione, è possibile programmare determinati eventi in modo che si verifichino in orari specifici, come la modifica del valore di un parametro, la modifica del tasso di errore in bit di una connessione, la rimozione o l'aggiunta di connessioni, la rimozione o l'aggiunta di percorsi in una tabella di instradamento, ecc., in modo da poter osservare il transitorio comportamento. ScenarioManager esegue uno script specificato in XML. Ha alcuni comandi incorporati, mentre altri comandi vengono inviati per essere eseguiti da determinati moduli semplici. (La classe C ++ di questi simple modules deve implementare l'interfaccia IScriptable e il metodo processCommand () deve essere ridefinito di conseguenza).

4.5.5 AodvRouter

Come ultimo componente, essenziale al fine della simulazione, viene inserito nella rete l'host vero e proprio. Poiché l'obiettivo di tale studio è quello di verificare le funzionalità e la sicurezza del protocollo AODV, si è scelto di creare questi nodi come dei dispositivi AodvRouter, è un modulo compound, offerto dal framework INET; tra i vari componenti presenti al suo interno, uno essenziale è sicuramente il simple module Aodv, che realizza di fatto il protocollo di instradamento su richiesta. Ogni nodo, di fatto, funziona come un router: gestisce le proprie tabelle di routing, invia richieste di rotte e consulta il livello IP per l'invio di dati (si è deciso di utilizzare pacchetti UDP). Ovvio è che, trattandosi di una rete opportunistica, il dispositivo in questione sarà un AdhocHost (in quanto il modulo AodvRouter è un'estensione di quest'ultimo); è un host wireless contenente componenti di routing, mobilità ed energia. Supporta il protocollo di rete IPv4, TCP, UDP e SCTP come protocollo di trasporto. Questo è un tipico nodo mobile che può partecipare al routing ad hoc e può avere applicazioni TCP / UDP installate; supporta volendo anche ICMP (ping).

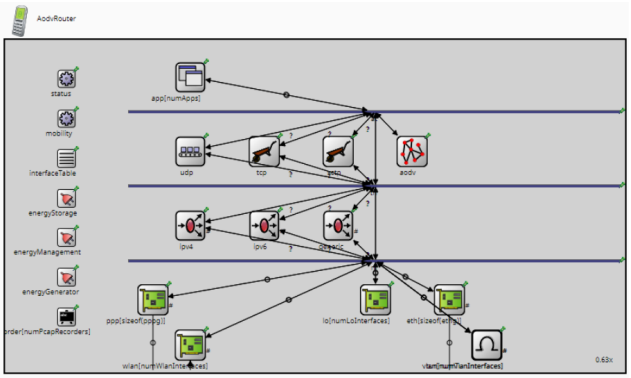

Figura 4.10: AODV Router

Capitolo 5

Sviluppo Idea Progettuale

5.1 Generazione mappa

Per quanto riguarda la generazione della mappa sulla quale viene svolta la simulazione, è stato utilizzato OpenStreetMap per esportare la mappa della città di Orlando. In particolare è stato possibile direttamente da browser, selezionare l'area di interesse per poi esportarla in un file "*downtown.osm*" a partire dal quale sono stati generati tutti i file necessari allo svolgimento della simulazione.

5.1.1 downtown.net.xml

A partire dal file "*downtown.osm*" esportato, mediante il comando:

```
net convert –osm downtown.osm –output-file downtown.net.xml –geometry.remove –roundabouts.guess –ramps.guess –junctions.join –tls.guess-signals –tls.discard-simple –tls.join
```

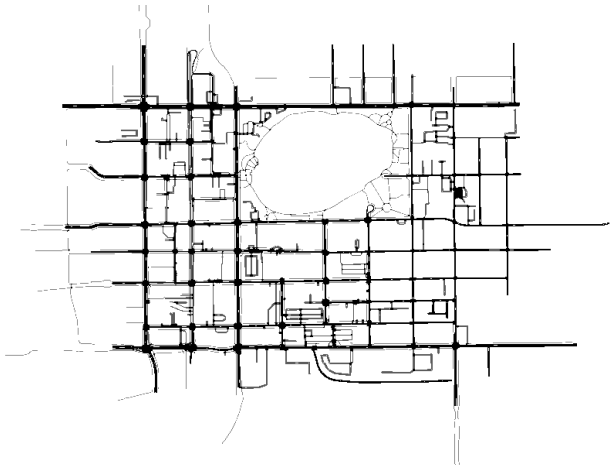

Figura 5.1: "*downtown.net.xml*" in *sumo-gui*

Il file "*downtown.net.xml*" appena generato risulta essere la rappresentazione della mappa esportata da OpenStreetMap in formato xml, è possibile aprire il file mediante *sumo-gui* e risulta come in figura.

5.1.2 downtown.rou.xml

Per la generazione di questo file si ci è avvalsi di uno script python chiamato "*randomTrips.py*", in particolare mediante il comando:

```
randomTrips.py -n downtown.net.xml -e 25 -o downtown.trips.xml
```

A partire dal file "downtown.net.xml" è stato generato un file "downtown.trips.xml" al cui interno sono stati definite i vari punti in cui i 25 veicoli dovranno muoversi, in maniera random. Dopo aver creato i *trips* occorre convertirli in *routes*, per fare ciò utilizziamo il comando:

```
duarouter -n downtown.net.xml –route-file downtown.trips.xml -o downtown.rou.xml –ignore-errors
```

5.1.3 downtown.sumo.cfg

Il file di configurazione di sumo si presenta nel modo seguente:

```xml
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/sumoConfiguration.xsd">

    <input>
        <net-file value="downtown.net.xml"/>
        <route-files value="downtown.rou.xml"/>
    </input>

    <time>
        <step-length value="0.1"/>
    </time>

    <report>
        <xml-validation value="never"/>
        <xml-validation.net value="never"/>
    </report>

    <gui_only>
        <start value="true"/>
    </gui_only>

</configuration>
```

Figura 5.2: File di configurazione "*downtown.sumo.cfg*"

Una volta generato questo file è possibile avviare una simulazione in sumo, mediante l'utilizzo di *sumo-gui*.

5.1.4 downtown.launchd.xml

File *launcher* della simulazione, si presenta nel modo seguente:

```
<launch>
    <copy file="downtown.net.xml" />
    <copy file="downtown.rou.xml" />
    <copy file="downtown.sumo.cfg" type="config" />
</launch>
```

Figura 5.3: File "*downtown.launchd.xml*"

5.2 File di configurazione

5.2.1 omnetpp.ini

Il file ini è il file per la gestione della configurazione della simulazione che verrà lanciata. In questo file vengono definite varie specifiche che sono state prima inserite nel file NED, che possono concernere gli host, l'ambiente di simulazione, il mezzo di comunicazione, la network e quanto segue. Un'altra feature di questo file è che, se si vuole, si può generare più di una configurazione funzionante, tale per cui si possono modificare dei parametri della simulazione stessa, come il protocollo che utilizzano gli host per comunicare, la network che viene implementata, il numero di auto presenti nella simulazione o la lunghezza in secondi della stessa. Si mostrano degli esempi per una maggiore chiarezza:

```
1  [General]
2  network = RoutingScenario
3  sim-time-limit = 600s
4  debug-on-errors = true
5  cmdenv-express-mode = true
6  image-path = ../../../../images
7
8
9  # UDPBasicApp
10 *.node[*].numApps = 1
11 *.node[*].app[0].typename = "UdpBasicApp"
12 *.node[*].app[0].destAddresses = "node[*]"
13 *.node[*].app[0].destPort = 5000
14 *.node[*].app[0].messageLength = 1000B
15 *.node[*].app[0].sendInterval = exponential(12ms)
16 *.node[*].app[0].packetName = "UDPData"
```

Figura 5.4: Estratto 1 del file "*omnetpp.ini*"

Come si può vedere, sotto la configurazione General si definisce il tipo di Network che si vuole che il file ini implementi ed utilizzi all'interno della simulazione. In questo caso si utilizza RoutingScenario (fa riferimento al file RoutingScenario.ned), si definisce anche la durata della simulazione ed altri parametri per il debug. Sono stati inseriti anche dei settings di default per la comunicazione tramite UDP, come il "destAddress" ed il "destPort", si può definire la lunghezza in byte del messaggio e anche il nome del pacchetto.

```
30 # VeinsInetMobility
31 *.node[*].mobility.typename = "VeinsInetMobility"
32 *.node[*].mobility.speed = 20mps
33
34 # VeinsInetManager
35 *.manager.updateInterval = 0.1s
36 *.manager.host = "localhost"
37 *.manager.port = 9999
38 *.manager.autoShutdown = true
39 *.manager.launchConfig = xmldoc("downtown.launchd.xml")
40 *.manager.moduleType = "vanetnetsec.veins_inet.VeinsInetRoutingCar"
```

Figura 5.5: Estratto 2 del file "*omnetpp.ini*"

Si può, inoltre, definire il tipo di mobilità che dovranno rispettare i nodi all'interno della simulazione, puo essere lineare, circolare, rettangolare, in questo caso si fanno rispettare le routes che sono state generate da uno script python (randomTrips.py) partendo da un file OpenStreetMap della mappa di Orlando, Florida. Per permettere al simulatore di utilizzare questa mappa di Orlando tramite SUMO, si inserisce la stringa che fornisce all'InetManager il nome del file "launchd" per la mappa custom che è stata generata. Per ultimo si definisce anche il moduleType che ognuno degli host dovrà rispettare. In aggiunta, si definiscono tutti i parametri che permettono la visualizzazione della mappa, delle strade, dello scambio dei messaggi, è stato inserito anche un modo per vedere qual è il raggio di ricezione/invio delle varie macchine all'interno della simulazione. Ultimo elemento, ma non per importanza, perché risulta di fondamentale importanza per la definizione di quale protocollo venga seguito dalle varie auto host all'interno della simulazione, è la stringa di definizione del router:

```
49 [Config canvas]
50 description = "Scenario SecureAodv sicuro con attaccante"
51
52 # IntegratedCanvasVisualizer (2D)
53 *.visualizer.*.obstacleLossVisualizer.displayIntersections = true
54 *.visualizer.*.obstacleLossVisualizer.displayFaceNormalVectors = true
55 *.visualizer.*.obstacleLossVisualizer.intersectionLineColor = "yellow"
56 *.visualizer.*.mediumVisualizer.signalPropagationAnimationSpeed = 500/3e8
57 *.visualizer.*.mediumVisualizer.signalTransmissionAnimationSpeed = 50000/3e8
58 *.visualizer.*.mediumVisualizer.displaySignals = true
59 *.visualizer.canvasVisualizer.mediumVisualizer.displaySignalDepartures = false
60 *.visualizer.canvasVisualizer.mediumVisualizer.displaySignalArrivals = false
61 *.visualizer.*.physicalLinkVisualizer.displayLinks = true
62 *.visualizer.*.mobilityVisualizer.displayPositions = true
63 *.visualizer.*.mobilityVisualizer.displayOrientations = true
64 *.visualizer.*.mobilityVisualizer.displayVelocities = true
65 *.visualizer.*.mobilityVisualizer.displayMovementTrails = true
66 *.visualizer.*.mobilityVisualizer.moduleFilter = "**.antenna.mobility"
67
68
69 # Routing
70 *.node[2].router = "AodvAttackerRSA"
71 *.node[5].router = "AodvAttackerRSA"
72 #*.node[10].router = "AodvAttackerRSA"
73 *.node[*].router = "SecureAodv"
```

Figura 5.6: Estratto 3 del file "*omnetpp.ini*"

Questa stringa, nelle varie simulazioni, che abbiamo chiamato "Scenari" verrà modificata per andare incontro alle necessità di progetto. Si verificherà il funzionamento normale del protocollo (AodvBase), come reagisce quest'ultimo

nel momento in cui viene inserito un nodo malevolo (AodvAttacker). Successivamente si cambierà protocollo, implementando SecureAodv e si verificherà come questo reagisce, a sua volta, all'inserimento di uno o piu host che implementano AodvAttackerRSA.

5.2.2 RoutingScenario.ned

All'interno del *RoutingScenario.ned* è stato definito come parametro il numero di veicoli presenti all'interno della simulazione (25), inoltre sono stati definiti alcuni *submodules*, in particolare:

- radioMedium: Ieee80211ScalarRadioMedium
- manager: VeinsInetManager
- visualizer: IntegratedVisualizer
- physicalEnvironment: PhysicalEnvironment
- roadsCanvasVisualizer: RoadsCanvasVisualizer

Successivamente vengono definiti i tipi dei nodi che saranno *VeinsInetRoutingCar*.

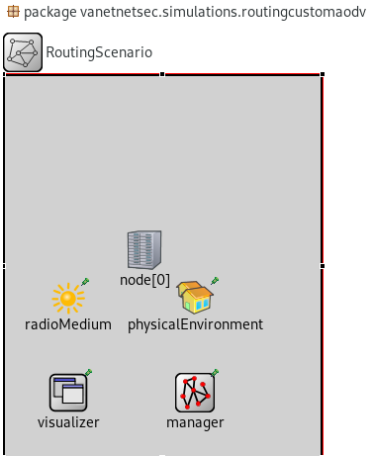

Figura 5.7: *RoutingScenario.ned*

5.2.3 VeinsInetRoutingCar.ned

All'interno di questo file è possibile specificare il tipo di router che deve essere assegnato ai veicoli, in particolare il binding viene effettuato tramite la stringa *router* definita all'interno del file, che viene specificata direttamente in fase di inizializzano all'interno del file "*omnetpp.ini*". Per come il suddetto progetto è stato strutturato è possibile assegnare 4 diversi tipi di router ai nodi della rete, attraverso la specifica del nome del router verrà utilizzato un diverso protocollo di routing. I protocolli definiti sono tutte varianti di AODV e sono i seguenti:

- *AodvBase* - versione base del protocollo AODV;
- *AodvAttacker* - versione del protocollo su cui è stato implementato l'attacco;
- *SecureAodv* - versione del protocollo che integra aspetti di sicurezza e mitigazioni per l'attacco;
- *AodvAttackerRSA* - versione del protocollo su cui è stato implementato l'attacco.

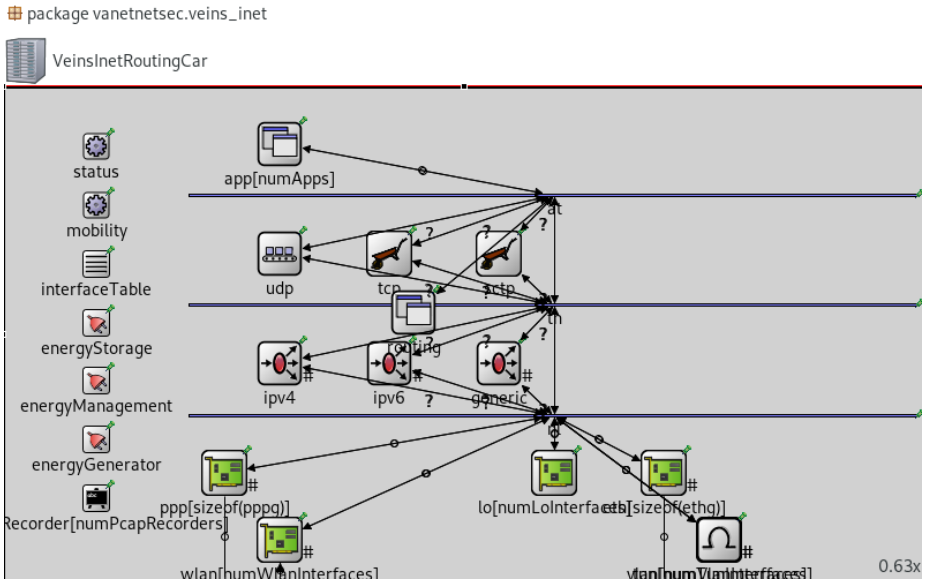

Figura 5.8: *VeinsInetRoutingCar.ned*

5.3 Configurazione protocollo AODV

5.3.1 AodvBase

Rappresentazione delle funzionalità base del protocollo AODV, disponibile attraverso il framework INET. Si compone principalmente di 3 file presenti all'interno di:

VanetNetSec/src/routing/aodv/

- *AodvBase.h*: rappresenta l'interfaccia del protocollo ed espone i metodi che verranno definiti all'interno del file *AodvBase.cc*;
- *AodvBase.cc*: all'interno di questo file vengono definiti i metodi dichiarati nell'*AodvBase.h* e implementa la logica di funzionamento del protocollo AODV;
- *AodvBase.ned*: file ned associato al protocollo AODV.

5.3.2 Inizializzazione Crypto++ per SecureAodv

Nel progetto che si è sviluppato, CryptoPP è stato di fondamentale importanza nell'implementazione del protocollo Secure Aodv. Come già noto, Secure Aodv non è altro che una versione di Aodv in cui viene implementato un layer di sicurezza che cerca di colmare alcune delle lacune del protocollo sottostante (nello specifico, proprio della parte di routing). Più nel dettaglio sono state effettuate delle aggiunte nei pacchetti di tipo RREQ e di tipo RREP per implementare un servizio di Signature e di Hash del messaggio stesso, ognuno per motivi diversi:

- *Signature*: il campo signature, all'interno del pacchetto, è stato aggiunto poiché era necessario un meccanismo di Autenticazione, che permettesse, quindi, al destinatario di essere sicuro del mittente del messaggio.
- *Hash*: il campo hash, all'interno del pacchetto, è stato aggiunto poiché era necessario un meccanismo di Integrità, che desse la possibilità al destinatario del pacchetto, di verificare che questo non fosse stato modificato o manomesso da un nodo malevolo durante la trasmissione dello stesso.

I pacchetti che si sono implementati hanno preso il nome di RREQSEC ed RREPSEC e hanno fatto utilizzo degli algoritmi offerti dalla libreria *CryptoPP* per implementare questi meccanismi, nello specifico, per implementare il meccanismo di Autenticazione si è utilizzato l'algoritmo di RSASS (RSA Signature Scheme), presente in *libcryptopp*, mentre, per quanto riguarda il meccanismo di Integrità si è utilizzato l'algoritmo SHA256, sempre facente parte della libreria *CryptoPP*. Prima di poter iniziare ad implementare quanto detto sopra, si è dovuto generare una coppia di chiavi (privata e pubblica) che gli host avrebbero utilizzato per il corretto funzionamento del protocollo. Si è creato un nuovo progetto C++ all'interno di OMNET++, si è generato un file "KeyGenerator.cc" in cui si è inserito il codice per la generazione di due chiavi RSA 1024 bit, si mostra di seguito:

```
void GenKeyPairRSA()
{
    // InvertibleRSAFunction is used directly only because the private ke
    // won't actually be used to perform any cryptographic operation;
    // otherwise, an appropriate typedef'ed type from rsa.h would have been used.
    CryptoPP::AutoSeededRandomPool rng;
    CryptoPP::InvertibleRSAFunction privkey;
    privkey.CryptoPP::Initialize(rng, 1024);

    // With the current version of Crypto++, MessageEnd() needs to be called
    // explicitly because Base64Encoder doesn't flush its buffer on destruction.
    CryptoPP::Base64Encoder privkeysink(new FileSink("privkeyRSA1024.txt"));
    privkey.DEREncode(privkeysink);
    privkeysink.MessageEnd();

    // Suppose we want to store the public key separately,
    // possibly because we will be sending the public key to a third party.
    CryptoPP::RSAFunction pubkey(privkey);

    CryptoPP::Base64Encoder pubkeysink(new FileSink("pubkeyRSA1024.txt"));
    pubkey.DEREncode(pubkeysink);
    pubkeysink.MessageEnd();

}
```

Figura 5.9: Metodo generazione chiavi

Purtroppo, questo codice, pur essendo corretto, presentava un errore a runtime che non permetteva di eseguirlo all'interno dell'IDE OMNETPP. Per questo motivo, si è provveduto ad eseguire la *build* il file da terminale tramite g++, come segue:

```
Terminal
File  Edit  View  Search  Terminal  Help
veins:~/workspace.omnetpp/KeyGenerator% g++ KeyGenerator-cc -o KeyGenerator -L/usr/local/include -lcryptopp
```

Figura 5.10: Compilazione di "*KeyGenerator.cc*"

Successivamente, si è eseguito il codice:

```
Terminal
File  Edit  View  Search  Terminal  Help
veins:~/workspace.omnetpp/KeyGenerator% ./KeyGenerator
```

Figura 5.11: Esecuzione di "*KeyGenerator.cc*"

Si mostra di seguito il risultato:

- *Private Key RSA 1024*:

Figura 5.12: Chiave privata RSA

- *Public Key RSA 1024*:

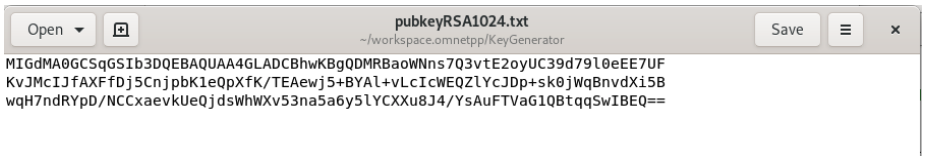

Figura 5.13: Chiave pubblica RSA

5.3.3 SecureAodv

Per l'implementazione di SecureAodv in OMNET, il primo passo che si è effettuato è stato quello di leggere alcuni paper di ricercatori e anche, ovviamente, l'RFC corrispondente. I problemi del protocollo AODV nascono dalla natura del protocollo stesso, che è un protocollo di routing, e viene applicato in contesti di route discovery. Per questo motivo è facilmente vittima di attacchi che vanno a manomettere la generazione della route, come attacchi DDOS, Flooding, Black Hole etc. Avendo fatto un primo lavoro di ricerca e documentazione, si è capito che Secure Aodv non è nient'altro che un layer di sicurezza che viene applicato al di sopra del già rinomato protocollo AODV. Dopo questo attento lavoro di ricerca, si è capito che le features addizionali sono relativamente ai pacchetti RreqSec e RrepSec i quali presentano 3 campi aggiuntivi, rispetto ai tradizionali pacchetti Rreq ed Rrep presenti in AODV. Questi campi servono per implementare due servizi molto importanti:

- Autenticazione: per quanto concerne l'autenticazione del pacchetto, è necessario che questo venga firmato con la chiave privata del mittente, così facendo il destinatario potrà verificare, mediante la chiave pubblica del mittente, che quel messaggio è stato effettivamente inviato dal mittente

"x" e che non ci sia un terzo elemento che si è inserito con intenzioni malevole all'interno della rete. Viene, quindi, aggiunto al tradizionale pacchetto RREQ/RREP un campo "signature" all'interno del quale verrà inserita la firma del mittente, che non sarà altro che il messaggio stesso cifrato tramite chiave privata.

- Integrità: per quanto concerne l'integrità del pacchetto, è necessario che questo venga hashato dal mittente, così facendo il destinatario potrà verificare, una volta ricevuto il pacchetto ed effettuando un *hashing* del pacchetto stesso, che il contenuto del pacchetto non è stato manomesso poiché i due hash corrispondono. Viene, quindi, aggiunto al tradizionale pacchetto RREQ/RREP un campo "hash" e "length". All'interno del primo dei due viene inserito il *digest hashato* del messaggio, mentre all'interno del secondo vi si inserisce la lunghezza dello stesso, che è un parametro che viene utilizzo poi in fase di verifica, dal metodo *verify* della classe hash della libreria CryptoPP

Data questa prima parte di nozioni teoriche, si passa a quella che è stata la vera e propria implementazione di queste features, in particolare sono stati generati, all'interno del path src/routing/aodv del progetto due file:

- *SecureAodv.h*: questo è il file "Header" all'interno del quale vengono definiti tutti i metodi che poi dovranno essere concretizzati, implementati.

- *SecureAodv.cc*: questo è il corrispettivo file all'interno del quale vengono implementati i metodi precedentemente definiti all'interno del file .h.

Si inserisce uno *snippet* di codice del file SecureAodv.h per una maggiore chiarezza, in particolare sono stati ridefiniti tutti i metodi che prima utilizzavano semplici pacchetti Rreq ed Rrep, adesso prenderanno in input i pacchetti custom (a cui sono stati aggiunti i 3 campi di cui sopra) RreqSec ed RrepSec. Si è provveduto a ridefinire quelli che sono i *Control packet creators*, tutti quei metodi che implementano la vera e propria creazione di questi pacchetti:

```
/* Control packet creators */
const Ptr<RrepAck> createRREPACK();
const Ptr<RrepSec> createHelloMessage();
const Ptr<RreqSec> createRREQ(const L3Address& destAddr);
const Ptr<RrepSec> createRREP(const Ptr<RreqSec>& rreq, IRoute *destRoute, IRoute *originatorRoute, const L3Address&
const Ptr<RrepSec> createGratuitousRREP(const Ptr<RreqSec>& rreq, IRoute *originatorRoute);
const Ptr<Rerr> createRERR(const std::vector<UnreachableNode>& unreachableNodes);
```

Figura 5.14: Snippet codice *Control packet creators*

Di conseguenza sono stati modificati anche i *Control packet handlers*, i metodi, cioè, che servono a permettere la ricezione dell'RrepSec o RreqSec per poi definire una possibile risposta:

```
/* Control Packet handlers */
void handleRREP(const Ptr<RrepSec>& rrep, const L3Address& sourceAddr);
void handleRREQ(const Ptr<RreqSec>& rreq, const L3Address& sourceAddr, unsigned int timeToLive);
void handleRERR(const Ptr<const Rerr>& rerr, const L3Address& sourceAddr);
void handleHelloMessage(const Ptr<RrepSec>& helloMessage);
void handleRREPACK(const Ptr<const RrepAck>& rrepACK, const L3Address& neighborAddr);
```

Figura 5.15: Snippet codice *Control packet handlers*

Se il nodo riceve una RreqSec e conosce il nodo richiesto dal mittente o è proprio lui il nodo che è stato richiesto, risponderà con una RrepSec tramite i *Control packet sender methods*:

```
/* Control Packet sender methods */
void sendRREQ(const Ptr<RreqSec>& rreq, const L3Address& destAddr, unsigned int timeToLive);
void sendRREPACK(const Ptr<RrepAck>& rrepACK, const L3Address& destAddr);
void sendRREP(const Ptr<RrepSec>& rrep, const L3Address& destAddr, unsigned int timeToLive);
void sendGRREP(const Ptr<RrepSec>& grrep, const L3Address& destAddr, unsigned int timeToLive);
```

Figura 5.16: Snippet codice *Control packet sender methods*

Altrimenti, il nodo si preoccuperà semplicemente di forwardare il messaggio ricevuto, per permettere la *discovery* del nodo ricercato:

```
/* Control Packet forwarders */
void forwardRREP(const Ptr<RrepSec>& rrep, const L3Address& destAddr, unsigned int timeToLive);
void forwardRREQ(const Ptr<RreqSec>& rreq, unsigned int timeToLive);
```

Figura 5.17: Snippet codice *Control packet forwarders*

Si vuole dare, ora, una migliore spiegazione di come ciò è stato implementato nel file *SecureAodv.cc*. Si parte da due metodi fondamentali per l'implementazione della sicurezza:

- **sign**: viene utilizzato dal nodo mittente per firmare il messaggio prima di inviarlo

```
std::string sign(std::string message) {
    AutoSeededRandomPool rng;

    //Read private key
    CryptoPP::ByteQueue bytes;
    FileSource file(
            "/home/veins/workspace.omnetpp/KeyGenerator/privkeyRSA1024.txt",
            true, new Base64Decoder);
    file.TransferTo(bytes);
    bytes.MessageEnd();
    RSA::PrivateKey privateKey;
    privateKey.Load(bytes);

    // Signature
    RSASS<PKCS1v15, SHA1>::Signer signer(privateKey);

    // Set up for SignMessage()
    byte* signature = new byte[signer.MaxSignatureLength()];
    if ( NULL == signature) {
        return "";
    }

    // Sign...
    size_t length = signer.SignMessage(rng, (const byte*) message.c_str(),
            message.length(), signature);
    len = length;
    std::string sig(reinterpret_cast<const char *>(signature), length);
    std::string messageBase64;
    CryptoPP::StringSource ss(sig, true,
            new CryptoPP::Base64Encoder(
                    new CryptoPP::StringSink(messageBase64)));
    return messageBase64;
}
```

Figura 5.18: Snippet codice metodo *sign*

- **verify**: viene utilizzato dal destinatario per verificare la firma del messaggio

```
bool verify(int length, std::string message, std::string signatureBase64) {
    std::string signature;
    CryptoPP::StringSource ss(signatureBase64, true,
            new CryptoPP::Base64Decoder(new CryptoPP::StringSink(signature)));

    CryptoPP::ByteQueue bytes;
    FileSource file2(
            "/home/veins/workspace.omnetpp/KeyGenerator/pubkeyRSA1024.txt",
            true, new Base64Decoder);
    file2.TransferTo(bytes);
    bytes.MessageEnd();
    RSA::PublicKey pubKey;
    pubKey.Load(bytes);

    RSASSA_PKCS1v15_SHA_Verifier verifier(pubKey);
    bool result = verifier.VerifyMessage((const byte*) message.c_str(),
            message.length(), (const byte*) signature.c_str(), length);

    ////////////////////////////////////
    // Result
    return result;
}
```

Figura 5.19: Snippet codice metodo *verify*

Definiti questi punti cardine, si prendono in considerazione i metodi:

- **createRREQ**: come si puo vedere, questo metodo genera un pacchetto di tipo RreqSec

```
const Ptr<RreqSec> SecureAodv::createRREQ(const L3Address& destAddr)
{
    auto rreqPacket = makeShared<RreqSec>(); // TODO: "AODV-RREQ");
    rreqPacket->setPacketType(usingIpv6 ? RREQ_IPv6 : RREQ);
    rreqPacket->setChunkLength(usingIpv6 ? B(48) : B(24));

    rreqPacket->setGratuitousRREPFlag(askGratuitousRREP);
    IRoute *lastKnownRoute = routingTable->findBestMatchingRoute(destAddr);

    // The Originator Sequence Number in the RREQ message is the
    // node's own sequence number, which is incremented prior to
    // insertion in a RREQ.
    sequenceNum++;

    rreqPacket->setOriginatorSeqNum(sequenceNum);
```

Figura 5.20: Snippet codice metodo *createRREQ*

Di seguito è riportato lo *snippet* di codice in cui viene effettuata la firma del *packet*:

```
//INSERIMENTO DEL CAMPO SIGNATURE NEL PACCHETTO (CIFRATURA DestAddr)

std::string destAddrString = rreqPacket->getDestAddr().str();
std::string message = destAddrString;

std::string messageBase64;
CryptoPP::StringSource ss(message, true,new CryptoPP::Base64Encoder(new CryptoPP::StringSink(messageBase64)));
std::string signature = sign(messageBase64);

//rreqPacket->setHash(signature.c_str());
rreqPacket->setLength(len);
rreqPacket->setSignature(signature.c_str());

//=======================FINE INSERIMENTO SIGNATURE=====================
```

Figura 5.21: Snippet codice firma *packet*

A seguire, si propone lo *snippet* di codice in cui viene generato l'hash del digest che successivamente inserito nel corrispondente campo:

```
// INSERIMENTO DEL CAMPO HASH NEL PACCHETTO (SHA256 DestAddr)

CryptoPP::HexEncoder encoder(new CryptoPP::FileSink(std::cout));

std::string digest;

CryptoPP::SHA256 hash;
hash.Update((const byte*)message.data(), message.size());
digest.resize(hash.DigestSize());
hash.Final((byte*)&digest[0]);

rreqPacket->setHash(digest.c_str());
//=======================FINE INSERIMENTO HASH=======================
```

Figura 5.22: Snippet codice inserimento hash nel pacchetto

- **createRREP**:

```
const Ptr<RrepSec> SecureAodv::createRREP(const Ptr<RreqSec>& rreq, IRoute *destRoute, IRoute *originatorRoute,
{
    auto rrep = makeShared<RrepSec>(); // TODO: "AODV-RREP");
    rrep->setPacketType(usingIpv6 ? RREP_IPv6 : RREP);
    rrep->setChunkLength(usingIpv6 ? B(44) : B(20));

    // When generating a RREP message, a node copies the Destination IP
    // Address and the Originator Sequence Number from the RREQ message into
    // the corresponding fields in the RREP message.

    rrep->setDestAddr(rreq->getDestAddr());

    // OriginatorAddr = The IP address of the node which originated the RREQ
    // for which the route is supplied.
    rrep->setOriginatorAddr(rreq->getOriginatorAddr());
```

Figura 5.23: Snippet codice metodo *createRREP*

```
//INSERIMENTO DEL CAMPO SIGNATURE NEL PACCHETTO (CIFRATURA DestAddr)

std::string destAddrString = rrep->getDestAddr().str();
std::string message = destAddrString;

std::string messageBase64;
CryptoPP::StringSource ss(message, true,new CryptoPP::Base64Encoder(new CryptoPP::StringSink(messageBase64)));
std::string signature = sign(messageBase64);

//rrep->setHash(signature.c_str());
rrep->setLength(len);
rrep->setSignature(signature.c_str());

//=======================FINE INSERIMENTO SIGNATURE=======================
```

Figura 5.24: Snippet codice firma *packet*

```
// INSERIMENTO DEL CAMPO HASH NEL PACCHETTO (SHA256 DestAddr)

CryptoPP::HexEncoder encoder(new CryptoPP::FileSink(std::cout));

std::string digest;

CryptoPP::SHA256 hash;
hash.Update((const byte*)message.data(), message.size());
digest.resize(hash.DigestSize());
hash.Final((byte*)&digest[0]);

rrep->setHash(digest.c_str());

//FINE INSERIMENTO HASH
```

Figura 5.25: Snippet codice inserimento hash nel pacchetto

- **handleRREQ**: questo metodo riceve un pacchetto di tipo RReqSec, ed effettua la verifica della signature a destinazione:

```
void SecureAodv::handleRREQ(const Ptr<RreqSec>& rreq, const L3Address& sourceAddr, unsigned int timeToLive)
{
    EV_INFO << "AODV Route Request arrived with source addr: " << sourceAddr << " originator addr: " << rreq->getOrigi
            << " destination addr: " << rreq->getDestAddr() << endl;

    // A node ignores all RREQs received from any node in its blacklist set.

    auto blackListIt = blacklist.find(sourceAddr);
    if (blackListIt != blacklist.end()) {
        EV_INFO << "The sender node " << sourceAddr << " is in our blacklist. Ignoring the Route Request" << endl;
        return;
    }

    //VERIFICA SIGNATURE DI INTEGRITA' DEL PACCHETTO

    std::string destAddrString = rreq->getDestAddr().str();
    std::string message = destAddrString;
    std::string messageBase64;
    CryptoPP::StringSource ss(message, true,
    new CryptoPP::Base64Encoder(
    new CryptoPP::StringSink(messageBase64)));
    std::string signature = rreq->getSignature();
    bool result = verify(len, messageBase64, signature);

     ////////////////////////////////////// RESULT ////////////////////////////////////

    if (true == result) {
    cout << "Message Verified" << endl;
    } else {
    cout << "Message Verification Failed" << endl;
    return;
    }

    //=======================FINE SIGNATURE=======================
```

Figura 5.26: Snippet codice metodo *handleRREQ*

Successivamente si effettua la verifica dell'hash a destinazione:

```
// VERIFICA DEL CAMPO HASH NEL PACCHETTO (SHA256 DestAddr)
SHA256 hash;
std::string digest = rreq ->getHash();
hash.Update((const byte*)message.data(), message.size());
bool verified = hash.Verify((const byte*)digest.data());

if (verified == true)
    std::cout << "Verified hash over message" << std::endl;
else
    std::cout << "Failed to verify hash over message" << std::endl;

//FINE INSERIMENTO HASH
```

Figura 5.27: Snippet codice verifica hash *packet*

- **handleRREP**: questo metodo riceve un pacchetto RrepSec, quindi si verifica la ricezione del mittente di una REPLY dal nodo destinatario della REQUEST:

```
void SecureAodv::handleRREP(const Ptr<RrepSec>& rrep, const L3Address& sourceAddr)
{
    // 6.7. Receiving and Forwarding Route Replies

    EV_INFO << "AODV Route Reply arrived with source addr: " << sourceAddr << " originator addr: " << rrep->getOrigina
            << " destination addr: " << rrep->getDestAddr() << endl;

    if (rrep->getOriginatorAddr().isUnspecified()) {
        EV_INFO << "This Route Reply is a Hello Message" << endl;
        handleHelloMessage(rrep);
        return;
    }

    //VERIFICA CHECK DI INTEGRITA' DEL PACCHETTO

    std::string destAddrString = rrep->getDestAddr().str();
    std::string message = destAddrString;
    std::string messageBase64;
    CryptoPP::StringSource ss(message, true,
    new CryptoPP::Base64Encoder(
    new CryptoPP::StringSink(messageBase64)));
    std::string signature = rrep->getSignature();
    bool result = verify(len, messageBase64, signature);

    ////////////////////////////////////// RESULT ////////////////////////////////////

    if (true == result) {
        cout << "Message Verified" << endl;
    } else {
        cout << "Message Verification Failed" << endl;
        return;
    }

    //=========================FINE SIGNATURE============================
```

Figura 5.28: Snippet codice verifica *signature*

E successivamente, la verifica dell'hash:

```
// VERIFICA DEL CAMPO HASH NEL PACCHETTO (SHA256 DestAddr)

SHA256 hash;
std::string digest = rrep ->getHash();
hash.Update((const byte*)message.data(), message.size());
bool verified = hash.Verify((const byte*)digest.data());

if (verified == true)
    std::cout << "Verified hash over message" << std::endl;
else
    std::cout << "Failed to verify hash over message" << std::endl;

//FINE INSERIMENTO HASH
```

Figura 5.29: Snippet codice verifica *hash*

Si vuole porre l'accento sul fatto che, questo approccio sicuro rispetto al protocollo AODV, permette una facile intercettazione di nodi malevoli all'interno della rete, poiché se un nodo volesse effettuare un qualsiasi attacco che vada ad inficiare sulla *route discovery*, ad esempio, un attacco di tipo DoS, in cui il nodo malevolo riceve il pacchetto RREQ e invece di "*forwardare*" il pacchetto, lo blocca fingendo di essere lui il nodo ricercato dal mittente e risponde con un RREP da lui generato, questo tipo di messaggio a destinazione verrà subito scartato dal mittente della RREQ poiché, grazie alle misure adottate dal protocollo *SecureAODV*, il nodo sarà capace di rendersi conto che quel pacchetto è

stato inviato da un nodo malevolo in quanto non presenta la firma del nodo che originariamente era il nodo destinatario della RREQ. In pratica, il nodo malevolo, non disponendo della chiave privata del nodo richiesto all'interno della RREQ, non puo fare altro che firmare il proprio messaggio con una sua chiave, questo approccio ovviamente verrà "detected" dal nodo che riceverà questa RREP fasulla, risultando in un drop o scarto del pacchetto. Vediamo i risultati che si ottengono in una simulazione in cui si fa uso del protocollo SecureAodv, ma non c'è presenza di nodi malevoli/attaccanti:

```
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
```

Figura 5.30: Snippet console esecuzione *SecureAodv* senza nodi *Attacker*

Mentre, questi sono i risultati che si ottengono in una simulazione in cui si fa uso del protocollo SecureAodv, ma c'è presenza di nodi malevoli:

```
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verification Failed
Message Verified
Verified hash over message
Message Verified
Verified hash over message
Message Verification Failed
```

Figura 5.31: Snippet console esecuzione *SecureAodv* con nodi *Attacker*

5.3.4 AodvControlPackets

L'insieme dei file seguente:

- *AodvControlPackets_m.h*;
- *AodvControlPackets_m.cc*;
- *AodvControlPackets.msg*;

è di fondamentale importanza per la definizione dei parametri che il pacchetto AODV necessita. Nelle sottosezioni che seguono verrà fornita una spiegazione dettagliata di ciò che viene definito all'interno di questi file.

AodvControlPackets_m.h

Questo file rappresenta l'interfaccia in cui vengono dichiarate le varie classi che il protocollo utilizzerà, in particolare sono le seguenti:

```
class AodvControlPacket;
class Rreq;
class Rrep;
struct UnreachableNode;
class Rerr;
class RrepAck;
class RreqSec;
class RrepSec;
class WaitForRrep;
class PacketHolderMessage;
```

Figura 5.32: Classi implementate

In particolar modo occorre soffermarsi maggiormente su due classi che sono state aggiunte a quella base presente all'interno del framework INET che sono:

- **RreqSec**: pacchetto avente la stessa struttura dell'Rreq di default, presenta l'aggiunta di tre campi utili all'implementazione della versione sicura del protocollo, i campi che il pacchetto espone sono i seguenti:

```
class RreqSec : public ::inet::aodv::AodvControlPacket
{
  protected:
    bool joinFlag = false;
    bool repairFlag = false;
    bool gratuitousRREPFlag = false;
    bool destOnlyFlag = false;
    bool unknownSeqNumFlag = false;
    uint16_t reserved = 0;
    unsigned int hopCount = 0;
    uint32_t rreqId = 0;
    L3Address destAddr;
    uint32_t destSeqNum = 0;
    L3Address originatorAddr;
    uint32_t originatorSeqNum = 0;
    omnetpp::opp_string hash;
    int length = 0;
    omnetpp::opp_string signature;
```

Figura 5.33: Campi pacchetto RreqSec

- **RrepSec**: pacchetto che presenta la stessa struttura dell'Rrep di default, ma estende le sue funzionalità mediante l'aggiunta di tre campi utili all'implementazione della versione sicura del protocollo, i campi che il pacchetto espone sono i seguenti:

```
class RrepSec : public ::inet::aodv::AodvControlPacket
{
  protected:
    bool repairFlag = false;
    bool ackRequiredFlag = false;
    uint16_t reserved = 0;
    unsigned int prefixSize = 0;
    unsigned int hopCount = 0;
    L3Address destAddr;
    uint32_t destSeqNum = 0;
    L3Address originatorAddr;
    omnetpp::simtime_t lifeTime = SIMTIME_ZERO;
    omnetpp::opp_string hash;
    int length = 0;
    omnetpp::opp_string signature;
```

Figura 5.34: Campi pacchetto RrepSec

Di seguito sono esposti i metodi che sono stati implementati all'interno del file "*AodvControlPackets_m.cc*" rispettivamente relativi ai pacchetti RreqSec e RrepSec:

```
public:
  RreqSec();
  RreqSec(const RreqSec& other);
  virtual ~RreqSec();
  RreqSec& operator=(const RreqSec& other);
  virtual RreqSec *dup() const override {return new RreqSec(*this);}
  virtual void parsimPack(omnetpp::cCommBuffer *b) const override;
  virtual void parsimUnpack(omnetpp::cCommBuffer *b) override;
  // field getter/setter methods
  virtual bool getJoinFlag() const;
  virtual void setJoinFlag(bool joinFlag);
  virtual bool getRepairFlag() const;
  virtual void setRepairFlag(bool repairFlag);
  virtual bool getGratuitousRREPFlag() const;
  virtual void setGratuitousRREPFlag(bool gratuitousRREPFlag);
  virtual bool getDestOnlyFlag() const;
  virtual void setDestOnlyFlag(bool destOnlyFlag);
  virtual bool getUnknownSeqNumFlag() const;
  virtual void setUnknownSeqNumFlag(bool unknownSeqNumFlag);
  virtual uint16_t getReserved() const;
  virtual void setReserved(uint16_t reserved);
  virtual unsigned int getHopCount() const;
  virtual void setHopCount(unsigned int hopCount);
  virtual uint32_t getRreqId() const;
  virtual void setRreqId(uint32_t rreqId);
  virtual const L3Address& getDestAddr() const;
  virtual L3Address& getDestAddrForUpdate() { handleChange();return const_cast<L3Address&>(const_cast<RreqSec*>(this)->getDestAddr());}
  virtual void setDestAddr(const L3Address& destAddr);
  virtual uint32_t getDestSeqNum() const;
  virtual void setDestSeqNum(uint32_t destSeqNum);
  virtual const L3Address& getOriginatorAddr() const;
  virtual L3Address& getOriginatorAddrForUpdate() { handleChange();return const_cast<L3Address&>(const_cast<RreqSec*>(this)->getOriginatorAddr());}
  virtual void setOriginatorAddr(const L3Address& originatorAddr);
  virtual uint32_t getOriginatorSeqNum() const;
  virtual void setOriginatorSeqNum(uint32_t originatorSeqNum);
  virtual const char * getHash() const;
  virtual void setHash(const char * hash);
  virtual int getLength() const;
  virtual void setLength(int length);
  virtual const char * getSignature() const;
  virtual void setSignature(const char * signature);
```

Figura 5.35: Metodi pacchetto RreqSec

```
public:
  RrepSec();
  RrepSec(const RrepSec& other);
  virtual ~RrepSec();
  RrepSec& operator=(const RrepSec& other);
  virtual RrepSec *dup() const override {return new RrepSec(*this);}
  virtual void parsimPack(omnetpp::cCommBuffer *b) const override;
  virtual void parsimUnpack(omnetpp::cCommBuffer *b) override;

  // field getter/setter methods
  virtual bool getRepairFlag() const;
  virtual void setRepairFlag(bool repairFlag);
  virtual bool getAckRequiredFlag() const;
  virtual void setAckRequiredFlag(bool ackRequiredFlag);
  virtual uint16_t getReserved() const;
  virtual void setReserved(uint16_t reserved);
  virtual unsigned int getPrefixSize() const;
  virtual void setPrefixSize(unsigned int prefixSize);
  virtual unsigned int getHopCount() const;
  virtual void setHopCount(unsigned int hopCount);
  virtual const L3Address& getDestAddr() const;
  virtual L3Address& getDestAddrForUpdate() { handleChange();return const_cast<L3Address&>(const_cast<RrepSec*>(this)->getDestAddr());}
  virtual void setDestAddr(const L3Address& destAddr);
  virtual uint32_t getDestSeqNum() const;
  virtual void setDestSeqNum(uint32_t destSeqNum);
  virtual const L3Address& getOriginatorAddr() const;
  virtual L3Address& getOriginatorAddrForUpdate() { handleChange();return const_cast<L3Address&>(const_cast<RrepSec*>(this)->getOriginatorAddr());}
  virtual void setOriginatorAddr(const L3Address& originatorAddr);
  virtual omnetpp::simtime_t getLifeTime() const;
  virtual void setLifeTime(omnetpp::simtime_t lifeTime);
  virtual const char * getHash() const;
  virtual void setHash(const char * hash);
  virtual int getLength() const;
  virtual void setLength(int length);
  virtual const char * getSignature() const;
  virtual void setSignature(const char * signature);
```

Figura 5.36: Metodi pacchetto RrepSec

AodvControlPackets_m.cc

All'interno di questo file vengono definiti i metodi dichiarati nell'interfaccia, in particolar modo avendo definito due nuove tipologie di pacchetti nell'interfaccia è stata necessaria l'aggiunta di codice necessario alla gestione di queste nuove tipologie di pacchetti.

AodvControlPackets.msg

Sono state necessarie alcune modifiche anche a questo file, in particolare sono stati aggiunti all'enumerazione iniziale dei valori per le nuove classi implementate RreqSec e RrepSec, inoltre sono stati dichiarati i campi aggiunti ai pacchetti, in particolare:

```
class RreqSec extends AodvControlPacket
{
    bool joinFlag;
    bool repairFlag;
    bool gratuitousRREPFlag;
    bool destOnlyFlag;
    bool unknownSeqNumFlag;
    uint16_t reserved = 0;
    unsigned int hopCount;
    uint32_t rreqId;
    L3Address destAddr;
    uint32_t destSeqNum;
    L3Address originatorAddr;
    uint32_t originatorSeqNum;
    string hash;
    int length;
    string signature;
}
```

Figura 5.37: Campi pacchetto RreqSec

```
class RrepSec extends AodvControlPacket
{

    bool repairFlag;
    bool ackRequiredFlag;
    uint16_t reserved = 0;
    unsigned int prefixSize;
    unsigned int hopCount;
    L3Address destAddr;
    uint32_t destSeqNum;
    L3Address originatorAddr;
    simtime_t lifeTime;
    string hash;
    int length;
    string signature;
}
```

Figura 5.38: Campi pacchetto RrepSec

5.3.5 AodvAttacker

Si compone da un insieme di file che definiscono la simulazione del funzionamento del protocollo AODV sotto uno scenario d'attacco specifico, implementato all'interno di:

- *AodvAttacker.cc*;
- *AodvAttacker.h*;
- *AodvAttacker.ned*.

É stato deciso di implementare un attacco di tipo *Denial Of Service* (DoS), al fine dell'implementazione i nodi malevoli presenti all'interno della rete utilizzeranno invece del router *AodvBase* (precedentemente illustrato) un router di tipo *AodvAttacker* il quale implementerà una versione del protocollo di routing AODV, appunto malevola. Sostanzialmente un nodo della rete che integra questo tipo di router non permette ad ogni pacchetto di tipo Rreq che riceve di essere propagato, inoltre non essendo il nodo destinatario del messaggio genera una Rrep fasulla che viene inoltrata agli altri nodi della rete. L'implementazione viene effettuata apportando le opportune modifiche al file *AodvAttacker.cc* all'interno del quale vengono definiti i metodi che il protocollo utilizza. In particolare i metodi che sono stati ridefiniti sono:

- **handleRREQ**: questo metodo si occupa della gestione dei messaggi di tipo RREQ nel momento in cui vengono ricevuti dal nodo che implementa il protocollo, di seguito in figura è possibile prendere visione dello snippet di codice che definisce il metodo:

```
// check (i) MODIFICA
if (getSelfIPAddress() == getSelfIPAddress()) { // rreq->getDestAddr() == getSelfIPAddress()
    EV_INFO << "I am the destination node for which the route was requested" << endl;

    // create RREP
    auto rrep = createRREP(rreq, destRoute, reverseRoute, sourceAddr);

    // send to the originator
    sendRREP(rrep, rreq->getOriginatorAddr(), 255);

    return;    // discard RREQ, in this case, we do not forward it.
}
```

Figura 5.39: Snippet codice *handleRREQ*

In particolare, nella versione del metodo malevola ridefinita, è stato inserita al posto di:

rreq->getDestAddr() == getSelfIPAddress()

che avrebbe correttamente verificato quale fosse il nodo destinatario del pacchetto, una tautologia che permette al nodo malevolo di intercettare ogni pacchetto di tipo RREQ e rispondere con un pacchetto di tipo RREP fasullo. In questo modo viene eseguito un attacco di tipo *Denial of Service* che per l'appunto non permette di raggiungere il nodo desiderato e che modifica in maniera malevola la struttura della rete fornendo informazioni non corrette al nodo mittente della RREQ e infine forgia un messaggio RREP fasullo.

- **createRREP**: questo metodo si occupa della creazione del messaggio RREQ da inoltrare agli altri nodi appartenenti alla rete, dopo aver illustrato come viene intercettata la richiesta, di seguito si propone uno snippet di codice in cui viene mostrata la generazione del pacchetto RREP:

```
const Ptr<Rrep> AodvAttacker::createRREP(const Ptr<Rreq>& rreq, IRoute *destRoute, IRoute *originatorRoute,
{
    auto rrep = makeShared<Rrep>(); // TODO: "AODV-RREP");
    rrep->setPacketType(usingIpv6 ? RREP_IPv6 : RREP);
    rrep->setChunkLength(usingIpv6 ? B(44) : B(20));

    // When generating a RREP message, a node copies the Destination IP
    // Address and the Originator Sequence Number from the RREQ message into
    // the corresponding fields in the RREP message.

    rrep->setDestAddr(rreq->getDestAddr());

    // OriginatorAddr = The IP address of the node which originated the RREQ
    // for which the route is supplied.
    rrep->setOriginatorAddr(rreq->getOriginatorAddr());

    // Processing is slightly different, depending on whether the node is
    // itself the requested destination (see section 6.6.1), or instead
    // if it is an intermediate node with an fresh enough route to the destination
    // (see section 6.6.2).

    if (getSelfIPAddress() == getSelfIPAddress()) {    //rreq->getDestAddr() == getSelfIPAddress()
        // node is itself the requested destination
        // 6.6.1. Route Reply Generation by the Destination

        // If the generating node is the destination itself, it MUST increment
        // its own sequence number by one if the sequence number in the RREQ
        // packet is equal to that incremented value.
```

Figura 5.40: a) Snippet codice *createRREP*

```
if (!rreq->getUnknownSeqNumFlag() && sequenceNum + 1 == rreq->getDestSeqNum())
    sequenceNum++;

// The destination node places its (perhaps newly incremented)
// sequence number into the Destination Sequence Number field of
// the RREP,
rrep->setDestSeqNum(sequenceNum);

// and enters the value zero in the Hop Count field
// of the RREP.
rrep->setHopCount(999); // MODIFICA prima era 0

// The destination node copies the value MY_ROUTE_TIMEOUT
// into the Lifetime field of the RREP.
rrep->setLifeTime(myRouteTimeout.trunc(SIMTIME_MS));
```

Figura 5.41: b) Snippet codice *createRREP*

Come si evince dalle figure appena riportate, il nodo malevolo genera una RREP fasulla in cui però in modo astuto non modifica l'indirizzo del destinatario (non inserisce il proprio indirizzo IP), successivamente apporta una modifica al parametro *HopCount*, invece di settarlo a 0 secondo le direttive del protocollo lo setta a 999, modificando di conseguenza l'architettura della rete. Poiché il nodo mittente della richiesta, ricevendo una RREP di questo tipo crederà che il nodo da lui richiesto sia molto distante da esso.

Questo tipo di attacco è particolarmente efficiente contro un routing protocol come AODV, poiché il nodo malevolo riceverà il pacchetto REQUEST da parte del mittente, non permetterà il forward dello stesso e anzi, forgerà un pacchetto REPLY fingendosi il vero destinatario dello stesso. Facendo ciò modificherà anche il paremetro "HopCount" e lo setterà a 999, invece che 0, cioè il valore indicato dalle specifiche di protocollo, andando cosi a modificare quella che è l'architettura sttessa della rete. Questo pacchetto, una volta arrivato a destinazione (cioè quando viene ricevuto dal nodo che ha originato la RREQ) verrà accettato, poiché il nodo mittente della RREQ non avrà metodi o strumenti che gli permettano di verificare l'autenticità del messaggio stesso.

5.3.6 AodvAttackerRSA

Sostanzialmente si compone da un insieme di file che definiscono la simulazione del funzionamento del protocollo AODV sotto uno scenario d'attacco specifico, implementato all'interno di:

- *AodvAttackerRSA.cc*;
- *AodvAttackerRSA.h*;
- *AodvAttackerRSA.ned*.

Sulla base dell'attacco definito nel paragrafo precedente, l'implementazione viene effettuata apportando le opportune modifiche al file *AodvAttackerRSA.cc* all'interno del quale vengono definiti i metodi che il protocollo utilizza. In particolare i metodi che sono stati ridefiniti sono:

- **handleRREQ**: questo metodo si occupa della gestione dei messaggi di tipo RREQ nel momento in cui vengono ricevuti dal nodo che implementa il protocollo, di seguito in figura è possibile prendere visione dello snippet di codice che definisce il metodo:

```
// check (i) //  MODIFICA
if (getSelfIPAddress() == getSelfIPAddress()) {
    EV_INFO << "I am the destination node for which the route was requested" << endl;

    // create RREP
    auto rrep = createRREP(rreq, destRoute, reverseRoute, sourceAddr);

    // send to the originator
    sendRREP(rrep, rreq->getOriginatorAddr(), 255);

    return;    // discard RREQ, in this case, we do not forward it.
}

// check (ii)
if (destRouteData && destRouteData->isActive() && destRouteData->hasValidDestNum() &&
    destRouteData->getDestSeqNum() >= rreq->getDestSeqNum())
{
    EV_INFO << "I am an intermediate node who has information about a route to " << rreq->getDestAddr() << endl;
```

Figura 5.42: Snippet codice *handleRREQ*

In particolare, nella versione del metodo malevola ridefinita, è stato inserita al posto di:

rreq->getDestAddr() == getSelfIPAddress()

che avrebbe correttamente verificato quale fosse il nodo destinatario del pacchetto, una tautologia che permette al nodo malevolo di intercettare ogni pacchetto di tipo RREQ e rispondere con un pacchetto di tipo RREP fasullo. In questo modo viene eseguito un attacco di tipo *Denial of Service* che per l'appunto non permette di raggiungere il nodo desiderato e che modifica in maniera malevola la struttura della rete fornendo informazioni non corrette al nodo mittente della RREQ e infine forgia un messaggio RREP fasullo.

```
const Ptr<RrepSec> AodvAttackerRSA::createRREP(const Ptr<RreqSec>& rreq, IRoute *destRoute, IRoute *originatorRoute,
{
    auto rrep = makeShared<RrepSec>(); // TODO: "AODV-RREP");
    rrep->setPacketType(usingIpv6 ? RREP_IPv6 : RREP);
    rrep->setChunkLength(usingIpv6 ? B(44) : B(20));

    // When generating a RREP message, a node copies the Destination IP
    // Address and the Originator Sequence Number from the RREQ message into
    // the corresponding fields in the RREP message.

    rrep->setDestAddr(rreq->getDestAddr());

    // OriginatorAddr = The IP address of the node which originated the RREQ
    // for which the route is supplied.
    rrep->setOriginatorAddr(rreq->getOriginatorAddr());

    // Processing is slightly different, depending on whether the node is
    // itself the requested destination (see section 6.6.1), or instead
    // if it is an intermediate node with an fresh enough route to the destination
    // (see section 6.6.2).

    if (getSelfIPAddress() == getSelfIPAddress()) {    // rreq->getDestAddr() == getSelfIPAddress()
        // node is itself the requested destination
        // 6.6.1. Route Reply Generation by the Destination

        // If the generating node is the destination itself, it MUST increment
        // its own sequence number by one if the sequence number in the RREQ
        // packet is equal to that incremented value.
```

Figura 5.43: a) Snippet codice *createRREP*

```
        if (!rreq->getUnknownSeqNumFlag() && sequenceNum + 1 == rreq->getDestSeqNum())
            sequenceNum++;

        // The destination node places its (perhaps newly incremented)
        // sequence number into the Destination Sequence Number field of
        // the RREP,
        rrep->setDestSeqNum(sequenceNum);

        // and enters the value zero in the Hop Count field
        // of the RREP.
        rrep->setHopCount(999); //MODIFICA, prima era 0

        // The destination node copies the value MY_ROUTE_TIMEOUT
        // into the Lifetime field of the RREP.
        rrep->setLifeTime(myRouteTimeout.trunc(SIMTIME_MS));
    }
```

Figura 5.44: b) Snippet codice *createRREP*

- **createRREP**: questo metodo si occupa della creazione del messaggio RREQ da inoltrare agli altri nodi appartenenti alla rete, dopo aver illustrato come viene intercettata la richiesta, di seguito si propone uno snippet di codice in cui viene mostrata la generazione del pacchetto RREP:

  Come si evince dalle figure appena riportate, il nodo malevolo genera una RREP fasulla in cui però in modo astuto non modifica l'indirizzo del destinatario (non inserisce il proprio indirizzo IP), successivamente apporta una modifica al parametro *HopCount*, invece di settarlo a 0 secondo le direttive del protocollo lo setta a 999, modificando di conseguenza l'architettura della rete. Poiché il nodo mittente della richiesta, ricevendo una RREP di questo tipo crederà che il nodo da lui richiesto sia molto distante da esso.

La versione del protocollo implementata risulta essere particolarmente dannosa se tutti i nodi della rete adottano la versione base del protocollo AODV, mentre non modifica in alcun modo il funzionamento di una rete che utilizza

la versione sicura del protocollo. Come anche riportato precedentemente nella sezione relativa all'implementazione di *SecureAodv* nel momento in cui un nodo *AodvAttackerRSA* forgia una RREP fasulla e questa arriva a destinazione (cioè al nodo che ha originato la RREQ) questo pacchetto viene automaticamente scartato, poiché il nodo è in grado di verificare che il pacchetto RREP da esso ricevuto presenta una firma fasulla, in quanto non è stato firmato mediante la chiave privata del nodo che originariamente doveva essere il reale destinatario. Questo tipo di attacco funziona bene con una versione del protocollo AODV che non implementa sicurezza in quanto il nodo che riceverà la RREP non ha alcun modo di verificare la firma di chi l'ha originato.

Capitolo 6

Simulazioni

All'interno di questo capitolo vengono illustrate in maniera dettagliata le esecuzioni delle simulazioni presenti nel progetto, in particolare sono state previste tre simulazioni:

- *Scenario 1*: presenta funzionamento di base del protocollo ADOV in assenza di nodi malevoli;
- *Scenario 2*: presenta funzionamento di base del protocollo AODV con presenza di nodi malevoli;
- *Scenario 3*: presenta funzionamento del protocollo Secure-AODV in assenza di nodi malevoli;
- *Scenario 4*: presenta funzionamento del protocollo Secure-AODV con presenza di nodi malevoli.

6.1 Scenario 1 - Funzionamento Aodv Standard

Questa simulazione prevede l'utilizzo da parte di tutti i nodi, di un router di tipo *AodvBase*.

6.1.1 Risultati empirici nodi

Di seguito è riportata la lista dei nodi presenti all'interno della rete simulata:

```
RoutingScenario (RoutingScenario) id=1
  radioMedium (ieee80211ScalarRadioMedium) id=2
  manager (VeinsInetManager) id=3
  visualizer (IntegratedVisualizer) id=4
  physicalEnvironment (PhysicalEnvironment) id=5
  roadsCanvasVisualizer (RoadsCanvasVisualizer) id=6
  canvas (cCanvas) 32 toplevel figure(s)
  node[0] (VeinsInetRoutingCar) id=37
  node[1] (VeinsInetRoutingCar) id=99
  node[2] (VeinsInetRoutingCar) id=161
  node[3] (VeinsInetRoutingCar) id=223
  node[4] (VeinsInetRoutingCar) id=285
  node[5] (VeinsInetRoutingCar) id=347
  node[6] (VeinsInetRoutingCar) id=409
  node[7] (VeinsInetRoutingCar) id=471
  node[8] (VeinsInetRoutingCar) id=533
  node[9] (VeinsInetRoutingCar) id=595
  node[10] (VeinsInetRoutingCar) id=657
  node[11] (VeinsInetRoutingCar) id=719
  node[12] (VeinsInetRoutingCar) id=781
  node[13] (VeinsInetRoutingCar) id=843
  node[14] (VeinsInetRoutingCar) id=905
  node[15] (VeinsInetRoutingCar) id=967
  node[16] (VeinsInetRoutingCar) id=1029
  node[17] (VeinsInetRoutingCar) id=1091
  node[18] (VeinsInetRoutingCar) id=1153
```

Figura 6.1: Nodi Scenario 1

Tutti i nodi della rete hanno la seguente struttura:

```
node[5] (VeinsInetRoutingCar) id=347
  radioIn[0] (cGate) not connected
  hasStatus (cPar) false
  osgModel (cPar) ""
  osgModelColor (cPar) ""
  canvasImage (cPar) ""
  canvasImageColor (cPar) ""
  numPcapRecorders (cPar) 0
  numLoInterfaces (cPar) 1
  numWlanInterfaces (cPar) 1
  numEthInterfaces (cPar) 0
  numPppInterfaces (cPar) 0
  numTunInterfaces (cPar) 0
  numVlanInterfaces (cPar) 0
  hasIpv4 (cPar) true
  hasIpv6 (cPar) false
  hasGn (cPar) false
  forwarding (cPar) true
  multicastForwarding (cPar) false
  hasUdp (cPar) true
  hasTcp (cPar) true
  hasSctp (cPar) false
  numApps (cPar) 1
  router (cPar) "AodvBase"
  canvas (cCanvas) 5 toplevel figure(s)
```

Figura 6.2: Nodo 0 - Scenario 1

6.1.2 Risultati empirici pacchetti

Di seguito vengono mostrati alcuni grafici che rappresentano, basandosi su una simulazione di 20 secondi, il *rate* di generazione dei pacchetti. In particolare il numero di pacchetti esistenti durante la simulazione è risultato essere:

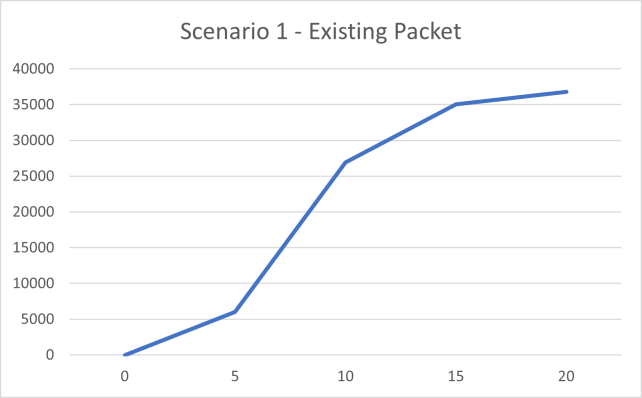

Figura 6.3: Pacchetti esistenti

Mentre il numero di pacchetti creati durante la simulazione:

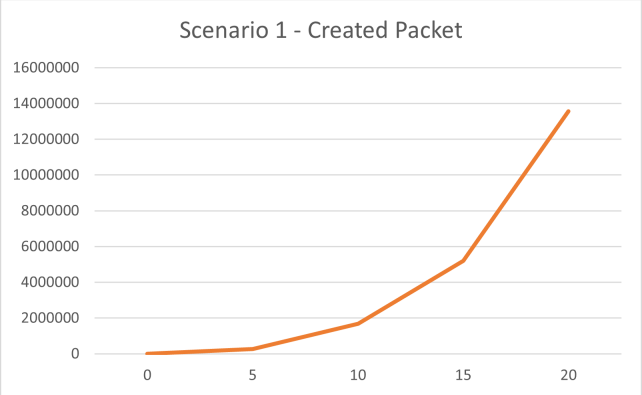

Figura 6.4: Pacchetti creati

6.2 Scenario 2 - Funzionamento Aodv con Attaccante

Questa simulazione prevede l'utilizzo da parte di tutti i nodi di un router di tipo *AodvBase*, e ci sono alcuni nodi della rete che implementano un router *AodvAttacker*, in particolare si tratta dei nodi: 2, 7 e 12. Di seguito è riportata una figura tratta dall'ambiente di simulazione offerto da OMNeT++:

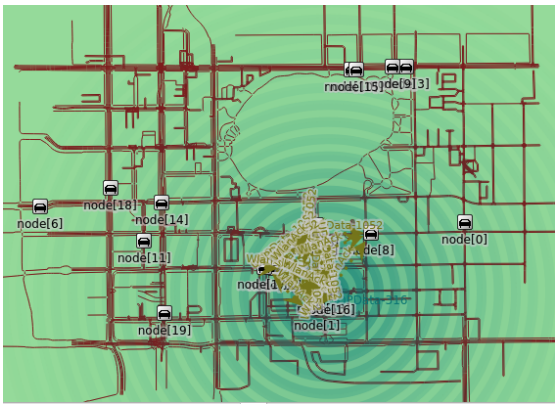

Figura 6.5: Simulazione Scenario 2

6.2.1 Risultati empirici nodi

Di seguito è riportata la lista dei nodi presenti all'interno della rete simulata:

- RoutingScenario (RoutingScenario) id=1
  - radioMedium (Ieee80211ScalarRadioMedium) id=2
  - manager (VeinsInetManager) id=3
  - visualizer (IntegratedVisualizer) id=4
  - physicalEnvironment (PhysicalEnvironment) id=5
  - roadsCanvasVisualizer (RoadsCanvasVisualizer) id=6
  - canvas (cCanvas) 32 toplevel figure(s)
  - node[0] (VeinsInetRoutingCar) id=37
  - node[1] (VeinsInetRoutingCar) id=99
  - node[19] (VeinsInetRoutingCar) id=1215
  - node[3] (VeinsInetRoutingCar) id=223
  - node[4] (VeinsInetRoutingCar) id=285
  - node[5] (VeinsInetRoutingCar) id=347
  - node[6] (VeinsInetRoutingCar) id=409
  - node[7] (VeinsInetRoutingCar) id=471
  - node[8] (VeinsInetRoutingCar) id=533
  - node[9] (VeinsInetRoutingCar) id=595
  - node[10] (VeinsInetRoutingCar) id=657
  - node[11] (VeinsInetRoutingCar) id=719
  - node[12] (VeinsInetRoutingCar) id=781
  - node[13] (VeinsInetRoutingCar) id=843
  - node[14] (VeinsInetRoutingCar) id=905
  - node[15] (VeinsInetRoutingCar) id=967
  - node[16] (VeinsInetRoutingCar) id=1029
  - node[17] (VeinsInetRoutingCar) id=1091
  - node[18] (VeinsInetRoutingCar) id=1153

Figura 6.6: Nodi Scenario 2

Analizzando maggiormente in dettaglio tali nodi è possibile verificare che il nodo 7 ad esempio si comporta da nodo malevolo:

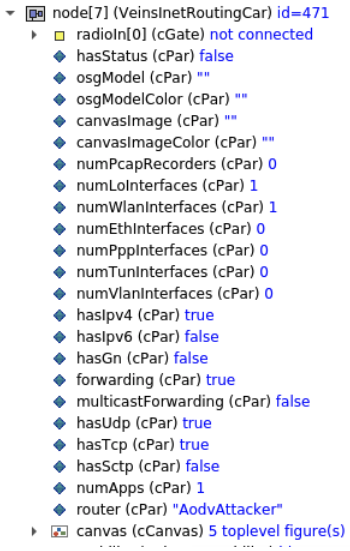

Figura 6.7: Nodo 7 - Scenario 2

Mentre gli altri nodi avranno la seguente struttura:

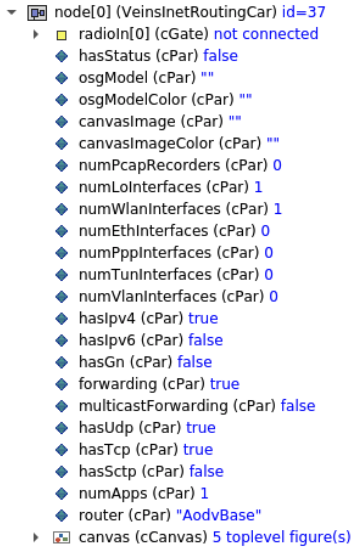

Figura 6.8: Nodo 0 - Scenario 2

6.2.2 Risultati empirici pacchetti

Di seguito vengono mostrati alcuni grafici che rappresentano, basandosi su una simulazione di 20 secondi, il *rate* di generazione dei pacchetti. In particolare il numero di pacchetti esistenti durante la simulazione è risultato essere:

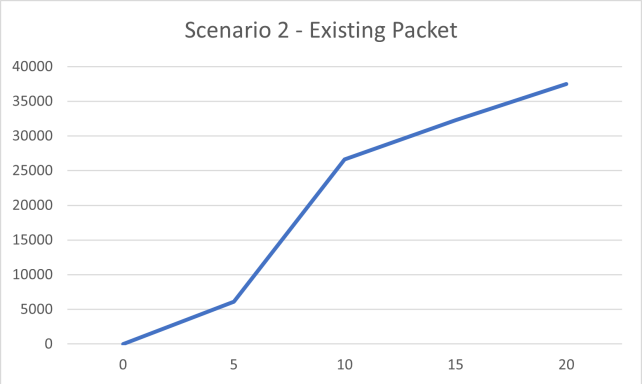

Figura 6.9: Pacchetti esistenti

Mentre il numero di pacchetti creati durante la simulazione:

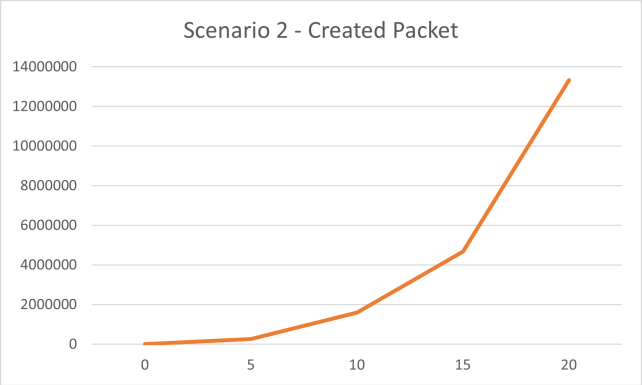

Figura 6.10: Pacchetti creati

6.3 Scenario 3 - Funzionamento Secure Aodv Standard

Questa simulazione prevede l'utilizzo da parte di tutti i nodi, di un router di tipo *SecureAodv*.

6.3.1 Risultati empirici nodi

Di seguito è riportata la lista dei nodi presenti all'interno della rete simulata:

Figura 6.11: Nodi Scenario 3

Tutti i nodi della rete hanno la seguente struttura:

Figura 6.12: Nodo 0 - Scenario 3

6.3.2 Risultati empirici pacchetti

Di seguito sono mostrati alcuni grafici che rappresentano, basandosi su una simulazione di 20 secondi, il *rate* di generazione dei pacchetti. In particolare il numero di pacchetti esistenti durante la simulazione è risultato essere:

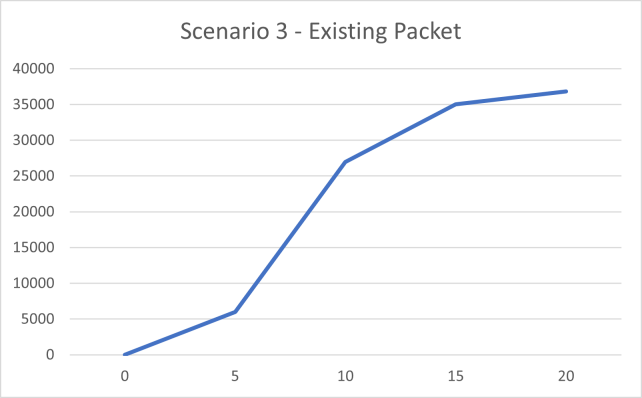

Figura 6.13: Pacchetti esistenti

Mentre il numero di pacchetti creati durante la simulazione:

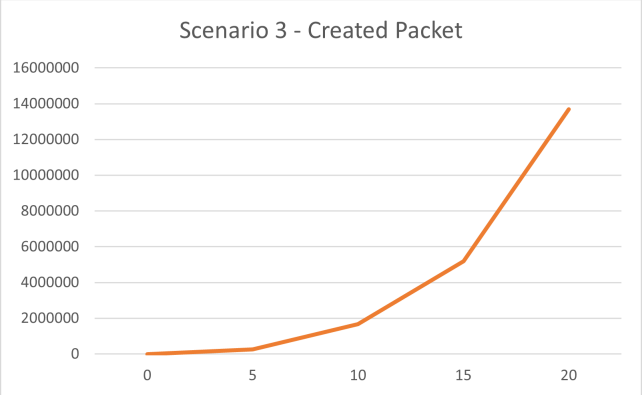

Figura 6.14: Pacchetti creati

6.4 Scenario 4 - Funzionamento Secure Aodv con Attaccante

Questa simulazione prevede l'utilizzo da parte di tutti i nodi di un router di tipo *SecureAodv*, e ci sono alcuni nodi della rete che implementano un router *AodvAttackerRSA*, in particolare si tratta dei nodi: 2, 7 e 12.

6.4.1 Risultati empirici nodi

Di seguito è riportata la lista dei nodi presenti all'interno della rete simulata:

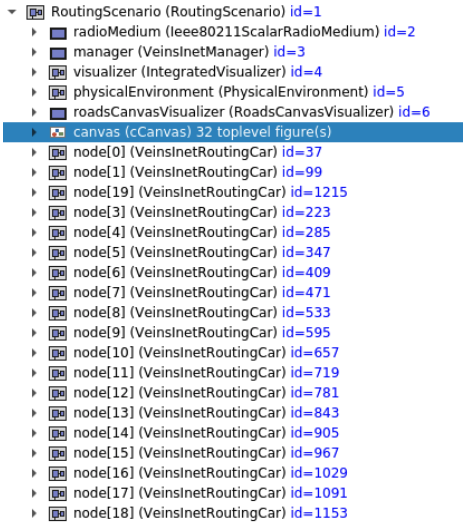

Figura 6.15: Nodi Scenario 4

Analizzando maggiormente in dettaglio tali nodi è possibile verificare che il nodo 7 ad esempio si comporta da nodo malevolo:

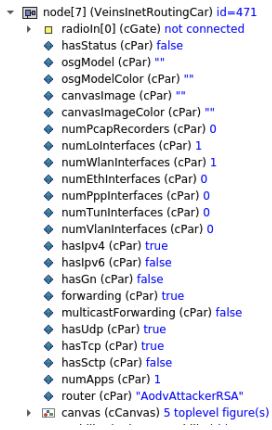

Figura 6.16: Nodo 7 - Scenario 4

Mentre gli altri nodi avranno la seguente struttura:

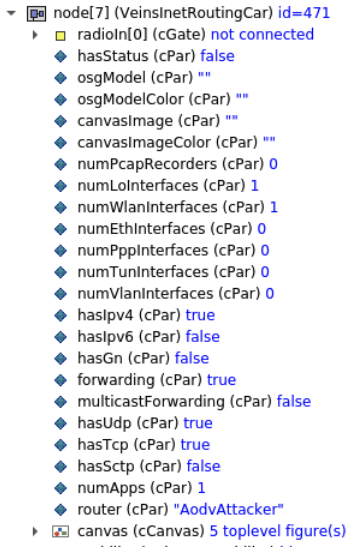

Figura 6.17: Nodo 0 - Scenario 4

6.4.2 Risultati empirici pacchetti

Di seguito sono mostrati alcuni grafici che rappresentano, basandosi su una simulazione di 20 secondi, il *rate* di generazione dei pacchetti. In particolare il numero di pacchetti esistenti durante la simulazione è risultato essere:

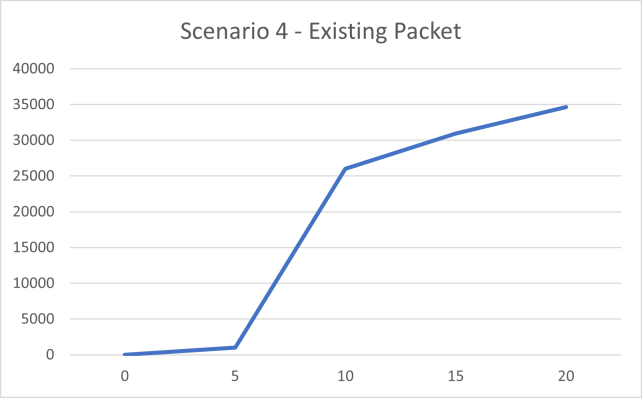

Figura 6.18: Pacchetti esistenti

Mentre il numero di pacchetti creati durante la simulazione:

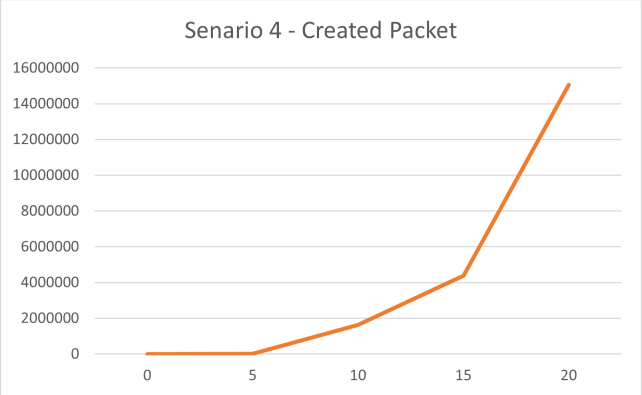

Figura 6.19: Pacchetti creati

6.5 Osservazioni Finali

Come si è potuto evincere dagli Scenari 1 e 3, relativi alle simulazioni in condizioni di funzionamento normale (in assenza di nodi malevoli), i risultati sono pressoché identici. Nello specifico si nota come il numero di messaggi esistenti al termine delle simulazioni sia pari in entrambi i casi a 36.805 e analogamente anche il numero di pacchetti creati sia lo stesso, ossia 13.576.126. Questo ci dà conferma che il comportamento dei due protocolli sia il medesimo e che l'aggiunta dello strato di sicurezza non vada ad inficiare sulle prestazioni della rete. Per quanto riguarda lo Scenario 2 i risultati sono differenti, poiché il numero di messaggi esistenti è pari a 37.515, mentre il numero di pacchetti creati è pari a 13.322.475. Si nota come la presenza dei tre nodi malevoli causi una lieve diminuzione del numero di pacchetti creati, ma al contempo è presente un maggior numero di pacchetti esistenti. La spiegazione a questo fenomeno risiede nel fatto che essendo questi nodi, dei nodi malevoli che effettuano un attacco di tipo DoS sulla rete, non permettono ai restanti nodi di comunicare verso i nodi che in realtà vorrebbero raggiungere. Contemporaneamente il numero maggiore di pacchetti esistenti è giustificato dal fatto che questi nodi malevoli forgiano dei pacchetti di tipo RREP che raggiungono la destinazione, ma non vengono *droppati* poiché i nodi non sono in grado di verificare la natura malevola del nodo che ha originato il pacchetto. Infine analizzando lo Scenario 4 si osservano le performance che offre l'utilizzo si Secure Aodv in ambiente in cui vi è la presenza di nodi malevoli. É possibile notare che è presente un minor numero di pacchetti esistenti (34.627) e un maggior numero di pacchetti creati (15.050.528), rispetto allo Scenario 3, in cui non c'è presenza di attaccanti. La spiegazione a questo fenomeno è giustificata dal fatto che i nodi malevoli forgeranno pacchetti RREP fasulli con il fine di inviarli ai nodi mittenti dei pacchetti RREQ. Arrivati a destinazione però, questi pacchetti vengono *droppati* proprio grazie alle verifiche di sicurezza che vengono attuate sia sul campo *signature* che sul campo *hash* del pacchetto.

Capitolo 7

Codice sorgente progetto

Di seguito è possibile prendere visione del codice sorgente del progetto sviluppato:

Figura 7.1: Link to GitHub Repository