Università degli Studi di Padova

Facoltà di Ingegneria

Studio ed Implementazione di Tecniche di Sicurezza Informatica per la Riprogrammazione di Reti di Sensori Radio

Relatore:
Ch.mo Prof. Michele Rossi

Laureando:
Moreno Dissegna

Corso di Laurea Specialistica in Ingegneria Informatica

Anno Accademico 2009/2010

Desidero ringraziare tutti coloro che mi hanno aiutato durante lo svolgimento di questa tesi di laurea, innanzitutto il prof. Michele Rossi per la disponibilità ed i consigli offertimi. Inoltre, ringrazio Nicola Bui per il suo costante sostegno e per le numerose ore dedicatemi. Intendo poi ringraziare Cristiano Tapparello e Riccardo Crepaldi per l'aiuto offerto durante l'avvio di questo progetto e Angelo Castellani per il suo supporto nell'utilizzo del testbed. Ringrazio inoltre Osman Ugus per le sue consulenze tecniche nel'ambito della sicurezza. Infine, desidero ringraziare Luca, Marco e Francesco per i consigli, l'aiuto ed il sostegno che mi hanno dato in questi nove mesi di lavoro e la mia famiglia, i miei amici e tutte le persone che mi sono state vicine in questi anni di studio.

Indice

Introduzione

Una rete di sensori wireless (Wireless Sensor Network, WSN) consiste in un insieme di nodi, detti mote, ognuno dei quali è equipaggiato con uno o più sensori, un sistema di comunicazione radio, un'unità di elaborazione e una memoria dove memorizzare temporaneamente i dati provenienti dai sensori. I sensori di un nodo misurano fenomeni quali eventi sismici, termici, ottici e acustici, mentre gli altri componenti processano i dati rilevati, e solitamente li trasmettono ad un punto di raccolta.

Il fattore principale che rende le WSN promettenti sia dal punto di vista economico, che dal punto di vista della ricerca, è il fatto che reti di dimensioni molto grandi possono essere installate senza imbattersi nelle barriere fisiche a cui sono soggetti i sistemi di tipo wired. Un altro aspetto da non sottovalutare è la possibilità di intervenire rapidamente sulla loro disposizione: i sensori infatti non essendo vincolati a nulla possono essere semplicemente spostati. Il basso costo e l'assenza del cablaggio ne rende possibile una fitta distribuzione, fornendo una quantità di dati superiore rispetti ai sistemi tradizionali.

In tali reti gli aggiornamenti del software sono di fatto essenziali per riconfigurare il sistema in seguito, ad esempio, a correzioni di bug o aggiunta di funzionalità alle applicazioni esistenti. La riprogrammazione manuale di ogni singolo nodo è impensabile quando la rete si estende oltre qualche decina di nodi, in quanto i costi e i tempi di riprogrammazione sarebbero

troppo elevati, rendendo di fatto più conveniente l'utilizzo di una rete cablata. La possibilità di riprogrammare la rete in modo wireless è quindi un servizio molto importante in una WSN. Questo servizio deve essere affidabile, efficiente dal punto di vista energetico, scalabile, e il più possibile rapido.

Motivazione

La possibilità di riprogrammare la rete in modo *wireless* pur essendo da un lato di fatto necessaria, dall'altro introduce problematiche nuove rispetto alla riprogrammazione manuale o a quella cablata. Oltre a problematiche tecniche dovute alla comunicazione broadcast che avviene nel canale radio e alle limitazioni energetiche, si pone anche una questione relativa alla sicurezza dell'operazione. È infatti di vitale importanza che la riprogrammazione della rete sia possibile solo da parte di individui autorizzati e mentre nel caso di reti cablate ottenere l'accesso alla rete è complicato, dovendo in qualche modo manipolare il sistema di cablatura, nelle reti wireless questo problema non si pone. È sufficiente infatti disporre di un dispositivo radio nelle vicinanze della rete per poter comunicare con i nodi.

In tutte le applicazioni necessarie in una WSN e di conseguenza anche nella riprogrammazione, è necessario quindi affrontare tematiche relative alla sicurezza. La riprogrammazione inoltre è un'applicazione assai sensibile da questo punto di vista, in quanto una falla nei sistemi di sicurezza può consentire ad un individuo malintenzionato (denominato in genere *avversario*) di prendere il controllo della rete, e magari di farlo addirittura senza essere notato.

È importante quindi prevedere innanzitutto un sistema di autenticazione per far si che solo individui autorizzati possano riprogrammare la rete. Un altra caratteristica desiderabile è inoltre la confidenzialità, che consiste nel

far si che origliando il canale radio un avversario non riesca ad ottenere informazioni utili relativamente all'applicazione che si sta disseminando. Infine è importante anche la protezione da attacchi di tipo Denial of Service (DoS), che sfruttando caratteristiche del protocollo di riprogrammazione possono impedire o rallentare il processo di riprogrammazione della rete.

Contributo dell'autore

L'obiettivo di questa tesi è quindi lo studio e lo sviluppo di alcune tecniche di sicurezza che garantiscono le suddette protezioni durante il processo di riprogrammazione wireless di una rete di sensori. Verranno discussi alcuni sistemi di firma digitale e di cifratura e la loro applicabilità nel contesto delle reti di sensori, e verrà presentato un algoritmo di firma digitale progettato per l'utilizzo in dispositivi dalle risorse limitate. Verranno quindi presentati i sistemi di sicurezza implementati, dei quali verrà valutato il livello di sicurezza garantito e successivamente misurato l'impatto sulla performance.

Organizzazione della tesi

Il resto della tesi è organizzato come segue. Nel capitolo 1 sono introdotte le reti di sensori wireless e alcune loro applicazioni. Nel capitolo 2 sono presentate le tecnologie sulle quali questo lavoro è stato fondato: la piattaforma hardware TMoteSky/TelosB, il sistema operativo TinyOS e il sistema di riprogrammazione wireless Synapse++ . Nel capitolo 3 sono discusse le tematiche relative agli algoritmi di sicurezza: i sistemi di firma digitale e la loro applicabilità nelle reti di sensori wireless, l'algoritmo di firma digitale TTimeSA , lo standard di cifratura AES. Nel capitolo 4 viene illustrato il lavoro svolto, mentre una breve discussione che conclude questa tesi è presentata nel capitolo 5.

Capitolo 1

Reti di sensori wireless

Una rete di sensori wireless (Wireless Sensor Network, WSN) consiste in un insieme di nodi, detti mote, ognuno dei quali è equipaggiato con uno o più sensori, un sistema di comunicazione radio, una semplice unità di elaborazione e una memoria dove memorizzare temporaneamente i dati provenienti dai sensori. I sensori di un nodo misurano fenomeni quali eventi sismici, termici, ottici e acustici, mentre gli altri componenti processano i dati rilevati, e solitamente li trasmettono ad un punto di raccolta. Il fattore principale che rende le WSN promettenti sia dal punto di vista economico, che dal punto di vista della ricerca, è il fatto che reti di dimensioni molto grandi possono essere installate senza imbattersi nelle barriere fisiche a cui sono soggetti i sistemi di tipo cablato. Infatti una caratteristica di queste reti è il fatto di essere "ad hoc", il che significa che non c'è nessuna topologia o gerarchia di rete definita a priori. Questo fatto, unito alla possibilità di utilizzare sistemi di comunicazioni wireless, apre scenari che non sarebbero altrimenti stati possibili. In questo capitolo vengono presentate alcune applicazioni delle reti di sensori wireless.

1.1 Caratteristiche

La maggior parte delle reti di sensori utilizzate oggi impiega un piccolo numero di sensori collegati ad un unità centrale cui spetta il compito di elaborare i segnali, le reti di sensori wireless invece sono di tipo distribuito, cioè permettono di demandare ai nodi stessi parte del lavoro di elaborazione. Inoltre l'uso di sensori wireless con un consumo di energia limitato risulta essere spesso una necessità quando la zona da monitorare risulta priva delle infrastrutture quali le sorgenti energetiche, necessarie ad una rete cablata, e risulta molto conveniente quando la posizione precisa di un segnale di interesse è sconosciuta in una certa regione, in quanto l'uso di sensori wireless e a basso costo ne permette una distribuzione più fitta. Grazie a queste caratteristiche le applicazioni delle WSN sono svariate.

1.2 Applicazioni

Le applicazioni delle reti di sensori wireless sono svariate, di seguito ne sono presentate alcune tra le più recenti, divise per categoria.

1.2.1 Monitoraggio ambientale

Un esempio di applicazione di monitoraggio ambientale è costituita dal progetto Volcano Sensorweb [1, 2], sviluppato dalla NASA, nel quale sono stati posizionati dei sensori all'interno e attorno a diversi vulcani, l'ultimo dei quali il Monte Sant'Elena, uno dei vulcani più attivi degli Stati Uniti, nel luglio 2009. L'ultima attività vulcanica risale al 2004 e non ha provocato danni rilevanti, mentre nel 1980 un'esplosione causò molte vittime e danni. Ogni nodo contiene un sismometro, un ricevitore GPS per rilevare la propria posizione e misurare anche sottili deformazioni del terreno, un sistema ad infrarossi per rilevare esplosioni vulcaniche e un sensore di luce per mon-

itorare la formazione di nubi di cenere. I nodi sono alimentati a batterie che hanno una durata minima di un anno. I nodi comunicano tra loro e con il satellite Earth Observing-1 (EO-1), al quale possono inviare automaticamente richieste di acquisizione di immagini ad alta risoluzione. Come è prerogativa delle reti di sensori wireless il deployment della rete è consistito solamente nel posizionamento dei nodi nel luogo desiderato, in questo caso calandoli nei luoghi di interesse con un elicottero, senza quindi necessità di un intervento umano a terra. Questa rete potrà essere utilizzata un giorno per rispondere rapidamente ad una eruzione imminente, ma il progetto prevede anche la possibilità di utilizzo per l'esplorazione di altri pianeti del sistema solare.

1.2.2 Monitoraggio di velivoli

GE Aviation sta sviluppando un sistema wireless di acquisizione e trasmissione dati per applicazioni aeronautiche nell'ambito dell'iniziativa WiTNESSS (WIreless Technologies for Novel Enhancement of Systems and Structures Serviceability)[3]. Il trasferimento di dati è essenziale per le applicazioni di monitoraggio strutturale dei velivoli e per i test della strumentazione di bordo. I potenziali vantaggi del trasferimento wireless dei dati per queste applicazioni sono variegati, e includono: un consistente risparmio di peso, integrazione semplificata (non essendoci cavi da stendere) e un accesso semplificato ai dati. Inoltre l'assenza di dati necessari al controllo del volo in queste applicazioni le rende un campo di prova ideale per la tecnologia wireless.

1.2.3 Monitoraggio di edifici

L'azienda italiana Lachesi partecipa invece al Progetto Guarini[4], un programma multidisciplinare mirato allo sviluppo di un sistema di monitorag-

gio strutturale della Cappella della Sacra Sindone del Guarini. Patrocinato dalla Soprintendenza ai Beni Culturali, il Progetto Guarini è un esempio di monitoraggio strumentale della sicurezza e dell'integrità di un manufatto di estremo valore architettonico e religioso. Progetto Guarini è iniziato nel 2007 e il suo orizzonte temporale è stimato in 4 anni. Obiettivo del progetto è l'installazione di uno sistema diagnostico integrato finalizzato al monitoraggio strutturale permanente della Cappella della Sacra Sindone del Guarini. Scopo finale di Progetto Guarini è la valutazione, in continuo, della sicurezza e dello stato dell'opera, in relazione alla sua integrità proiettata nel tempo, e la definizione delle modalità di intervento per la salvaguardia del prezioso manufatto fortemente danneggiato dall'incendio occorso nel 1997.

1.2.4 Monitoraggio del traffico

Sensys networks commercia un sistema di monitoraggio del traffico[5] che sfrutta sensori wireless posti sotto il suolo stradale che comunicano con un access point situato in un luogo accessibile, e rendono possibile un monitoraggio molto accurato del passaggio dei veicoli. Questi sistemi sono utilizzati quindi per il controllo della semaforica agli incroci, per il rilevamento del traffico autostradale, e con un sistema non invasivo della privacy anche per monitorare i tempi di percorrenza urbani. Questo sistema in particolare sfrutta una sorta di "impronta magnetica" univoca per ogni autovettura per tracciare il tempo impiegato da ogni singola vettura per seguire il proprio percorso all'interno dell'area monitorata, e consente quindi una stima basata su dati reali dei tempi di percorrenza per andare da un punto ad un altro della città.

1.2.5 Applicazioni medicali

Un esempio di applicazione medicale è costituito dal "cerotto digitale" Sensium[6], sviluppato da Toumaz Technology, in fase di sperimentazione dal novembre 2009. Questa tecnologia permette il monitoraggio dei parametri vitali di un paziente 24 ore su 24, con un'autonomia di una settimana. L'uso di sensori wireless invece di sensori cablati da un lato rende il monitoraggio molto meno invasivo per il paziente e dall'altro facilita il processo di acquisizione dei dati stessi, che può avvenire in modo continuo via wireless invece che tramite un dispositivo di memorizzazione indossato dal paziente.

Capitolo 2

Tecnologia utilizzata

Per la comprensione del lavoro svolto sono necessarie alcune conoscenze sulle piattaforme hardware e software utilizzate.

La piattaforma hardware utilizzata è la piattaforma TMoteSky. È dotata di un microcontrollore a 16 bit con 10KB di RAM, di sensori di luce, temperatura e umidità, di una memoria flash da 1MB, e di un chip di comunicazione radio.

La piattaforma software è costituita dal sistema operativo TinyOS e dal software su cui è basato il lavoro svolto, Synapse++ .

TinyOS è un sistema operativo per reti di sensori wireless con architettura a componenti. È progettato in modo da consentire la portabilità delle applicazioni attraverso diverse piattaforme, grazie ad un'opportuna astrazione delle risorse hardware.

Synapse++ è un software di riprogrammazione wireless che implementa un protocollo di disseminazione basato su rateless codes, per avere la massima efficenza nel recupero degli errori durante trasmissioni broadcast.

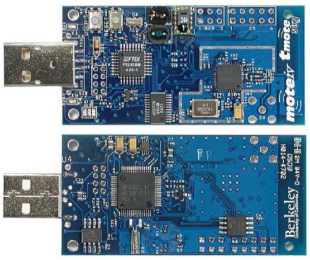


Immagine 2.1: Piattaforma TMote Sky

2.1 Piattaforma hardware

La piattaforma hardware utilizzata in questo lavoro è la piattaforma open source TMoteSky/TelosB. TMoteSky è il nome con cui era commercializzata da Sentilla, mentre TelosB è il nome di quella commercializzata tuttora da Crossbow. Il design di entrambe le piattaforme è comunque identico, e il software è completamente portabile tra le due.

Le caratteristiche principali di questa piattaforma sono:

- Microcontrollore TI MSP430, clock a 1MHz, 10KB di RAM integrata e 48KB di memoria flash per il codice eseguibile
- 1MB di memoria flash
- Sensori opzionali di luce, temperatura, umidità e pressione atmosferica
- Chip radio CC2420 compatibile con lo standard 802.15.4 e operante sulla banda ISM 2.4 - 2.4835 GHz
- Programmazione e comunicazione seriale tramite interfaccia USB
- Basso consumo energetico

In questo lavoro sono state sfruttate alcune funzionalità offerte dal chip radio, mentre altre sue caratteristiche costituiscono delle limitazioni. Entrambi questi punti verranno approfonditi nella prossima sezione.

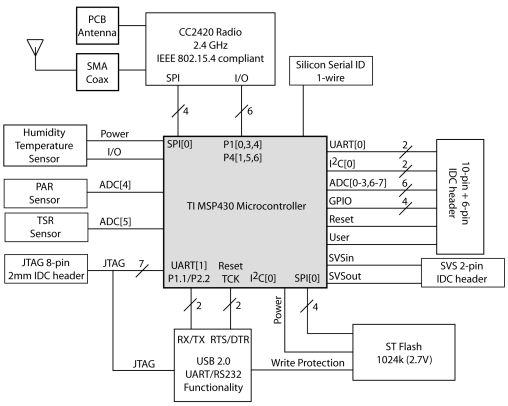


Immagine 2.2: Schema funzionale a blocchi della piattaforma TMoteSky

2.1.1 Caratteristiche del CC2420

Il chip radio utilizzato nella piattaforma hardware offre oltre alle funzioni di trasmissione e ricezione, anche una serie di primitive di sicurezza implementate in hardware. Queste primitive sono: counter mode (CTR) encryption/decryption, CBC-MAC authentication e CCM encryption/authentication. Tutte queste operazioni sono basate sulla cifratura AES [7] e utilizzano chiavi a 128 bit. Istruzioni approfondite sull'utilizzo di queste funzionalità si possono trovare in [8]. Ulteriori dettagli su CBC-MAC e AES sono disponibili nelle sezioni 3.4.1 e 3.4.3

In questo lavoro è stata sfruttata la primitiva CBC-MAC authentication, per implementare il sistema di protezione da Denial of Service. Quando viene utilizzata questa funzionalità il chip radio calcola per ogni pacchetto trasmesso un Message Authentication Code (MAC) che viene scritto alla fine dello stesso. Per calcolare il MAC viene utilizzata una chiave a 128 bit che deve essere impostata dall'esterno, modificando appositi registri di configurazione. Al momento della ricezione del pacchetto, spetta al ricevitore capire in base al contenuto del pacchetto quali siano le operazioni da effettuare. Grazie quindi ad alcuni campi di controllo inseriti nei pacchetti trasmessi il ricevitore è in gradi di capire se il pacchetto è autenticato e di

conseguenza verificarne l'autenticità. Questo viene fatto impostando prima di tutto la chiave, che deve essere uguale a quella usata dal trasmettitore, e successivamente inviando al chip il comando di decodifica, che scriverà il byte `0x00` come ultimo byte del pacchetto nel caso in cui la verifica abbia dato esito positivo, `0xFF` altrimenti. La presenza di questa funzionalità implementata direttamente in hardware è molto conveniente in quanto consente sia di risparmiare spazio per il codice eseguibile, evitando la necessità di implementare l'algoritmo di cifratura in software, che di risparmiare tempo, in quanto l'esecuzione in hardware dell'algoritmo di cifratura è di vari ordini di grandezza più veloce di quella software.

Un'altra funzionalità offerta dal CC2420 e utilizzata in questo lavoro è la possibilità di eseguire una cifratura stand-alone, cioè di cifrare dei dati senza doverli inserire nel buffer di trasmissione. Questa funzionalità è stata sfruttata per implementare gli algoritmi di decifratura e di hashing, in modo da minimizzare l'occupazione in ROM del codice relativo. Purtroppo però nonostante l'esecuzione in hardware delle operazioni sia di per sé molto veloce, in questo caso vi è un'altra limitazione hardware che annulla completamente questo vantaggio, rendendo di fatto almeno l'operazione di hashing che utilizza il CC2420 più onerosa rispetto ad un'implementazione software alternativa. Questa limitazione si può osservare nel diagramma 2.2 ed è rappresentata dal bus di comunicazione tra il microcontrollore ed il CC2420. Si tratta di un bus SPI, che ha pertanto capacità di 1 bit in ciascuna direzione. Attraverso questo bus quindi è possibile comunicare al massimo ad un bit per ciclo di clock, ma la velocità effettiva è minore e risulta essere di 19μs per byte. Pertanto in questo caso la scelta di utilizzo di questa funzionalità è stata guidata dalla priorità data al risparmio di codice rispetto al tempo di esecuzione, che risulta leggermente maggiore di un'implementazione software.

2.2 TinyOS

TinyOS è un sistema operativo open source progettato per reti di sensori wireless. Il ruolo di un sistema operativo è rendere possibile lo sviluppo di applicazioni affidabili mettendo a disposizione tramite delle interfacce di programmazione un'adeguata astrazione delle risorse hardware.

La divisione tra sistema operativo e applicazioni è poco comune nei sistemi embedded, dove le applicazioni sono strettamente legate all'hardware su cui vengono eseguite, e dove è necessario porre attenzione, oltre che alla logica dell'applicazione, anche al corretto ed efficiente utilizzo delle limitate risorse hardware disponibili.

TinyOS è stato progettato appositamente per le reti di sensori wireless. È caratterizzato da un'architettura software basata sui componenti, che permette di sviluppare applicazioni robuste, di minimizzare il consumo energetico e di facilitare lo sviluppo di algoritmi e protocolli sofisticati, garantendo allo stesso tempo un alto grado di concorrenza e generando un codice di dimensioni ridotte.

TinyOS mette a disposizione una libreria contenente numerosi componenti e interfacce, che possono essere connessi allo stesso modo con cui si connettono dei moduli hardware, che permettono di gestire le diverse piattaforme hardware semplicemente collegando tra loro i componenti necessari.

2.2.1 Caratteristiche generali

TinyOS è caratterizzato da un'architettura basata sui componenti, che permette un rapido sviluppo ed una rapida implementazione di applicazioni, minimizzando allo stesso tempo la dimensione del codice, come richiesto dalle forti limitazioni di memoria presenti nelle reti di sensori.

I componenti (Figura 2.3) sono oggetti ben delimitati, con delle inter-

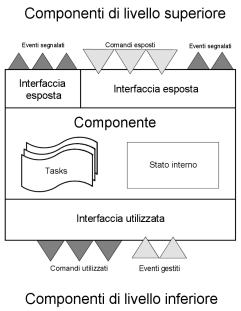


Immagine 2.3: Struttura di un generico componente di TinyOS

facce definite, uno stato interno e una gestione interna della concorrenza. Nella libreria di TinyOS sono presenti degli oggetti basilari che incapsulano gli elementi hardware come la radio, gli ADC, i timer e così via. Le loro interfacce rispecchiano le operazioni hardware e gli interrupt, mentre lo stato e il parallelismo sono gli stessi del dispositivo. Componenti di livello più alto incapsulano funzionalità più astratte, ma con lo stesso principio di funzionamento, mettendo a disposizione comandi ed eventi, mantenendo un proprio stato e disponendo di una gestione della concorrenza tramite task. L'interazione tra task è possibile solo all'interno di un singolo componente, mentre le interazioni tra componenti avvengono solamente attraverso i comandi e gli eventi esposti dalle loro interfacce. Ogni componente di TinyOS utilizza e implementa un insieme di interfacce bidirezionali, che mettono a disposizione comandi e richiedono gestione di eventi. Le interfacce che il componente dichiara di implementare possono essere implementate direttamente dal componente, oppure possono essere implementate collegando tra loro un insieme di sotto-componenti ed esponendone le interfacce di interesse. Questa architettura modulare garantisce flessibilità, robustezza e facilità di programmazione.

In un sistema operante con TinyOS il software in esecuzione è costituito da un' unica applicazione, costituita sia da codice del sistema operativo che da codice dei componenti specifici dell'applicazione, come si può vedere in

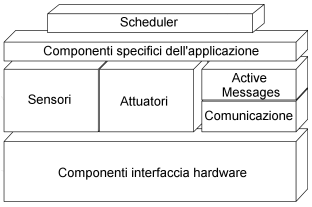


Immagine 2.4: Architettura software di una applicazione basata su TinyOS

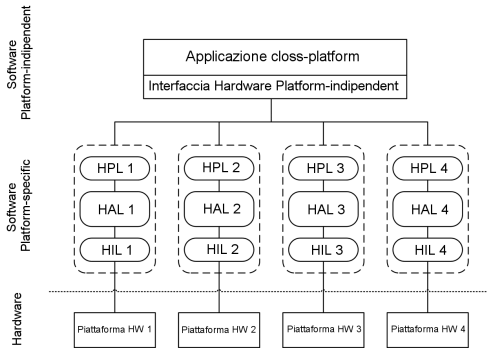


Immagine 2.5: Struttura che permette la portabilità cross-platform di un' applicazione basata su TinyOS

figura 2.4.

Alla base della struttura (Figura 2.5) vi è il livello di interfacciamento con l'hardware, costituito da una serie di componenti specifici di ogni piattaforma che permettono ai componenti dei livelli superiori di avere una visione uniforme e sufficientemente astratta dei componenti hardware presenti nel sistema. Questo livello espone ai livelli superiori un'interfaccia indipendente dalla piattaforma hardware sottostante, e rende così possibile la migrazione del sistema da una piattaforma ad un'altra sostituendo solamente i componenti di questo livello.

Le caratteristiche fondamentali di TinyOS sono descritte approfonditamente in [9]. In seguito vengono riportati e approfonditi alcuni concetti la cui comprensione si è rivelata necessaria o utile durante lo sviluppo di questo lavoro.

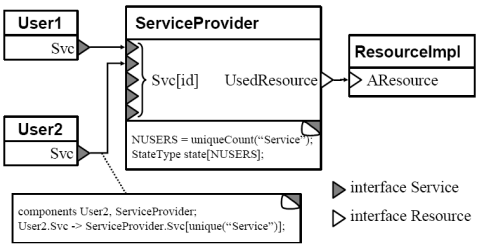


Immagine 2.6: Struttura del design pattern Service Instance

2.2.2 Service Instance design pattern

Questo design pattern, i cui dettagli sono disponibili in [9], è importante per la comprensione del funzionamento di vari componenti di TinyOS, dallo stack radio alla gestione dello storage. Questo pattern è stato inoltre adottato nello sviluppo di alcuni componenti utilizzati in questo lavoro.

Questo pattern viene applicato quando si vuole mettere a disposizione di un numero di componenti client, noto a compile-time, un determinato servizio. È importante conoscere questo pattern per evitare un'erronea scelta di design quale quella di utilizzare componenti generici per offrire istanze multiple del servizio. Utilizzando componenti generici infatti il codice viene duplicato per ogni istanza del componente, mentre utilizzando lo schema Service Instance il codice viene allocato solamente una volta, ed è solo lo spazio di memoria necessario a mantenere lo stato dei vari client ad essere allocato in base al numero effettivo di client presenti. L'uso di componenti generici va invece ristretto nei limiti del possibile ai casi in cui sia necessario parametrizzare in modo differente le interfacce esposte dal componente per i vari client. La motivazione per cui questo viene fatto è il tentativo continuo di minimizzare la dimensione del codice di ogni componente, in quanto lo spazio in ROM è senz'altro una delle risorse più limitanti in applicazioni complesse.

In figura 2.6 si notano le due interfacce interessate da questo pattern:

l'interfaccia Service e l'interfaccia Resource. L' interfaccia Service è quella utilizzata dai vari client, che vi accedono in modo indipendente e senza bisogno di coordinazione esplicita. L'interfaccia Resource è invece utilizzata dal componente ServiceProvider per fornire il servizio. Sarà quindi ServiceProvider a gestire il coordinamento nell'accesso alla risorsa da parte dei suoi client.

Un esempio basilare di utilizzo di questo pattern è costituito dal componente `AMQueueImplP` che gestisce e coordina l'accesso da parte di diversi client alla risorsa radio. Questa è la dichiarazione delle interfacce esposte dal modulo:

```
generic module AMQueueImplP(int numClients) @safe() {
    provides interface Send[uint8_t client];
    uses{
        interface AMSend[am_id_t id];
        interface AMPacket;
        interface Packet;
    }
}
```

Listato 1: AMQueueImplP: un esempio di componente che ricopre il ruolo di ServiceProvider nel pattern Service Instance

L'interfaccia `Send` costituisce l'interfaccia `Service` del pattern, mentre le altre interfacce sono utilizzate per fornire il servizio e costituiscono quindi l'interfaccia `Resource`. È da notare che il componente è generico in quanto gli sviluppatori hanno deciso di dare la possibilità di avere più code indipendenti, per esempio nella piattaforma telosb si avrà una coda per la comunicazione radio e una per la comunicazione seriale. Ma anche in questo caso, indipendentemente dal numero di client che utilizzano la risorsa, saranno generate solo due istanze del componente.

La determinazione del numero di client associati al componente è affidata come da prassi alle funzioni `unique` e `uniqueCount`. Ogni client che

necessiti di utilizzare la risorsa radio per spedire pacchetti alloca un' istanza del componente `AMSenderC` che va a richiamare `unique(UQ_AMQUEUE_SEND)`, mentre il componente `AMQueueImplP` viene allocato tramite la chiamata `new AMQueueImplP( uniqueCount( UQ_AMQUEUE_SEND ))`.

Analizzando l'implementazione del componente è possibile vedere invece come sia la RAM ad essere allocata in funzione del numero di client associati al componente.

```
...
implementation {
    typedef struct {
        message_t* msg;
    } queue_entry_t;

    uint8_t current = numClients; // mark as empty
    queue_entry_t queue[numClients];
    uint8_t cancelMask[numClients/8 + 1];
...
```

Listato 2: AMQueueImplP: caratteristiche comuni nell'implementazione del ServiceProvider

Come accade sempre quando questo pattern viene utilizzato, viene definito un tipo di dati, in questo caso `queue_entry_t`, che rappresenta l'informazione che è necessario mantenere per ogni client, ed un array di dimensione pari al numero di client che mantiene questa informazione. Nel nostro caso gli array sono due: `queue` e `cancelMask`.

In questo modo quindi viene gestita la virtualizzazione delle risorse che consentono la comunicazione radio e seriale, minimizzando il codice generato e allocando RAM esattamente nella quantità necessaria per gestire il numero di client connessi al componente.

2.2.3 Packet level time synchronization

L'istante in cui avvengono determinati eventi è spesso di interesse in una rete di sensori, ma il mantenimento di un tempo globale in una rete può comportare un significativo overhead di comunicazione che può non sempre essere necessario per l'applicazione.

In TinyOS è implementato un sistema di sincronizzazione tra nodi che garantisce la sincronizzazione sul singolo hop. Quello che si ottiene è la capacità di un nodo di comunicare ad un altro nodo l'istante di tempo globale in cui un evento locale è avvenuto.

Questo non è possibile trasmettendo solamente il valore che timer locale aveva al momento dell'evento, in quanto i timer interni dei vari nodi della rete non sono sincronizzati. Si trasmette invece la differenza tra l'istante in cui il pacchetto viene spedito e l'istante in cui l'evento era accaduto. Questa differenza non è più un valore valido solo localmente, ma globalmente, in quanto non indica più un istante di tempo indicato da un timer locale, ma indica invece un intervallo di tempo, che è valido globalmente*. Un esempio di questo sistema è in figura 2.7.

Questo sistema non è limitato alla sincronizzazione tra due nodi solamente, ma in generale ogni nodo che riceva il pacchetto si può sincronizzare con il mittente di quel pacchetto.

In TinyOS il sistema di sincronizzazione viene implementato nel componente `TimeSyncMessageC` che espone l'interfaccia `TimeSyncAMSendMilli` e l'interfaccia `TimeSyncAMSend32Khz` per inviare pacchetti contenenti le informazioni per la sincronizzazione, sfruttando il timer a 1KHz o a 32KHz rispettivamente, e le interfacce `TimeSyncPacketMilli` e `TimeSyncPacket32khz` per estrarre dal pacchetto ricevuto l'istante di tempo espresso rispetto al timer locale.

*Viene considerata trascurabile in questo caso l'eventuale differenza di frequenza tra i timer dei vari nodi.

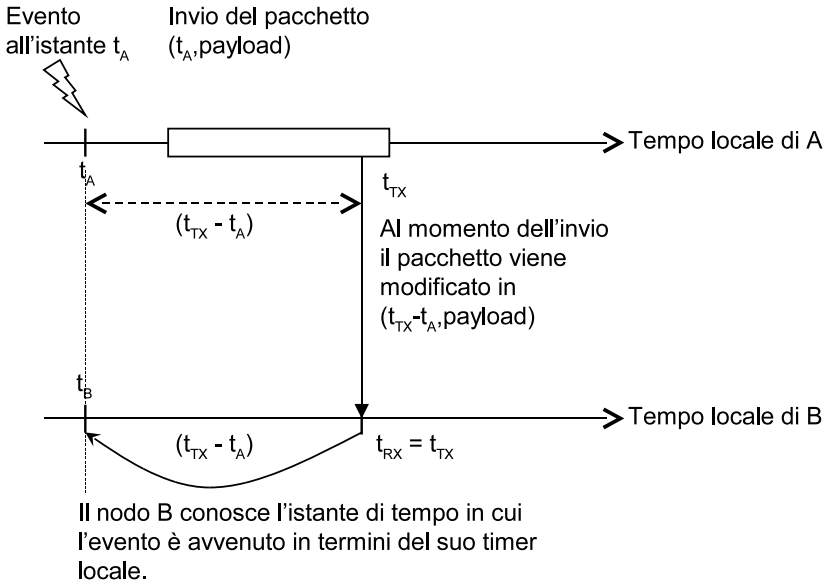


Immagine 2.7: Esempio di packet level synchronization

2.2.4 Funzionalità di sicurezza del CC2420 in TinyOS

In questo lavoro sono state sfruttate alcune capacità del chip radio per le quali non è ancora disponibile un interfacciamento software nella release ufficiale di TinyOS. Si tratta delle funzionalità di sicurezza offerte dal CC2420.

Come spiegato nella sezione 2.1.1, il chip radio implementa in hardware i servizi di sicurezza per l'autenticazione e la cifratura dei pacchetti trasmessi. Oltre a questi servizi offre inoltre la possibilità di utilizzare il modulo di cifratura AES in modalità stand-alone. In TinyOS attualmente l'implementazione dell'interfaccia verso i servizi di sicurezza dei pacchetti trasmessi sono disponibili sul repository SVN di TinyOS, ma non sono ancora presenti nell'ultima release ufficiale, la 2.1.0, che risale al 2008. In questo lavoro è stato fatto uso quindi della versione non ancora rilasciata ufficialmente di TinyOS. Inoltre per utilizzare la funzionalità di cifratura stand-alone ci si è basati su del codice sviluppato dal Cryptography and Information Security Lab dell'università di Shangai [10], apportando alcune modifiche e correzioni a problemi di funzionamento.

Conflitto tra packet level authentication e time synchronization

Alcune limitazioni imposte dal chip radio purtroppo influiscono sulla possibilità di autenticare un pacchetto contenente un offset temporale come quelli precedentemente descritti.

Il meccanismo di autenticazione del pacchetto implementato nel CC2420 prevede che al payload sia annesso il MAC non appena venga dato il comando di invio del pacchetto. Ma questo sistema entra inevitabilmente in conflitto con il meccanismo di sincronizzazione, che invece richiede di modificare il payload (per scrivere l'offset) solamente quando il pacchetto comincia ad essere effettivamente trasmesso sul canale radio. È necessario infatti, per garantire l'autenticità, che se il payload viene modificato dopo che il MAC è stato calcolato e incluso nel pacchetto, la verifica del MAC lato ricevitore fallisca.

Questo ostacolo sarebbe aggirabile se il chip radio permettesse di ricalcolare il MAC anche una volta che il pacchetto è in fase di trasmissione, ma non essendo questo possibile nel CC2420 è stato necessario trovare un'altra soluzione.

È stato quindi deciso di ridurre la precisione del servizio di sincronizzazione per garantire comunque l'affidabilità del servizio di autenticazione. Ciò è stato fatto modificando il codice del componente `CC2420TransmitP` che si occupa della gestione del CC2420 per quanto riguarda la trasmissione, andando a scrivere l'offset nel pacchetto non al momento dell'invio effettivo sul canale radio, ma subito prima dell'invio al chip radio del comando di invio del pacchetto. L'immagine 2.8 illustra questo cambiamento.

Il cambiamento introdotto comporta una sincronizzazione imperfetta tra trasmettitore e ricevitore, in particolare si avrà che il ricevitore si troverà in ritardo rispetto al trasmettitore del tempo intercorso tra l'invio del comando di trasmissione al chip radio e l'inizio effettivo di trasmissione del frame sul

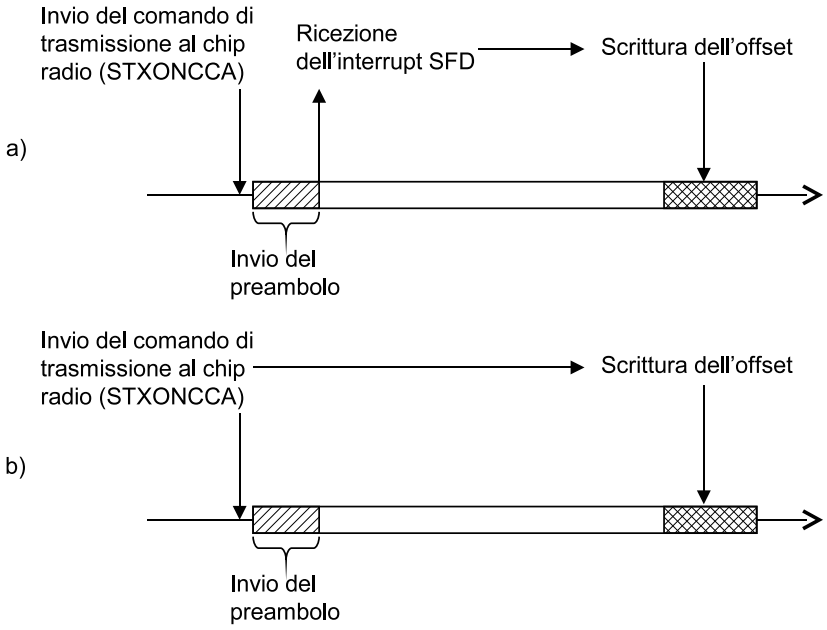


Immagine 2.8: Combinazione dei servizi di autenticazione e sincronizzazione

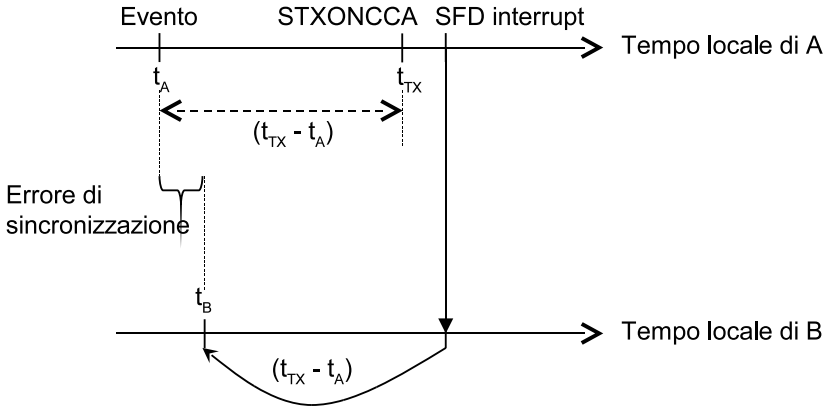


Immagine 2.9: Errore di sincronizzazione dovuto al servizio di autenticazione

canale, come si può vedere in figura 2.9.

Questo errore dovrebbe essere un errore sistematico pari al tempo necessario al chip per elaborare il comando più il tempo di trasmissione del preambolo, e pertanto dovrebbe essere possibile la sua eliminazione una volta ottenuto il suo valore. Attualmente l'entità di questo errore non si è rivelata tale da pregiudicare il funzionamento di SecureSynapse e pertanto

la sua correzione è rimandata a sviluppi futuri.

Un problema più importante introdotto dall'autenticazione invece è dovuto al meccanismo di CCA (Clear Channel Assessment) implementato dal CC2420. Quando viene invocato il comando STXONCCA infatti quello che il chip fa è prima di tutto controllare che il canale sia libero, e solamente dopo averlo accertato trasmettere il pacchetto. Se il canale non è libero allora la trasmissione non viene eseguita e questo viene rilevato nel software interrogando il registro di stato del CC2420. A questo punto TinyOS implementa un meccanismo di backoff e ritrasmette il pacchetto dopo un certo intervallo di tempo. Il conflitto di questo sistema con il servizio di autenticazione nasce dal fatto che l'offset scritto nel pacchetto va modificato al momento della ritrasmissione, ma il MAC non viene ricalcolato quando viene ridato il comando STXONCCA. Non è chiaro il motivo per cui questo accade, probabilmente è necessario eseguire uno svuotamento (flush) del buffer di trasmissione per far si che il chip riesegua il calcolo del MAC, ma questo implicherebbe modifiche abbastanza profonde al complesso software che gestisce la trasmissione. Sarebbe inoltre necessario discutere queste modifiche con i responsabili dello sviluppo di questo componente software sulla mailing list degli sviluppatori di TinyOS in modo da sviluppare un sistema robusto che venga incluso nelle prossime release del sistema operativo.

Anche questa parte quindi è stata lasciata a sviluppi futuri, ed è stato deciso che nei casi in cui dovesse accadere una ritrasmissione dovuta a canale occupato, sarà trasmesso un pacchetto con l'offset corretto ma con il MAC errato. In questo modo il pacchetto sarà semplicemente scartato al momento in cui ne verrà controllata l'autenticità. Questa scelta è stata preferita rispetto alla scelta di trasmettere il pacchetto con offset errato ma MAC corretto in quanto l'errore di sincronizzazione che può apportare questo problema non è trascurabile come nel caso precedente. È stata preferita anche alla scelta di non eseguire per niente la ritrasmissione in quanto anche

questa avrebbe comportato modifiche radicali al codice di TinyOS.

2.3 Synapse++

Synapse++ [11] è un sistema per la riprogrammazione wireless di reti di sensori, che adotta un meccanismo di recupero degli errori basato su rateless Fountain Codes. Questo sistema permette una scalabilità maggiore al crescere della densità della rete e una migliore tolleranza al rumore rispetto alle soluzioni utilizzanti strategie di ARQ basate su NACK. Synapse++ è a sua volta una versione migliorata di Synapse[12], rispetto al quale introduce un sistema di pipelining atto a velocizzare il processo di disseminazione, e ottimizza l'implementazione dei fountain codes in modo da massimizzare gli errori corretti grazie ad overhearing, minimizzando così il numero di ritrasmissioni esplicite.

Nelle sezioni seguenti verranno trattati i concetti più rilevanti per lo sviluppo e la comprensione del lavoro svolto. Nella sezione 2.3.1 verranno approfonditi alcuni concetti basilari sulla riprogrammazione wireless, mentre nella sezione 2.3.2 verranno trattati alcuni aspetti del funzionamento di Synapse++ .

2.3.1 Riprogrammazione wireless

Indipendentemente dal protocollo di disseminazione utilizzato, i sistemi per la riprogrammazione wireless di reti di sensori condividono alcune caratteristiche. La prima di queste è la necessità di dividere l'applicazione da disseminare in blocchi, di una dimensione sufficientemente piccola da consentire la loro elaborazione all'interno della memoria RAM del nodo, prima di essere scritti nella memoria permanente esterna. Questi blocchi sono denominati blocchi di trasporto, o Transport Blocks (TBs). Una volta che

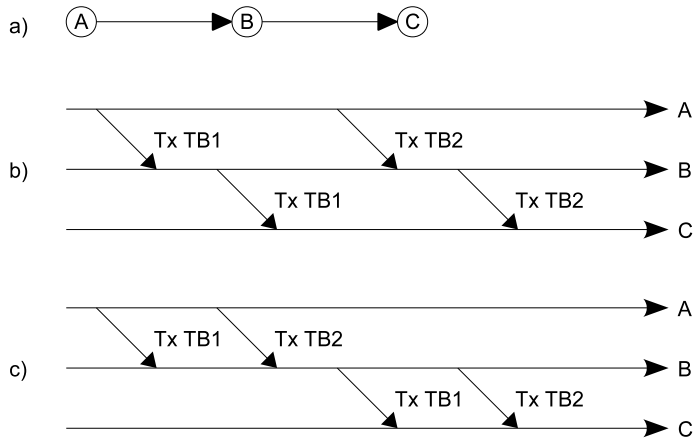


Immagine 2.10: Meccanismo di pipelining.

l'applicazione è stata divisa in blocchi ogni blocco viene trasmesso via radio ed è a questo punto che i vari protocolli di disseminazione si differenziano sostanzialmente. Anche in questa fase comunque si possono ritrovare caratteristiche comuni. In primo luogo la disseminazione dei blocchi può avvenire seguendo l'ordine dei blocchi stessi oppure no (disseminazione out-of-order) e generalmente protocolli che ammettono la disseminazione out-of-order hanno migliori prestazioni in termini di velocità. Un ulteriore tecnica che aumenta la velocità di disseminazione è quella del pipelining, che consiste nel dare priorità alla diffusione dei blocchi in "profondità" nella rete piuttosto che in "ampiezza". Un esempio si può vedere in figura 2.10. Consideriamo una rete composta da soli tre nodi, A, B e C, dove A dista un hop da B e C dista un hop da B. In (b) è possibile vedere il meccanismo di pipelining che dà precedenza all'avanzamento del blocco 1 nella rete piuttosto che all'avanzamento locale dell'intera applicazione da disseminare, come accade invece in (c).

In questo modo si promuove il parallelismo in quanto zone della rete che distano più di un hop potranno disseminare i blocchi al loro interno in modo indipendente l'una dall'altra, velocizzando quindi il processo di disseminazione.

2.3.2 Dettagli su Synapse++

Vari aspetti del funzionamento di Synapse++ determinano scelte di design che sono state fatte nello sviluppo dei sistemi di sicurezza, e determinano le performance del sistema risultante. Questa sezione sarà un richiamo o un approfondimento di questi aspetti, una spiegazione esaustiva del funzionamento di Synapse++ è invece consultabile in [11].

Fountain Codes

In Synapse++ vengono utilizzati i Fountain Codes, noti anche come rateless erasure codes. Questi sono una classe di codici caratterizzati dalla proprietà che una sequenza potenzialmente illimitata di simboli codificati possono essere generati da un dato insieme di simboli sorgente. Da qui il nome rateless, in quanto questi codici non presentano un rate di codifica fissato a priori. I simboli originali possono essere idealmente recuperati a partire da un qualsiasi sottoinsieme dei simboli codificati di grandezza uguale o leggermente maggiore rispetto al numero di simboli originali. Le modalità di utilizzo e l'importanza dei codici fontana in Synapse++ comunque sono discusse approfonditamente in [11].

Protocollo di disseminazione

La comunicazione necessaria per la disseminazione si riduce sostanzialmente a tre messaggi scambiati tra i nodi: ADV, REQ e DATA. Ogni nodo periodicamente invia in broadcast dei messaggi ADV, nei quali specifica l'id dell'applicazione che intende disseminare, e una bitmask indicante i blocchi che ha a disposizione. Quando un nodo riceve un messaggio di ADV, verifica se è di suo interesse controllando che il mittente possieda dei blocchi ad esso mancanti. In generale un nodo può ricevere più messaggi ADV di interesse da diversi nodi. Tra questi il nodo scelto è il nodo più vicino alla base sta-

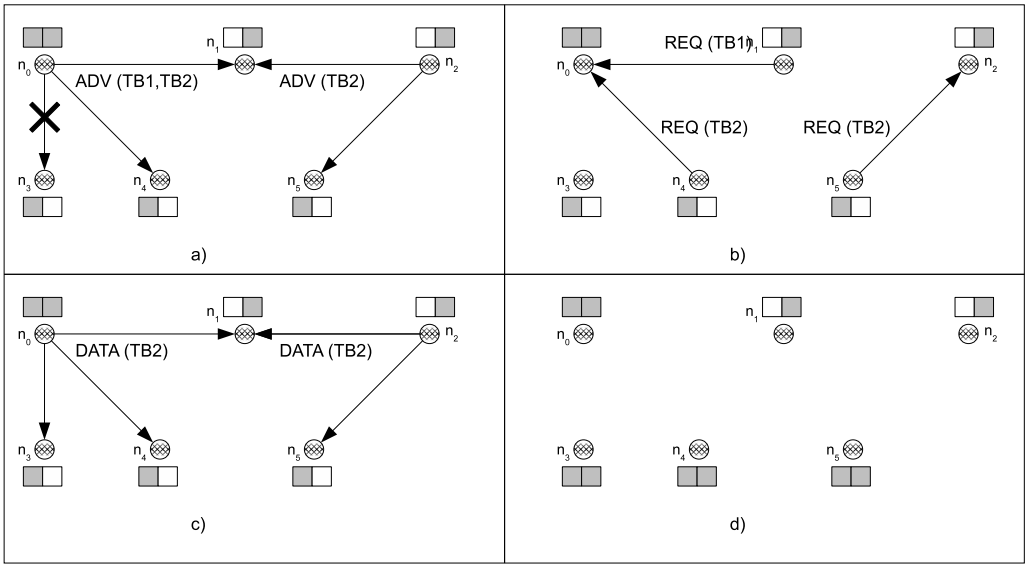


Immagine 2.11: Messaggi scambiati tra i nodi di una rete a due hop durante un ciclo ADV-REQ-DATA

tion. Una volta scelto il nodo a cui richiedere il blocco, si procede all'invio di un messaggio REQ. In questo messaggio verrà specificato il blocco richiesto. A questo punto il nodo che ha inviato il messaggio di ADV avrà ricevuto potenzialmente molti messaggi REQ, e sceglierà di trasmettere un blocco di quelli richiesti. La scelta che viene fatta è di trasmettere il blocco di indice maggiore, in modo da promuovere l'avanzamento. Una volta deciso quale blocco disseminare, il nodo invierà i pacchetti DATA necessari per trasmetterlo. Oltre a questi messaggi c'è un quarto messaggio, CMD, che serve a trasmettere comandi dalla base station in broadcast verso tutti i nodi, e viene usato per esempio per formattare la flash o richiedere l'avvio di una applicazione.

In figura 2.11 è possibile vedere un esempio di un ciclo ADV-REQ-DATA in una semplice rete con 2 hop. Si nota come il nodo n_1 riceva due messaggi ADV, ma solo il messaggio da n_0 sia di suo interesse. A sua volta n_0 riceve due messaggi REQ, e decide di soddisfare quello che richiede il blocco di

indice più alto. Il nodo n_3 invece non riceve il messaggio ADV a causa per esempio di errori sul canale, ma è riceve comunque il blocco trasmesso da n_0 grazie all'overhearing.

Sincronizzazione e pipelining

In Synapse++ viene implementato il sistema di sincronizzazione su singolo hop descritto nella sezione 2.2.3. La sincronizzazione sfrutta il messaggio ADV, e consente di dividere il tempo in frame, ciascuno dei quali è diviso nei periodi ADV,REQ e DATA, nei quali vengono trasmessi i messaggi corrispondenti. La base station inizia il processo di sincronizzazione all'interno del primo hop quando trasmette il primo messaggio ADV. Successivamente quando i nodi al primo hop hanno ricevuto un blocco ed iniziano a loro volta a trasmettere ADV anche i nodi al secondo hop si sincronizzano, e così via lungo tutta la rete. È importante comunque sottolineare che i nodi non sono sincronizzati *globalmente*, quindi quanto più due nodi distano in termini di hop tanto più potrà essere la differenza di sincronizzazione tra i due. Per il funzionamento di Synapse++ comunque è sufficiente che i nodi siano sincronizzati *localmente*. Grazie a questa sincronizzazione è possibile coordinare efficientemente le attività dei vari nodi. Inoltre in Synapse++ la sincronizzazione viene utilizzata per implementare il sistema di priorità necessario per la tecnica del pipelining. Come è possibile vedere in figura 2.12 i periodi ADV e REQ sono divisi a loro volta in due sotto-intervalli, uno ad alta e uno a bassa priorità. Quello che accade quindi è che quando un nodo ha ricevuto correttamente un blocco, imposta il suo livello di priorità ad *alto*, e lo mantiene per tutto il *frame* successivo. Gli ADV che esso invierà quindi verranno spediti nel periodo di alta priorità dell'intervallo ADV, e avranno quindi la precedenza rispetto agli ADV spediti nell'intervallo a bassa priorità. In questo modo viene favorita la propagazione in profondità

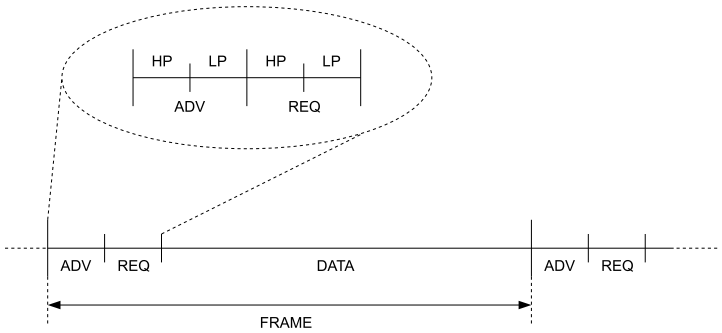


Immagine 2.12: Suddivisione temporale dei frame in Synapse++

nella rete dei nuovi blocchi rispetto alla correzione locale degli errori e alla diffusione locale di nuovi blocchi.

Gestione della memoria esterna

In TinyOS la memorizzazione di grandi (più di 2KB) quantità di dati è possibile solamente utilizzando il componente `BlockStorageC`, che espone le interfacce `BlockRead` e `BlockWrite`. Il modello di lettura e scrittura è un modello *erase, write once, read many*. La memoria va quindi prima di tutto inizializzata con una operazione di *erase*, che elimina i dati presenti in porzioni di flash di grandi dimensioni, nel nostro caso 64KB, inizializzando ogni bit ad un valore predefinito. Una volta eseguita l'operazione di *erase* la memoria può essere scritta una e una sola volta, dopodiché deve essere utilizzata solo in lettura. La motivazione fisica alla base di questo funzionamento risiede nella natura della memoria flash. Quello che accade di fatto è che portare i bit di memoria dal valore 0 al valore 1 (operazione di *erase*) è un'operazione costosa in termini di energia, e può essere eseguita solo su blocchi di molti bit adiacenti (denominati *settori*, nel nostro caso 64KByte). Una volta portati i bit a valore 1, i bit possono essere riportati al valore 0 in modo indipendente l'uno dall'altro (operazione di *write*), consentendo così la scrittura di dati arbitrari. L'operazione di scrittura in flash si traduce

quindi in una operazione di AND bit a bit, tra il dato presente in flash e il dato che si desidera scrivere. Se si tenta di scrivere più volte in una stessa locazione di flash si esegue ogni volta un AND del contenuto della flash con il dato che si desidera scrivere, ottenendo così dati corrotti.

Nel contesto della riprogrammazione di una rete di sensori, è ragionevole supporre che sia desiderabile poter mantenere nella memoria esterna di ogni nodo un certo insieme composto da più applicazioni, per poter eseguire l'una o l'altra a seconda delle necessità. Ognuna di queste applicazioni inoltre deve poter essere aggiornabile in modo indipendente dalle altre, e queste necessità si scontrano con il modello di memoria esterna offerto da TinyOS. Si vorrebbe infatti aver accesso ad una serie di partizioni di grandezza variabile (una per ogni applicazione da mantenere nel nodo), ognuna delle quali sia cancellabile in modo indipendente dalle altre (per permettere l'aggiornamento indipendente delle applicazioni). Si potrebbe pensare di allocare un *settore* per ogni applicazione, ma questo diventerebbe costoso visto che la massima dimensione occupabile da un'applicazione nel nostro caso è di 48KB, e visto che di rado le applicazioni raggiungono una tale dimensione. La soluzione adottata in Synapse è quindi quella di implementare un sistema di gestione di partizioni. Non avendo la possibilità di eliminare il contenuto di una partizione quello che viene fatto è associare un id ad ogni partizione, e nel caso si voglia aggiornare il contenuto di una partizione, verrà solamente creata una nuova partizione con lo stesso id. Un esempio di come siano memorizzate le partizioni e la tabella delle partizioni è osservabile in figura 2.13. La tabella delle partizioni contiene un record per ogni partizione presente, nel quale è memorizzato l'id della partizione, la sua dimensione e la sua posizione. Si nota come la tabella stessa occupi una partizione, di dimensione 256B. Le altre partizioni sono scritte consecutivamente a partire dalla fine di questa.

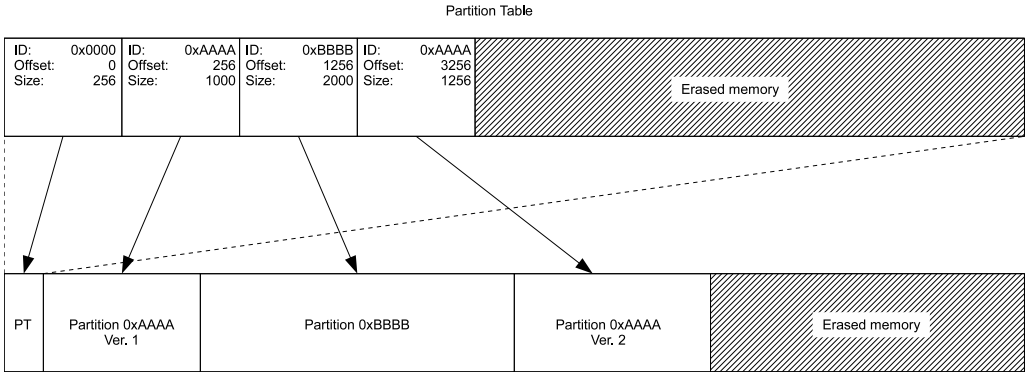


Immagine 2.13: Gestione delle partizioni in Synapse++

Capitolo 3

Sistemi di sicurezza

In questo capitolo verranno esposti alcuni concetti teorici riguardanti i sistemi di sicurezza che sono stati utilizzati per implementare i meccanismi di protezione in Synapse++ .

In primo luogo viene presentato un modello di valutazione del livello di sicurezza di un sistema. Dopodiché vengono introdotte le funzioni di hash, di cui vengono discusse le proprietà fondamentali. Successivamente viene presentata la firma digitale, e due algoritmi di firma digitale: la Merkle One-Time Signature e l'algoritmo T-TimeSA. Infine vengono discussi gli algoritmi di cifratura, spiegando i concetti di cifrari a blocchi e di modalità operative e introducendo quindi le tecnologie di cifratura utilizzate in questo lavoro.

3.1 Valutazione del livello di sicurezza

La sicurezza di un sistema, sia fisico che informatico, non è mai assoluta, la valutazione che si può dare è invece il costo che un avversario deve affrontare per superarla. Il costo è inteso sia nei termini economici, ma soprattutto nel nostro caso in termini temporali, come il tempo di calcolo necessario per trovare una soluzione al problema computazionale che costituisce la barriera di sicurezza. Se per esempio la sicurezza di un qualche algoritmo di cifratura è garantita finché un avversario non riesce a fattorizzare un dato numero, allora il livello di sicurezza garantito è pari al tempo di calcolo necessario a risolvere il problema della fattorizzazione. Il livello di sicurezza oltre ad essere espresso in termini temporali, viene anche misurato come probabilità, nel qual caso indica la probabilità che un avversario ha di riuscire a superare i sistemi di sicurezza. In questo caso si sottintende un meccanismo di attacco a tentativi, in cui l'attaccante non conosce un algoritmo in grado di calcolare direttamente la soluzione al problema in questione, ma è in possesso invece di un algoritmo che è in grado di stabilire l'esattezza o meno di una data soluzione. In questo caso quindi si misura la sicurezza calcolando la probabilità che un avversario ha di "indovinare" la soluzione corretta. Esprimendo questa probabilità come potenza di 2, l'opposto dell'esponente risulterà invece essere lo stesso livello di sicurezza espresso in bit.

Livello di sicurezza in bit	64
Probabilità del singolo tentativo	2^{-64}
Numero atteso di tentativi necessari	2^{63}
Tempo di calcolo per eseguire un tentativo	1 µs
Tempo di calcolo necessario a risolvere il problema	circa 18 anni

Tabella 1: Esempio di calcolo del livello di sicurezza

3.2 Algoritmi di hash

Un primo elemento da introdurre per la costruzione di sistemi di sicurezza sono le funzioni di hash. Una funzione di hash è una funzione non iniettiva che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita. Le ulteriori proprietà che una funzione di hash deve avere dipendono generalmente dall'applicazione. Per la costruzione di una tabella hash per esempio è sufficiente garantire che la distribuzione di probabilità degli hash sia uniforme. Nel caso di applicazioni crittografiche invece possono essere richieste le seguenti proprietà:

- resistenza alla pre-immagine: deve essere computazionalmente intrattabile la ricerca di una stringa in input che dia in output un dato hash;

- resistenza alla seconda pre-immagine: deve essere computazionalmente intrattabile la ricerca di una stringa in input che dia un hash uguale a quello di una data stringa;

- resistenza alle collisioni: deve essere computazionalmente intrattabile la ricerca di una coppia di stringhe in input che diano lo stesso hash.

Per considerare un algoritmo di hash come non sicuro è generalmente sufficiente che un gruppo di ricercatori riesca a trovare una collisione. Questo è quanto è accaduto per esempio per l'algoritmo MD5 tra 2005 e 2006, per il quale è possibile trovare collisioni in meno di 30 secondi [13]. Nello stesso periodo inoltre è stato scoperto un attacco anche verso l'algoritmo SHA-1. In questo caso si è dimostrato di poter ridurre di circa tre ordini di grandezza il tempo di ricerca di una collisione rispetto ad un attacco brute-force, ma comunque il tempo di calcolo necessario rimane ancora molto elevato.

3.2.1 Algoritmo SHA-1

Le specifiche originali dell'algoritmo risalgono al 1993, quando furono pubblicate dall'agenzia governativa statunitense NIST (National Institute of Standards and Technology), come funzione hash sicura standard. Questa versione ora denominata SHA-0 è stata ritirata poco dopo la pubblicazione, a causa della scoperta di alcune falle di sicurezza, e sostituita da una versione rivisitata pubblicata nel 1995 e conosciuta come SHA-1. SHA-1 produce stringhe di 160 bit a partire da messaggi di lunghezza massima di (2^{64}-1) bit.

3.3 Algoritmi di firma digitale

Uno schema di firma digitale è uno schema matematico per dimostrare l'autenticità di un messaggio digitale. Una firma digitale valida garantisce al ricevente che il messaggio è stato creato da un mittente conosciuto, e che il messaggio non è stato alterato. La firma digitale assicura inoltre il non ripudio, cioè il firmatario di un documento trasmesso non può negare di averlo firmato.

Il sistema per la creazione e la verifica di firme digitali sfrutta le caratteristiche della crittografia asimmetrica e consiste generalmente in tre algoritmi:

- Un algoritmo di generazione delle chiavi, che sceglie una chiave privata uniformemente in uno spazio di chiavi private possibili. L'algoritmo produce in output la chiave privata e la corrispondente chiave pubblica.

- Un algoritmo di firma, che dato un messaggio e una chiave privata, produce una firma.

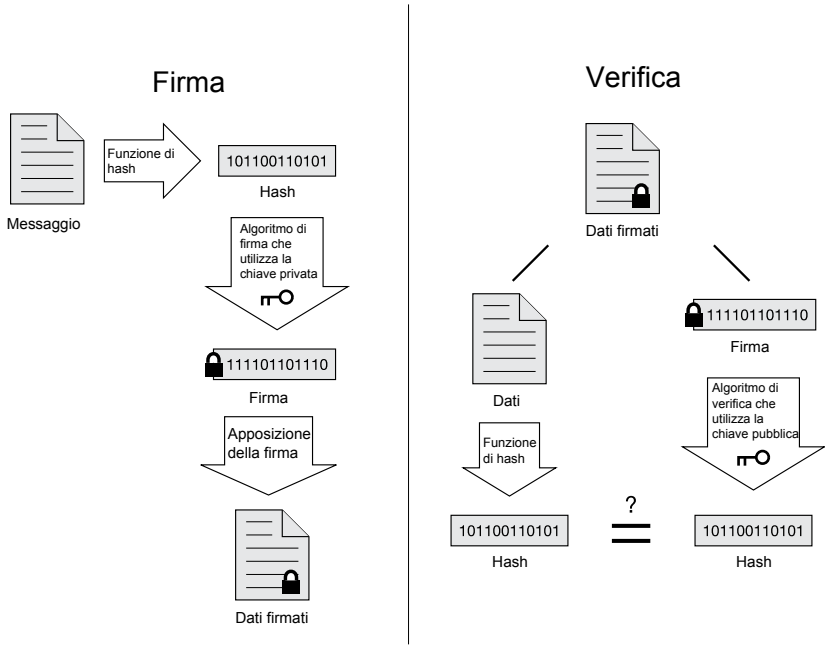


Immagine 3.1: Esempio di firma digitale

- Un algoritmo di verifica, che dato un messaggio, una chiave pubblica e una firma, conferma o smentisce l'autenticità del messaggio.

Generalmente inoltre l'algoritmo di firma non viene applicato al messaggio in sé, ma ad un hash del messaggio stesso, ottenuto con una funzione di hash sicura. Questo viene fatto per ridurre la dimensione del messaggio da firmare, generalmente a lunghezza variabile, ad un numero prefissato di bit. Questo numeri sarà pari al livello di sicurezza desiderato, in quanto invertire una funzione di hash sicura richiede un attacco brute force.

3.3.1 Valutazione della sicurezza

Ai fini della valutazione del livello di sicurezza ottenibile con un certo sistema di firma digitale, in [14] viene definita una gerarchia di possibili attacchi:

1. In un attacco di tipo "key-only", l'attaccante conosce solamente la

chiave pubblica.

2. In un attacco di tipo "known-message", l'attaccante conosce firme valide per un determinato insieme di messaggi non di sua scelta.

3. In un attacco di tipo "adaptive chosen message", l'attaccante è in grado di entrare a conoscenza di firme valide per messaggi di sua scelta.

Viene inoltre descritta una gerarchia di possibili risultati degli attacchi:

1. Un "total break" consiste nel ritrovamento della chiave segreta.

2. Un "universal forgery" consiste nell'abilità di creare firme per un messaggio arbitrario.

3. Un "selective forgery" consiste nell'abilità di creare una firma valida per un messaggio scelto dall'attaccante prima dell'attacco.

4. Un "existential forgery" consiste nell'abilità di creare una firma valida per un messaggio scelto dall'attaccante durante dell'attacco.

Il livello maggiore di sicurezza si ha quindi quando l'avversario non è in grado di concludere con successo un "existential forgery" sottoponendo il sistema ad un attacco di tipo "adaptive chosen message".

3.3.2 Merkle One-Time Signature

La Merkle One-Time Signature [15] è un algoritmo di firma digitale che sfrutta una qualsiasi funzione di hash sicura come primitiva crittografica asimmetrica. Le proprietà che la funzione deve avere sono esattamente quelle esposte nella sezione 3.2, cioè resistenza alla pre-immagine, alla seconda pre-immagine e alle collisioni. Questo meccanismo di firma prevede che

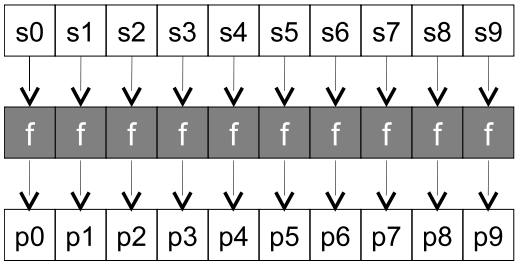


Immagine 3.2: Esempio di Merkle One-Time Signature. Generazione delle chiavi

le chiavi possano essere utilizzate una volta sola, dopodiché devono essere rigenerate e le chiavi pubbliche ridistribuite, da qui il nome "One-Time".

I tre algoritmi che costituiscono il sistema di firma digitale sono i seguenti.

Generazione delle chiavi

Innanzitutto devono essere decisi il parametro di sicurezza n e una funzione di hash sicura f. Il parametro n corrisponde al livello di sicurezza che si desidera ottenere, come esposto nella sezione 3.1, e limita inoltre a n bit la lunghezza del messaggio che si vuole firmare. A questo punto vengono generati in modo casuale $m = n + \lfloor \log_2 n \rfloor + 1$ vettori s_i di n bit ciascuno, che costituiranno la chiave segreta S. Dalla chiave segreta S viene generata la chiave pubblica P, applicando la funzione f a ciascun vettore s_i che compone la chiave. In figura 3.2 si può vedere un esempio di generazione delle chiavi con parametro $n = 7$.

Creazione della firma

In input all'algoritmo di firma vi sono il messaggio da firmare M e la chiave segreta S. Il primo passo consiste nell'aggiungere al messaggio M gli $\lfloor \log_2 n \rfloor + 1$ bit che rappresentano il conteggio del numero di bit che assumono valore 0 in M, ottenendo il messaggio M'. Questo checksum serve ad impedire una banale contraffazione della firma come verrà spiegato in

seguito. A questo punto si analizza il messaggio M' e si crea la firma F concatenando le chiavi segrete s_i corrispondenti alle posizioni in M' in cui appare un bit a valore 1. È a questo punto sufficiente quindi allegare la firma F al messaggio originale M. L'esempio della fase di firma è illustrato in figura 3.3.

Verifica della firma

In input all'algoritmo di verifica vi sono il messaggio da autenticare M, la firma F, e la chiave pubblica P. Come nella fase di firma è necessario come primo passo aggiungere la rappresentazione del conteggio del numero di bit a valore 0 in M, ottenendo M'. A questo punto si estraggono dalla firma F le chiavi segrete s_i in essa contenute, e si considerano le chiavi pubbliche p_i corrispondenti alle posizioni in M' in cui appare un bit di valore 1. Il messaggio viene considerato autentico se applicando la funzione f ad ognuna delle chiavi segrete s_i si ottiene la corrispondente chiave pubblica p_i. Anche questo passaggio è illustrato in figura 3.3.

Principio di funzionamento

Il principio di funzionamento di questo algoritmo di firma si fonda sull'impossibilità pratica di invertire una one-way function, nel nostro caso la funzione di hash. Nel momento in cui non è ancora stata rilasciata alcuna firma l'unico attacco possibile verso questo sistema di firma è l'attacco "key-only", essendo le chiavi pubbliche l'unica informazione a disposizione dell'attaccante. Ma dalle chiavi pubbliche essendo la funzione di hash non invertibile non è possibile ricavare alcuna informazione sulle chiavi private, rendendo quindi impossibile un qualsiasi tipo di attacco.

Nel momento in cui un messaggio firmato viene rilasciato, l'attaccante potrebbe essere in grado di intercettare e modificare queste informazioni

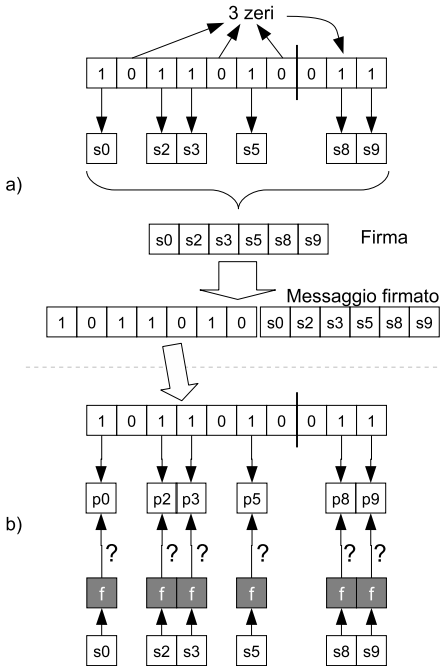


Immagine 3.3: Esempio di Merkle One-Time Signature. Creazione (a) e verifica (b) della firma

per compiere un attacco, in questo caso di tipo "known-message". È per assicurare la protezione in questo tipo di attacco che nell'algoritmo di firma è presente la fase di calcolo del checksum. Se questo non fosse presente infatti un attacco banale consisterebbe nel modificare un bit del messaggio da 1 a 0 e contemporaneamente rimuovere dalla firma la chiave corrispondente, ottenendo una firma comunque valida. L'attaccante potrebbe quindi forgiare firme valide per tutti i messaggi ottenibili dal messaggio originale sostituendo bit 1 con bit 0, ottenendo un "existential forgery". Introducendo il checksum invece non è possibile cambiare dei bit da 1 a 0 nel messaggio firmato senza cambiarne nessuno da 0 a 1 (e quindi dovendo conseguentemente inserire una chiave non ancora rilasciata nella firma) e mantenendo un checksum valido. Questo schema è quindi sicuro al massimo livello di sicurezza.

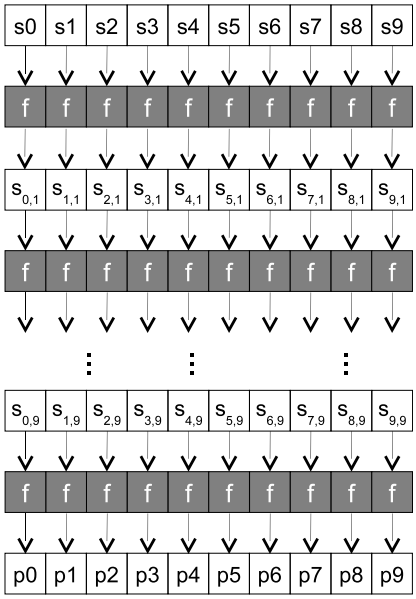


Immagine 3.4: Esempio di T-TimeSA. Generazione delle chiavi

3.3.3 TTimeSA

Il T-Time Signature Algorithm (TTimeSA) è un'estensione dell'algoritmo Merkle One-Time Signature, proposta in [16], come possibile sistema di firma adottabile in reti di sensori wireless. Questo algoritmo introduce rispetto a quello di Merkle la possibilità di eseguire T autenticazioni senza la necessità di ridistribuire le chiavi. I tre algoritmi che costituiscono il sistema di firma digitale sono i seguenti.

Generazione delle chiavi

La fase generazione delle chiavi è del tutto simile a quella dell'algoritmo di Merkle. Le differenze consistono per prima cosa in un ulteriore parametro da scegliere, il parametro T, che indica il numero di operazioni di autenticazione eseguibili prima di dover ridistribuire le chiavi. Una volta scelto il parametro T inoltre, la generazione delle chiavi pubbliche p_i a partire dalle chiavi private s_i non avviene applicando la funzione di hash f una sola volta, ma T volte. Un esempio di ciò è visibile in figura 3.4. Con questo procedimento

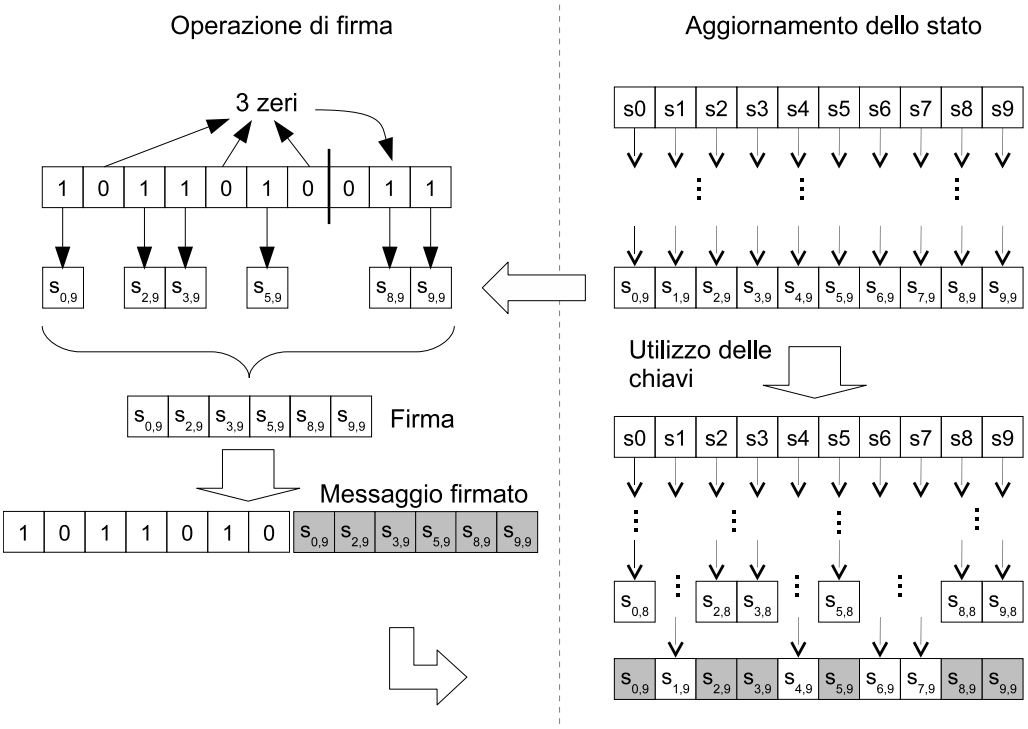


Immagine 3.5: Esempio di T-TimeSA. Operazione di firma. L'applicazione della funzione di hash in questo caso è rappresentata dalla singola freccia, a differenza dell'immagine 3.4

otteniamo anche $T-1$ chiavi intermedie $s_{i,j}$, dove $s_{i,j}$ è la chiave ottenuta applicando j volte la funzione di hash alla chiave s_i. Questo algoritmo inoltre prevede a differenza dell'algoritmo di Merkle un mantenimento di uno stato comune tra mittente e ricevente, costituito dalla conoscenza da parte di entrambi del numero di volte che nelle operazioni di firma è stata usata ogni chiave s_i. Indichiamo questo stato con st_i, che sarà inizializzato a 0 per ogni chiave al momento della distribuzione.

Creazione della firma

Anche la fase di creazione della firma è molto simile all'algoritmo di Merkle. Come illustrato in figura 3.5, la differenza è che invece di inserire nella firma le chiavi s_i vengono inserite le chiavi $s_{i,T-st_i}$ corrispondenti alle posizioni in

M' in cui appare un bit a valore 1. Inoltre una volta utilizzata la chiave $s_{i,T-st_i}$ lo stato st_i, indicante il numero di utilizzi della chiave s_i, viene aggiornato di conseguenza con un incremento di un'unità.

Questo significa che alla prima operazione di firma l'algoritmo si comporta in modo simile a quello di Merkle, utilizzando però invece delle chiavi segrete s_i le chiavi "intermedie" $s_{i,T-1}$, cioè quelle che precedono le chiavi p_i nelle catene di hash che le hanno generate. Queste chiavi poi risulteranno "consumate", e nella successiva autenticazione verranno utilizzate quelle precedenti ancora.

Verifica della firma

La fase di verifica della firma procede come nell'algoritmo di Merkle verificando che la funzione di hash applicata alle chiavi contenute nella firma dia come risultato le corrispondenti chiavi pubbliche. Una volta che la firma è stata verificata con successo lo stato deve essere aggiornato, e questo al ricevitore viene fatto rimpiazzando le chiavi pubbliche appena verificate con le chiavi ricevute nella firma, "consumando" così le chiavi anche al lato ricevitore. Un esempio del "consumo" delle chiavi è illustrato in figura 3.6.

Analisi della sicurezza

In [16] viene condotta un'analisi dettagliata del livello di sicurezza garantito da questo algoritmo. L'attacco che viene considerato possibile a questo meccanismo di firma sfrutta la necessità di mantenere uno stato condiviso tra trasmettitore e ricevitore, e prevede che l'avversario sia in grado di bloccare la comunicazione tra trasmettitore e ricevitore per un certo numero volte consecutive, intercettando i messaggi firmati. Si tratta quindi di un attacco di tipo "known-message", e il risultato ottenibile è un "existential forgery", in quanto dopo aver ottenuto un certo numero di messaggi firmati,

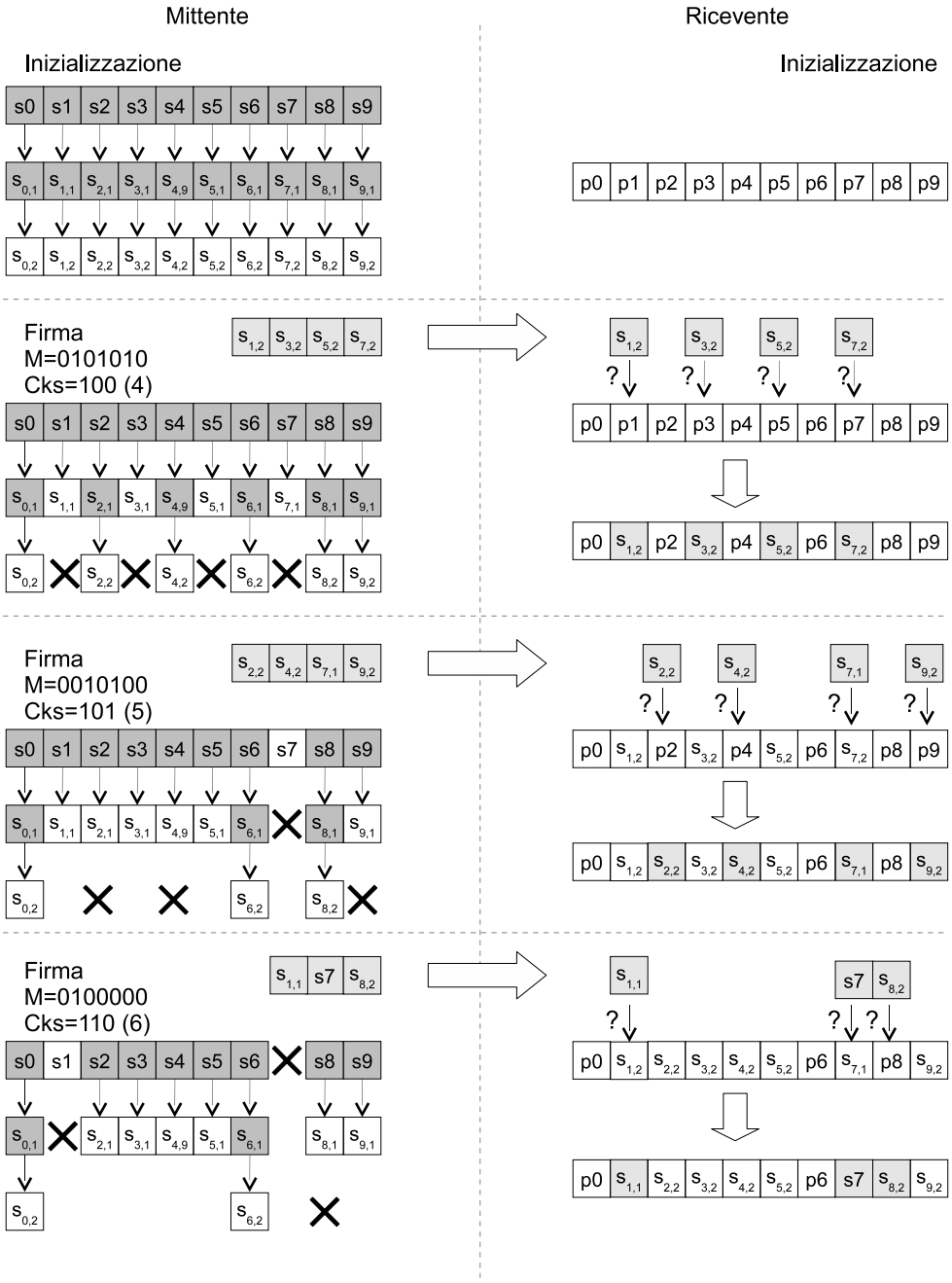


Immagine 3.6: Consumo delle chiavi in T-TimeSA.

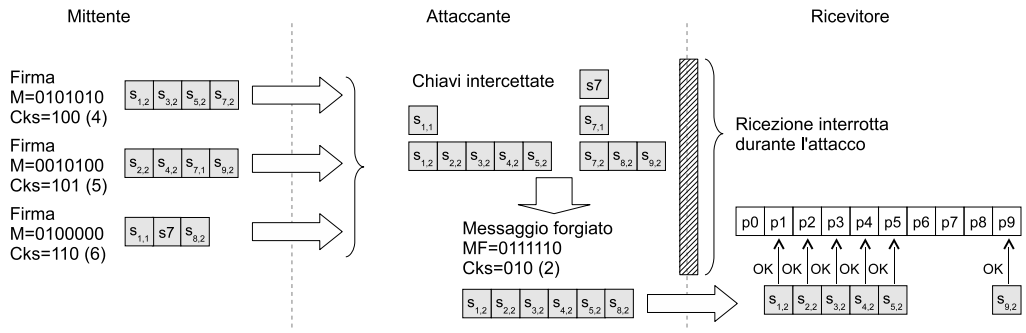


Immagine 3.7: Attacco known-message all'algoritmo T-TimeSA.

l'attaccante entra in possesso di alcune chiavi private con le quali può poi forgiare un messaggio autenticato valido da inviare al ricevitore. Un esempio è visibile in figura 3.7.

La protezione da questo tipo di attacco si ottiene di fatto utilizzando la Merkle One-Time signature, aggiungendo però un meccanismo di aggiornamento delle chiavi in modo da estenderne la durata indefinitamente. Quello che viene fatto è quindi includere nel messaggio da autenticare anche delle nuove chiavi pubbliche, da utilizzare in sostituzione delle chiavi consumate per l'autenticazione del messaggio corrente.

3.3.4 Message Autentication Code (MAC)

Un Message Authentication Code (MAC) è un piccolo blocco di dati utilizzato per autenticare un messaggio digitale, analogamente in questo senso alle firme digitali. L'algoritmo di generazione accetta in ingresso una chiave segreta ed un messaggio da autenticare di lunghezza arbitraria, e restituisce un MAC. Il MAC protegge sia l'integrità dei dati del messaggio sia la sua autenticità permettendo al destinatario dello stesso (che deve anch'egli possedere la chiave segreta) di rilevare qualsiasi modifica al messaggio. I MAC differiscono dalle firme digitali in quanto sono sia generati che verificati utilizzando la stessa chiave segreta. Questo implica che il mittente ed

il destinatario del messaggio devono scambiarsi la chiave prima di iniziare le comunicazioni. Per questa ragione i MAC non forniscono la proprietà del "non ripudio" offerta dalle firme digitali: qualunque utente che può verificare un MAC è anche capace di generare MAC per altri messaggi. Il MAC non va confuso con il Message Integrity Code (MIC). Questo si differenzia dal MAC per il fatto che non viene usata nessuna chiave segreta. Un dato messaggio produrrà sempre lo stesso MIC se si utilizza lo stesso algoritmo; contrariamente, lo stesso messaggio produrrà MAC identici solo se sarà utilizzata la stessa chiave segreta.

3.4 Algoritmi di cifratura

La cifratura è il processo di trasformazione di informazione (denominata in questo contesto testo in chiaro, o plaintext), utilizzando un algoritmo (detto cifrario, o cipher) per renderla incomprensibile da chiunque non sia in possesso di una conoscenza speciale, a cui ci si riferisce con il nome di chiave. Il risultato dell'operazione è il testo cifrato (o ciphertext).

I cifrari possono essere categorizzati in diversi modi:

- basandosi sulla dimensione dell'input che accettano si distingue tra cifrari a blocco (block cipher) nel caso in cui accettino in input un blocco di lunghezza prefissata, oppure cifrari a flusso (stream cipher) nel caso in cui accettino un flusso continuo di simboli.

- basandosi sulle chiavi con cui vengono eseguite cifratura e decifratura si distingue tra algoritmi a chiave simmetrica nel caso in cui la chiave utilizzata sia la stessa e algoritmi a chiave asimmetrica se cifratura e decifratura richiedono chiavi distinte. Nel caso di algoritmi a chiave asimmetrica in cui una chiave non sia derivabile dall'altra, l'algoritmo è un algoritmo a chiave pubblica/privata, dove la chiave privata è non

deducibile dalla chiave pubblica. In questo caso la chiave pubblica viene usata per la cifratura e può essere resa nota a chiunque senza perdita di confidenzialità

In questo lavoro è stato fatto uso di un cifrario a blocchi, che viene introdotto nella prossima sezione.

3.4.1 Cifrari a blocchi

Un cifrario a blocchi è un algoritmo che prende in ingresso un plaintext e una chiave di lunghezza fissata, e restituisce in output un ciphertext anch'esso di lunghezza fissata. Per utilizzare un cifrario a blocchi per cifrare un messaggio di lunghezza arbitraria è necessario definire un altro algoritmo, che raggiunge questo obiettivo utilizzando più volte il cifrario a blocchi. Questo algoritmo viene detto *modalità operativa* (*operation mode*). Per cifrare messaggi di lunghezza arbitraria può inoltre essere necessario effettuare un padding dei dati da dare in input al cifrario; questo padding deve essere scelto accuratamente a seconda del cifrario utilizzato, pena il rischio di rendere la cifratura vulnerabile.

Nelle prossime sezioni vengono presentate alcune modalità operative di cui è stato fatto uso in questo lavoro.

Modalità operativa Electronic CodeBook (ECB)

Questa modalità operativa è riportata solamente come esempio, in quanto nella pratica non deve essere mai utilizzata. Questa modalità prevede infatti semplicemente di suddividere il messaggio da cifrare in blocchi della dimensione attesa dal cifrario, e di cifrare ogni blocco indipendentemente, concatenando poi i blocchi di output. La vulnerabilità di questa modalità operativa consiste nel fatto che blocchi uguali nell'input corrispondono a

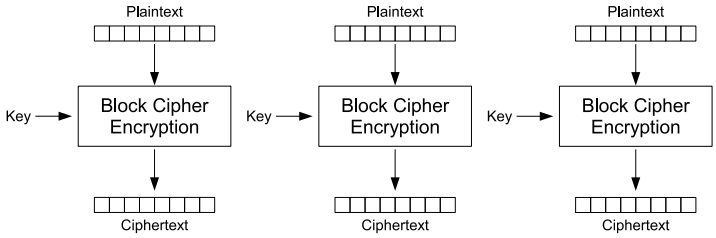


Immagine 3.8: Modalità operativa Electronic CodeBook (ECB)

blocchi uguali nell'output, e questa è una rivelazione di informazione non accettabile per un algoritmo di cifratura.

Modalità operativa Output FeedBack (OFB)

La modalità Output FeedBack costruisce un cifrario a flusso a partire da un cifrario a blocchi. Il suo funzionamento è comprensibile dall'immagine 3.9. A partire da un vettore di inizializzazione, che costituisce il primo input al cifrario a blocchi, l'output di un block cipher viene dato in input al block cipher successivo, generando così un *keystream*, cioè un flusso di blocchi che dipende solo dalla chiave e dal vettore di inizializzazione. Il ciphertext è ottenuto applicando l'operazione di XOR tra lo stream di blocchi così ottenuto e il plaintext. La fase di decifratura è identica a quella di cifratura: dal vettore di inizializzazione e dalla chiave si ricava il keystream che combinato con il ciphertext con l'operazione di XOR restituisce il plaintext.

Questa modalità operativa è sicura fintanto che vengono utilizzati vettori di inizializzazione distinti e casuali per tutte le operazioni di cifratura eseguite con una stessa chiave.

Modalità operativa Cipher Block Chaining (CBC)

In modalità Cipher Block Chaining ogni blocco di plaintext viene combinato con una operazione di XOR con il blocco di ciphertext precedente prima di essere cifrato. In questo modo ogni blocco di ciphertext dipende da tutti i

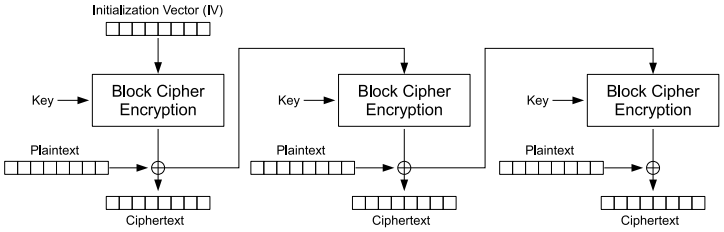


Immagine 3.9: Modalità operativa Output FeedBack (OFB)

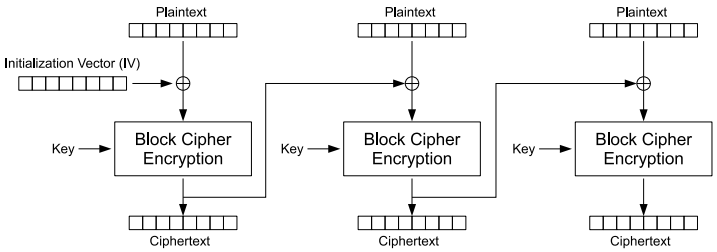


Immagine 3.10: Modalità operativa Cipher Block Chaining (CBC)

blocchi di plaintext processati fino a quel punto. Anche in questa modalità è previsto un vettore di inizializzazione casuale.

Questa modalità operativa viene utilizzata oltre per la cifratura anche per costruire MAC (vedi sezione 3.3.4), e questo si ottiene cifrando il messaggio con vettore di inizializzazione zero. L'algoritmo corrispondente è denominato CBC-MAC. L'implementazione della funzionalità di autenticazione dei messaggi nel chip radio CC2420 è basata su questa modalità operativa.

3.4.2 Schema di Davies-Meyer

Un cifrario a blocchi può essere utilizzato anche per costruire una funzione di hash sicura, aggiungendo poca logica di calcolo. Lo schema di Davies-Meyer è appunto un metodo per trasformare un cifrario a blocchi in una funzione di hash. In questo caso vettori di inizializzazione differenti definiscono funzioni di hash differenti, quindi affinché due individui ottengano lo stesso hash con lo stesso messaggio è necessario che utilizzino lo stesso vettore di

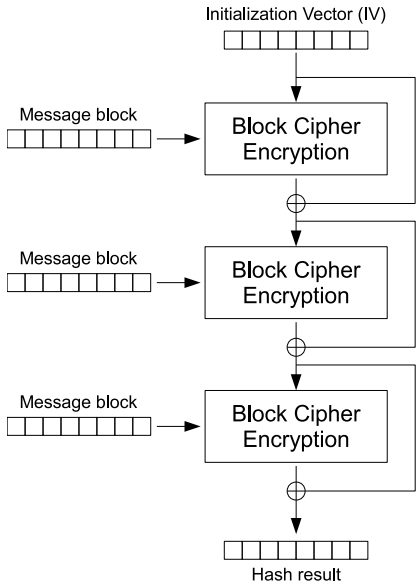


Immagine 3.11: Schema di Davies-Meyer

inizializzazione.

3.4.3 Advanced Encryption Standard (AES)

In crittografia, l'Advanced Encryption Standard (AES), conosciuto anche come Rijndael, di cui più propriamente ne è una specifica implementazione, è un algoritmo di cifratura a blocchi utilizzato come standard dal governo Statunitense. Le dimensioni di plaintext, ciphertext e chiave sono di 128 bit ciascuna.

Capitolo 4

Meccanismi di sicurezza sviluppati

In questo capitolo verranno illustrati i meccanismi di sicurezza introdotti in Synapse++ e verranno discusse e motivate le varie scelte di progettazione che sono state intraprese.

Verranno analizzati i tre obiettivi su cui il lavoro è stato focalizzato, ossia l'autenticazione, la confidenzialità e la protezione da attacchi di tipo Denial of Service.

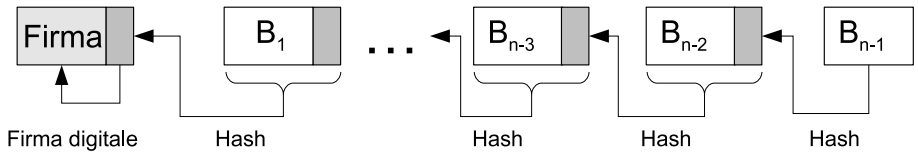


Immagine 4.1: Utilizzo di catene di hash per l'autenticazione

4.1 Autenticazione e integrità

L'obiettivo primario di questo lavoro è stato il garantire l'autenticità e l'integrità delle applicazioni installate nei nodi. Autenticità significa che solamente utenti autorizzati, tramite il rilascio di una chiave segreta, devono essere in grado di riprogrammare la rete. Integrità significa che l'applicazione deve arrivare ad ogni nodo della rete priva di corruzioni. In questo modo si scongiura il pericolo maggiore costituito da avversari che tentino di inserire applicazioni maligne nella rete.

4.1.1 Altre soluzioni nella letteratura

Nell'ambito della riprogrammazione wireless di reti di sensori sono stati sviluppati diversi meccanismi di autenticazione [17, 18, 19, 20, 21] integrati in applicazioni esistenti, in particolare in Deluge[22], l'applicazione per la riprogrammazione wireless inclusa nella distribuzione ufficiale di TinyOS. Questi approcci generalmente condividono la stessa idea di base, ossia dividere l'applicazione da disseminare in blocchi, costruire una catena di hash con questi blocchi e firmare la testa della catena. Un esempio è visibile in figura 4.1. In questo modo la firma digitale garantisce l'autenticità del primo blocco B_0, mentre l'hash contenuto in ogni blocco B_i garantisce l'autenticità del blocco successivo B_{i+1} grazie alla sua proprietà di non invertibilità. La differenza tra le varie applicazioni consiste sostanzialmente nel numero e nella granularità delle catene di hash utilizzate e nell'algoritmo di firma scelto.

In [17, 21] la granularità della divisione in blocchi scende sino al singolo pacchetto. Ogni pacchetto quindi contiene l'hash del pacchetto successivo. Il pregio di questo tipo di soluzione è la capacità di garantire oltre che l'autenticazione anche la protezione da attacchi DoS, in quanto ogni pacchetto ricevuto può essere immediatamente accettato come autentico oppure scartato. Il difetto sostanziale però consiste nella necessità di ricevere i pacchetti nell'ordine esatto in cui sono posizionati nella catena di hash per poterli verificare, e questa limitazione può portare ad un drastico calo delle performance di un algoritmo di disseminazione.

In [18] viene costruita una catena singola come nel caso precedente, però in questo caso i blocchi non sono più singoli pacchetti ma blocchi di trasporto. Questa soluzione ha come pregio la flessibilità data dalla possibilità di ricevere pacchetti fuori ordine, ma dal lato opposto ha anche due difetti. Il primo difetto consiste nel fatto che ricezione dei blocchi deve ancora avvenire in ordine, e anche questa limitazione impatta negativamente sulle performance. Il secondo difetto consiste nella vulnerabilità di questo meccanismo ad attacchi DoS. Un avversario può infatti trasmettere un solo pacchetto corrotto per ogni blocco disseminato, corrompendolo, e costringendo così i nodi sotto attacco a richiedere la ritrasmissione dell'intera pagina. Non è infatti possibile in questa soluzione determinare quale sia il pacchetto corrotto all'interno del blocco.

In [19] viene utilizzata una catena di hash costruita sui transport blocks come nella soluzione precedente, ma in più ogni blocco contiene un hash tree costruito sui pacchetti di dati. In questo modo è possibile l'individuazione del pacchetto o dei pacchetti corrotti nel caso in cui la verifica dell'hash sul blocco rilevi una corruzione. In figura 4.2 è illustrato il processo di costruzione dell'hash tree e il processo di individuazione del pacchetto corrotto. Il pregio di questo sistema è la possibilità di ricevere i pacchetti fuori ordine all'interno di un blocco ma contemporaneamente riuscire ad indi-

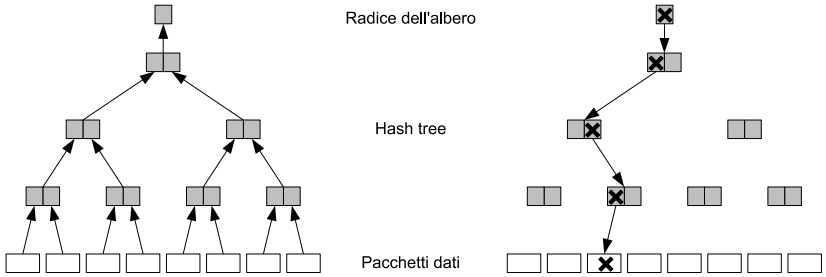


Immagine 4.2: Costruzione e utilizzo di un hash tree per l'individuazione di un pacchetto corrotto.

viduare eventuali pacchetti corrotti, riducendo quindi l'efficacia di attacchi DoS. Il difetto principale è invece costituito dal grande overhead necessario per costruire l'hash tree all'interno di ogni singolo blocco. Anche in questo caso inoltre non sono ammessi blocchi fuori ordine.

In [20] infine viene costruito un hash tree sui pacchetti del primo blocco, e ogni pacchetto è collegato in una catena di hash con il pacchetto corrispondente nel blocco successivo. Anche in questa soluzione si ottiene quindi una protezione da attacchi DoS in quanto ogni pacchetto è immediatamente verificabile. Questo può essere fatto utilizzando l'hash tree se si tratta di un pacchetto del primo blocco, utilizzando invece l'hash incluso nel pacchetto corrispondente del blocco precedente, se si tratta di un blocco successivo al primo. D'altro canto però anche questa soluzione comporta un notevole overhead ed è limitata dalla necessità di ricevere i blocchi in ordine.

4.1.2 Soluzione adottata

Nel nostro caso è stato deciso di adottare una soluzione modulare, ossia si è cercato di raggiungere i tre obiettivi autenticazione, confidenzialità e continuità del servizio in modo indipendente l'uno dall'altro. In questo modo è stato possibile un processo di implementazione semplificato e si è ottenuta inoltre una modularità che permette di abilitare ciascuna delle caratteristiche di sicurezza indipendentemente.

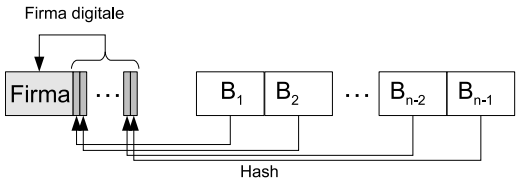


Immagine 4.3: Autenticazione in SecureSynapse .

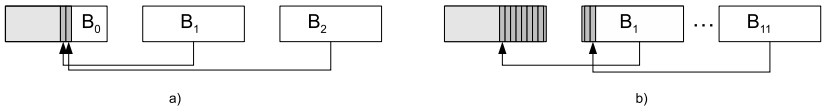


Immagine 4.4: Ottimizzazioni apportate al meccanismo di autenticazione.

Limitandosi quindi per ora al solo obiettivo di autenticazione il sistema adottato, come illustrato in figura 4.3, è il seguente. Per ogni blocco di trasporto viene calcolato un hash, il quale è memorizzato nel primo blocco, che viene firmato. L'autenticità della firma quindi garantisce l'autenticità degli hash che garantiscono a loro volta l'autenticità dei blocchi ricevuti. Questa soluzione, a parità di overhead rispetto alle soluzioni che costruiscono una catena di hash sui blocchi, permette di verificare i blocchi anche se arrivano fuori ordine. Questa capacità permette di incrementare drammaticamente la velocità di disseminazione in Synapse++ . Lo svantaggio di questa soluzione è solamente la necessità di memorizzare gli hash contenuti nel primo blocco fintanto che gli altri blocchi non sono ricevuti, ma questa non si è dimostrata essere una limitazione.

L'algoritmo adottato per la firma digitale è l'algoritmo T-TimeSA, presentato nella sezione 3.3.3.

4.1.3 Ottimizzazioni

A questa idea di base sono state applicate alcune ottimizzazioni implementative per aumentare l'efficienza del sistema. In primo luogo è stato fatto in modo che nel primo blocco possa essere inclusa anche una parte di dati da

disseminare nel caso in cui il blocco non sia riempito completamente dalla firma e dagli hash (figura 4.4a). Questo può permettere in alcuni casi di diminuire di un'unità il numero di blocchi da trasmettere. In secondo luogo è stata aggiunta la possibilità di inserire gli hash anche nei blocchi successivi al primo nel caso in cui nel primo blocco non vi sia spazio a sufficienza (figura 4.4b). In questo modo questa soluzione diventa scalabile rispetto alla dimensione dell'applicazione da disseminare e rispetto alla dimensione dei blocchi di trasporto.

Queste ottimizzazioni nascondono tuttavia una complicazione non del tutto banale. Per entrambe infatti c'è la necessità di conoscere lo spazio rimanente nel primo blocco di trasporto dopo avervi incluso la firma, e quindi c'è la necessità di conoscere la dimensione della firma. Questa però essendo generata dall'algoritmo T-TimeSA, è di lunghezza variabile, a seconda del numero di bit a valore 1 del messaggio da firmare. Il messaggio da firmare è a sua volta l'hash della parte del primo blocco che non comprende la firma, per conoscere il quale è necessario prima conoscere la lunghezza della firma. Si entra quindi in una ricorsione che sembra non essere risolvibile.

La soluzione a questo problema però esiste e consiste nel procedere per tentativi, ipotizzando che la dimensione della firma sia un certo valore, calcolando di conseguenza la dimensione dello spazio rimanente, calcolando la firma sui dati determinati in questo modo, e verificando infine se la dimensione risultante corrisponde a quella ipotizzata. In caso contrario si cambia il valore stimato e si reitera. L'algoritmo corrispondente, utilizzando un livello di sicurezza a 80 bit come implementato in SecureSynapse è illustrato con lo pseudocodice 4.1.

Potrebbe però accadere che comunque non si riesca a trovare una soluzione, pur provando per tutte le lunghezze possibili della firma. Per ovviare a questo problema è stato introdotto nella parte autenticata del primo blocco un byte variabile. Dopo aver ipotizzato una certa lunghezza per la firma

Pseudocodice 4.1 Individuazione della lunghezza della firma

```
for i = 0 to 87 do
  {Sia D la parte di dati da firmare se la firma contenesse i chiavi}
  {Sia h l'hash di D}
  {Aggiungi il checksum ad h}
  {Sia n il numero di bit 1 in h}
  if n = i then
    {Procedi al calcolo della firma su h}
    return SUCCESS
  end if
end for
return FAIL
```

quindi, si procede a determinare la parte di dati necessaria a riempire il blocco, e si calcola infine la firma. Se questa non è della lunghezza attesa, invece di ipotizzare un'altra lunghezza, si prova semplicemente a cambiare il valore di questo byte. In questo modo si incrementa notevolmente la probabilità di indovinare la lunghezza della firma. L'algoritmo è illustrato con lo pseudocodice 4.2.

Pseudocodice 4.2 Individuazione migliorata della lunghezza della firma

```
for i = 0 to 87 do
  for b = 0 to 255 do
    {Sia D la parte di dati, incluso b, da firmare se la firma contenesse i
    chiavi}
    {Sia h l'hash di D}
    {Aggiungi il checksum ad h}
    {Sia n il numero di bit 1 in h}
    if n = i then
      {Procedi al calcolo della firma su h}
      return SUCCESS
    end if
  end for
end for
return FAIL
```

Il funzionamento di questo procedimento può essere spiegato con la

seguente analisi, che per semplicità viene fatta ipotizzando un livello di sicurezza a 80 bit come è implementato in SecureSynapse .

Utilizzando una funzione di hash sicura, l'output della funzione di hash applicato alla porzione di dati che vogliamo firmare sarà un vettore di 80 bit, a cui vengono aggiunti 7 bit di checksum, in cui ciascun bit B_i sarà indipendente dagli altri e assumerà i valori 0 o 1 con uguale probabilità $p_i = 0.5$. Quindi B_i sarà una variabile aleatoria bernoulliana, e il numero di bit a valore 1 in tale vettore sarà di conseguenza una variabile aleatoria binomiale N, con

$$P(N = n) = \frac{\binom{87}{n}}{2^{87}}$$

Quando calcoliamo l'hash come nell'algoritmo 4.1 in pratica generiamo ad ogni iterazione i un vettore casuale e "speriamo" che abbia i bit di valore 1, ma la probabilità che questo avvenga ha distribuzione binomiale, e quindi raggiunge al massimo, per i=43 e i=44 il valore di circa 0.085, meno del 10%.

Quando utilizziamo l'algoritmo 4.2 invece ad ogni iterazione generiamo 256 vettori, e la probabilità che almeno uno di essi abbia il numero desiderato i di bit 1 è

$$P_i = 1 - \left(1 - \frac{\binom{87}{m}}{2^{87}}\right)^{256}$$

e raggiunge per i=43 e 44 praticamente il 100% $(1 - 10^{-10})$. Nell'immagine 4.5 è possibile vedere la differenza tra le due probabilità, la curva (a) indica per ogni iterazione i la probabilità che almeno uno tra i 256 vettori abbia i bit pari a 1, mentre la curva (b) indica la probabilità che un vettore estratto all'iterazione i abbia i bit pari a 1.

Quello che succede quindi è che in realtà la probabilità di generare un vettore con la lunghezza desiderata è molto alta anche per valori minori rispetto alla media, quindi è possibile cercare di minimizzare la lunghezza della firma. Con questa ottimizzazione infatti si ottiene una stabilizzazione ed una riduzione della lunghezza media della firma. Questo si può di-

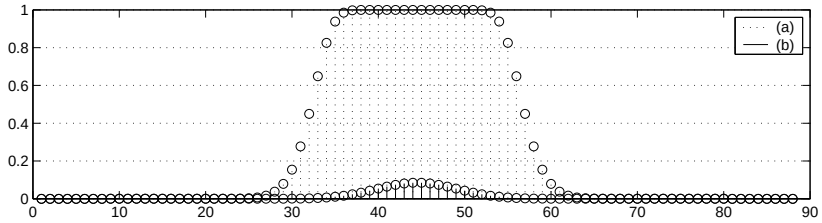


Immagine 4.5: Probabilità di ottenere un vettore della lunghezza desiderata.

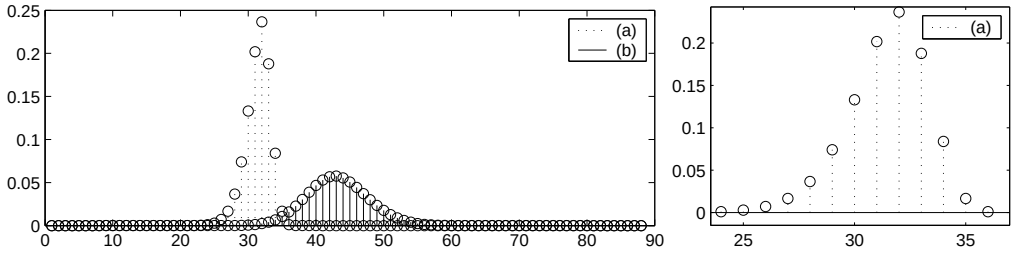


Immagine 4.6: Distribuzione di probabilità della lunghezza della firma

mostrare andando a calcolare la distribuzione di probabilità della lunghezza della firma utilizzando l'algoritmo 4.2.

La probabilità che l'algoritmo dia in uscita una firma contenente i chiavi è pari alla probabilità di ottenere un vettore casuale con i bit 1 all'iterazione i, cioè P_i, moltiplicata la probabilità che nelle iterazioni precedenti non sia stato ottenuto il vettore della lunghezza desiderata. Denominata questa variabile L la sua distribuzione sarà quindi

$$P(L = i) = P_i(1 - P_{i-1})(1 - P_{i-2}) \dots (1 - P_1)(1 - P_0)$$

Nell'immagine 4.6a possiamo vedere la distribuzione di L (curva (a)) calcolata numericamente, confrontata con la distribuzione calcolata analogamente per l'algoritmo 4.1 (curva (b)). Si nota come l'intervallo di valori a maggior probabilità sia più piccolo, e come di conseguenza la probabilità di ciascun valore sia più grande. Nell'immagine 4.6b si nota come i valori a maggiore probabilità siano quelli compresi tra 26 e 35. Questo intervallo

difatti copre il 99.5% circa delle possibilità.

Con questo sistema quindi non solo diventa possibile ottimizzare l'utilizzo dello spazio a disposizione in un blocco, ma viene inoltre ridotto e stabilizzato l'overhead introdotto dalla firma. Grazie a questa proprietà inoltre non è più necessario eseguire tutte le iterazioni dell'algoritmo, per i da 0 a 87, ma è sufficiente limitarsi ai valori dal 26 in poi, in quanto la probabilità di successo nelle iterazioni precedenti è estremamente ridotta. In questo modo si riduce anche il tempo necessario al calcolo della firma stessa.

L'ultima analisi necessaria è relativa alla sicurezza di questa ottimizzazione, ma in questo caso è sufficiente dire che essendo la funzione di hash sicura, dato un hash $h(m + e)$, dove m è il messaggio e e il byte aggiuntivo, trovare una seconda pre-immagine m', con $h(m') = h(m + e)$ è difficile quanto trovare una seconda pre-immagine per il messaggio originale m, cioè trovare m' tale che $h(m') = h(m)$. Questo significa che aggiungere un byte supplementare non è un problema. Inoltre siccome questo byte riduce il numero di chiavi private incluse nella firma, riduce la vulnerabilità del sistema ad attacchi di tipo known-message, in quanto le chiavi private rilasciate con ogni messaggio autenticato saranno minori.

Calcolo del numero di blocchi necessari

Per implementare questa ottimizzazione è necessario conoscere, oltre alla lunghezza della firma, anche il numero di blocchi necessari, in modo da allocare nel primo blocco lo spazio necessario per gli hash. La formula necessaria viene ricavata nel seguente modo.

Sia D la dimensione dell'applicazione da disseminare, B la dimensione del blocco di trasporto, H la dimensione degli hash e F la dimensione della firma. Quello che vogliamo calcolare è il numero di blocchi necessario n e lo spazio per i dati p disponibile nel primo blocco. Le condizioni necessarie

sono:

i) i dati devono essere contenuti in n blocchi e nello spazio p disponibile nel primo blocco.

ii) n deve essere minimo, quindi $n-1$ blocchi non devono bastare

iii) il primo blocco deve essere completamente riempito

Queste condizioni si traducono nel sistema

$$\begin{cases} P + nB \geq D \\ P + (n-1)B < D \\ F + nH + p = B \end{cases}$$

la prima equazione ci dà

$$p = B - F - nH$$

che sostituita nelle altre due disequazioni diventa

$$\frac{D+F-B}{B-H} \leq n < \frac{D+F}{B-H}$$

e quindi

$$\frac{D+F}{B-H} - \frac{B}{B-H} \leq n < \frac{D+F}{B-H}$$

Abbiamo quindi una forma

$$\alpha - \beta \leq n < \alpha$$

dove in questo caso $\beta = \frac{B}{B-H} > 1$ in quanto la dimensione dell'hash è sicuramente positiva e sicuramente minore della dimensione del blocco. Quindi tra $\alpha - \beta$ e α esisterà sicuramente un numero intero, che sarà $\lceil \alpha - \beta \rceil$ e quindi avremo

$$n = \left\lceil \frac{D+F-B}{B-H} \right\rceil$$

Nel caso in cui non rimanesse spazio nel primo blocco, ma anzi gli hash dovessero protrarsi anche nel secondo, la formula rimane valida ma avremo un valore di p negativo, che indica quanto spazio dei blocchi successivi al primo viene utilizzato per gli hash.

4.1.4 Considerazioni su TTimeSA

Una caratteristica dell'algoritmo TTimeSA può essere limitante nell'ambito della riprogrammazione wireless di reti di sensori. Un'assunzione su cui è basato il funzionamento dell'algoritmo così come è stato presentato finora è che trasmettitore e ricevitore mantengano uno stato coerente tra loro, e per far ciò è necessario nella pratica che ogni nodo riceva ed autentichi correttamente il primo blocco di ogni applicazione disseminata, e che ogni volta salvi correttamente le chiavi aggiornate nella memoria non volatile. Se da un lato queste sembrano assunzioni ragionevoli, dall'altro nel contesto delle WSN bisogna sempre considerare l'eventualità che si verifichino guasti imprevisti. In questo caso quindi un guasto che renda impossibile l'aggiornamento delle chiavi in un nodo in una singola disseminazione porterebbe il nodo in uno stato incoerente dal quale non avrebbe più possibilità di uscita. Questa assenza di tolleranza ai guasti potrebbe non essere desiderabile in alcuni casi.

Per ovviare a questo inconveniente quindi la soluzione è allentare il vincolo di sincronizzazione permettendo al nodo di perdere fino ad un massimo di W disseminazioni, modificando l'algoritmo di verifica della firma. Data la chiave pubblica p_i in possesso del nodo e la chiave s_i contenuta della firma, se $p_i \neq h(s_i)$ allora la firma non viene dichiarata subito non valida, ma prima viene controllato anche se $p_i = h(h(s_i)), p_i = h^3(s_i), \ldots, p_i = h^{W+1}(s_i)$. Se una di queste uguaglianze è valida allora significa che il nodo ha perso una riprogrammazione e alcune chiavi non erano sincronizzate.

Questa soluzione però non può essere applicata nel caso in cui si voglia protezione assoluta contro attacchi di tipo known-message, in quanto questa implica un rinnovo delle chiavi utilizzate ad ogni firma, le quali non sono una l'hash dell'altra, ma sono totalmente indipendenti. Nel caso si voglia ottenere una protezione completa contro attacchi known-message è necessario quindi rinunciare alla tolleranza alle disseminazioni perse.

4.2 Confidenzialità

Confidenzialità significa proteggere la riservatezza dell'informazione impedendo che persone non autorizzate ne entrino in possesso. Nel caso della riprogrammazione di reti di sensori garantire la confidenzialità è importante per prevenire che soggetti malintenzionati vengano a conoscenza delle applicazioni installate nei nodi della rete. Se ciò non avviene si ha un conseguente aumento della loro capacità di perpetrare attacchi contro i nodi, attuando per esempio tecniche di reverse engineering sul software di cui entrano in possesso al fine di trovare qualche falla di sicurezza. Se la confidenzialità della riprogrammazione è garantita invece un avversario deve essere in grado di accedere fisicamente ad un nodo per ricavare le stesse informazioni.

4.2.1 Altre soluzioni nella letteratura

Tra le soluzioni già citate per quanto riguarda l'autenticità solamente [21] si occupa di garantire la confidenzialità della riprogrammazione. La soluzione adottata prevede la cifratura di ogni pacchetto utilizzando come chiave l'hash del pacchetto precedente. Il primo pacchetto viene cifrato utilizzando una chiave pre-installata in ogni nodo, che viene aggiornata ad ogni riprogrammazione. Questa soluzione è efficace in quanto garantisce la confidenzialità ed efficiente per la strategia di riprogrammazione utilizzata, che

prevede l'invio sequenziale dei pacchetti non gestendo l'arrivo fuori ordine. Questa soluzione non è tuttavia adottabile in questo lavoro in quanto deve essere gestita la ricezione di pacchetti e di blocchi fuori ordine.

4.2.2 Soluzione adottata

Anche per quanto riguarda la confidenzialità è stato scelto un approccio modulare in modo da permettere la maggiore indipendenza tra le parti del sistema. È stato scelto quindi di implementare questa funzionalità in modo trasparente rispetto all'autenticazione e questo è stato fatto cifrando l'applicazione prima di consegnarla al sistema di disseminazione autenticato, decifrandola infine a disseminazione completata. In questo modo oltre ad ottenere una maggiore modularità, si ha che il tempo necessario alla decifratura dell'applicazione viene allocato alla fine della disseminazione invece che all'interno di ogni ciclo. Si ottiene così un minor tempo di disseminazione in quanto il numero di cicli necessari a concludere la disseminazione non è praticamente mai pari al numero di blocchi da disseminare, a causa di multihop e di errori di trasmissione.

Dettagli implementativi

L'algoritmo di cifratura utilizzato è il block cipher AES utilizzato in modalità OFB. Alla base station vengono utilizzate le librerie bouncycastle [23] per l'implementazione di AES e OFB, mentre nei nodi viene utilizzato il block cipher AES hardware offerto dal CC2420 con un'implementazione di OFB sviluppata appositamente. Durante lo sviluppo sono state incontrate principalmente due difficoltà: l'utilizzo di funzionalità a basso livello del chip radio per sfruttare il block cipher hardware e la necessità di decifrare l'applicazione dopo averla ricevuta completamente.

Per quanto riguarda l'interfacciamento a basso livello con il chip radio

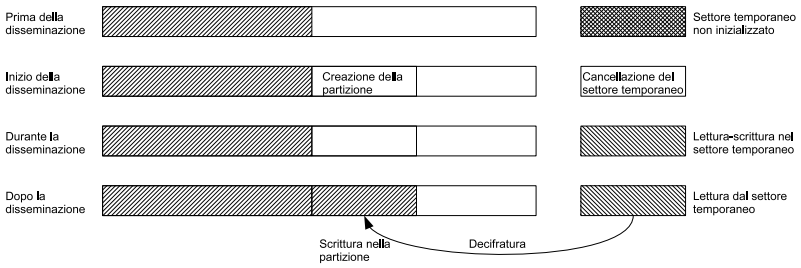


Immagine 4.7: Utilizzo della memoria esterna per la disseminazione di applicazioni cifrate.

è stato dapprima utilizzato del codice sviluppato dal Cryptography and Information Security Lab dell'università di Shangai [10]. Questo codice si è rivelato funzionante se utilizzato da solo, ma ha rivelato in seguito conflitti con il codice che gestisce l'invio e la trasmissione di pacchetti sulla radio, ed è stato quindi corretto. I conflitti nascevano da un'errata configurazione dei registri di controllo delle funzionalità di sicurezza del CC2420.

La necessità di decifrare l'applicazione solo dopo averla ricevuta completamente, se da un lato aumenta la modularità e diminuisce i tempi di disseminazione, dall'altro impone di memorizzare nella memoria permanente l'applicazione cifrata, che dovrà poi essere riletta, decifrata e riscritta. Come già spiegato nella sezione 2.3.2, la memoria flash è una memoria funzionante in modo *erase, write once, read many*, ed è quindi stato implementato un sistema di gestione di partizioni per gestire la memorizzazione e l'aggiornamento delle applicazioni. Utilizzando questo sistema quindi sarebbe necessario creare due partizioni per ogni applicazione disseminata: una nella quale memorizzare l'applicazione cifrata e una nella quale memorizzare l'applicazione in chiaro. Questo comporterebbe un raddoppio nell'utilizzo della memoria permanente da parte del sistema. Per ovviare a questo inconveniente quindi è stato deciso di mantenere nella memoria partizionata solamente le applicazioni in chiaro, e di utilizzare invece un settore di flash distinto per la memorizzazione temporanea dell'applicazione cifrata

durante la disseminazione. Il processo è illustrato nell'immagine 4.7.

4.3 Protezione da attacchi DoS

Un attacco di tipo Denial of Service mira a consumare le risorse di uno o più nodi della rete inviando messaggi contraffatti, al fine di indurre il nodo soggetto all'attacco ad avviare operazioni lunghe o dispendiose in termini di energia. La protezione da attacchi di tipo Denial of Service si ottiene imponendo che operazioni costose dal punto di vista temporale o energetico vengano eseguite solo a fronte di richieste provenienti da fonti autenticate. Per esempio l'inizio della trasmissione di un blocco dovrebbe avvenire solo se è stato verificato che il messaggio richiedente quel blocco sia stato originato da un nodo facente parte della rete.

4.3.1 Altre soluzioni nella letteratura

Le altre soluzioni considerate prevedono in sostanza la verifica di ogni pacchetto dati utilizzando un hash di tale pacchetto trasmesso in precedenza. La protezione offerta da questo meccanismo è di tipo asimmetrico, in quanto è basata su funzioni di hash, e la compromissione di qualche nodo quindi non comporta alcun vantaggio ad un eventuale avversario. Tuttavia lo svantaggio di una tale soluzione consiste nel fatto che siccome i pacchetti devono essere autenticati in precedenza dalla base station, non è possibile per un nodo autenticare pacchetti "al volo", come è invece necessario fare per i messaggi di controllo. In questo modo quindi un avversario potrebbe comunque essere in grado di portare avanti attacchi di tipo DoS semplicemente inviando messaggi di controllo contraffatti, come per esempio richieste di invio di blocchi.

4.3.2 Problematiche dovute ai rateless codes

Si potrebbe comunque pensare di adottare un approccio ibrido, utilizzando gli hash per autenticare i pacchetti dati in modo asimmetrico, e qualche altro meccanismo per autenticare i pacchetti di controllo. Nel nostro caso però si ha una complicazione dovuta all'utilizzo dei Fountain Codes, che rende impraticabile l'autenticazione tramite hash. Utilizzando i Fountain Codes infatti il pacchetto dati trasmesso da un nodo non è più noto a priori alla base station, in quanto è invece il risultato di un'operazione di xor tra diversi pacchetti. Una soluzione a questo problema potrebbe essere l'uso di una funzione di hash omomorfica [24], che è una funzione di hash tale per cui $h(m \oplus n) = h(m) \otimes h(n)$, dove $\oplus$ e $\otimes$ sono due operazioni a scelta. Una tale funzione permetterebbe alla base station di calcolare solamente l'hash dei vari pacchetti, consentendo comunque ai nodi di verificare una qualsiasi loro combinazione. Il problema nell'utilizzo di una funzione omomorfica però è costituito dal carico computazionale introdotto e dalla quantità di codice necessario all'implementazione, che la rendono impraticabile nel nostro caso. Di conseguenza anche l'approccio ibrido è stato tralasciato.

4.3.3 Soluzione adottata

È stato quindi deciso di adottare una soluzione con una sicurezza di tipo simmetrico sfruttando le funzionalità di autenticazione a livello link layer offerte dal chip radio CC2420. In questo caso quindi l'accesso fisico ad un nodo della rete da parte di un avversario dà accesso alle informazioni necessarie per forgiare pacchetti autenticati. Questo tuttavia è stato ritenuto un compromesso accettabile, avendo osservato come i meccanismi di sicurezza asimmetrici siano comunque vulnerabili ad attacchi sui pacchetti di controllo e tenendo presente che comunque l'autenticazione dell'applicazione viene mantenuta.

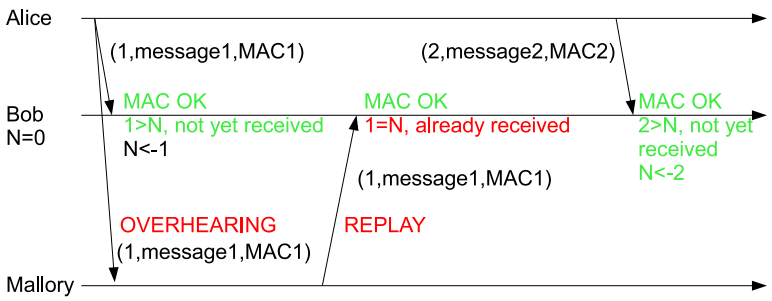


Immagine 4.8: Freschezza dei messaggi in comunicazioni punto-punto

La soluzione adottata prevede quindi l'installazione in ogni nodo prima del deployment di una chiave condivisa, che viene usata per autenticare ogni pacchetto trasmesso tramite un MAC a 32 bit.

4.3.4 Protezione da attacchi replay

L'introduzione di un MAC che garantisca l'autenticità di ogni pacchetto trasmesso non è sufficiente a garantire la protezione da attacchi DoS. Un avversario può infatti ascoltare i messaggi trasmessi e decidere di ritrasmetterne alcuni (attacco replay), che avendo un MAC valido saranno considerati validi e processati regolarmente dai ricevitori. In questo modo potrebbe essere possibile per esempio semplicemente ritrasmettere un messaggio di richiesta di un blocco per perpetrare con successo un attacco. È necessario quindi garantire anche la *freschezza* di un messaggio in modo complementare alla garanzia di autenticità.

In una comunicazione punto-punto garantire la freschezza del messaggio è piuttosto semplice, in quanto è sufficiente che ogni nodo includa un contatore nei messaggi che trasmette, e mantenga il valore dell'ultimo contatore ricevuto, rifiutando messaggi con un contatore minore o uguale (figura 4.8). Questo contatore viene definito *nonce*, che significa *number used once*, cioè un numero utilizzato una volta sola per garantire che il messaggio non è un replay.

In una rete tuttavia non è pensabile di mantenere un contatore per ogni possibile comunicazione punto-punto, quindi le soluzioni applicabili sono principalmente due: la creazione di una *neighbor table*, contenente i contatori per comunicazioni punto-punto con un sottoinsieme di nodi della rete "vicini" (neighbor), oppure l'analisi del protocollo di comunicazione per stabilire un insieme corretto di nonce da utilizzare. La prima soluzione è scalabile e mantiene la semplicità di implementazione di trasmissione e verifica dei pacchetti, ma risulta invece complessa per quanto riguarda la gestione della neighbor table. È stato quindi scelto in questo lavoro di analizzare il protocollo di comunicazione di Synapse++ per progettare un insieme ad-hoc di nonce.

Contatore di sessione

Il primo nonce da inserire nei pacchetti inviati è un contatore di sessione, che viene incrementato dalla base station ad ogni sessione di disseminazione di Synapse++ . Questo contatore viene incluso in ogni pacchetto trasmesso ed è l'unico nonce memorizzato nella memoria permanente di ogni nodo. I nonce descritti di seguito sono inseriti nei pacchetti trasmessi in aggiunta al contatore di sessione, e i loro valori vengono reinizializzati ad ogni nuova sessione di disseminazione. Una rappresentazione di tutti i nonce inseriti nei vari messaggi è data in figura 4.10.

Comunicazione broadcast: messaggio CMD

Il messaggio CMD in Synapse++ è quello più semplice in quanto si tratta di un messaggio trasmesso dalla base station in broadcast e replicato su ogni hop. È quindi gestibile al pari di una comunicazione punto-punto, con la base station che mantiene un contatore dei messaggi trasmessi e ogni nodo che mantiene il valore dell'ultimo ricevuto. Tuttavia è risultato più conve-

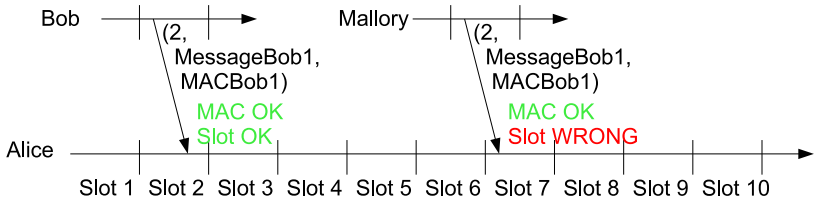


Immagine 4.9: Freschezza dei messaggi utilizzando la sincronizzazione

niente far si che sia direttamente il contatore di sessione ad essere incrementato ad ogni messaggio CMD e utilizzato quindi come nonce. Questo perché alcuni messaggi CMD provocano un reboot del nodo, con una conseguente reinizializzazione di tutti i nonce tranne il contatore di sessione. I messaggi CMD inoltre non sono mai trasmessi nel corso della disseminazione, ma piuttosto ne determinano l'inizio e la fine oppure ordinano l'esecuzione di altre operazioni al di fuori del periodo di disseminazione. I messaggi CMD quindi conterranno il solo contatore di sessione come nonce.

Sfruttamento della sincronizzazione: messaggi ADV e REQ

Come spiegato nella sezione 2.3.2 il tempo in Synapse++ è diviso in *frame* e i nodi vicini sono sincronizzati. All'interno di ogni frame vi sono i periodi ADV, REQ e DATA, all'interno dei quali vengono trasmessi i messaggi corrispondenti. Inoltre se un nodo riceve un messaggio ADV o REQ duplicato all'interno dello stesso slot, questo non avrà influenza sul comportamento del nodo stesso. Grazie a queste caratteristiche è possibile garantire la freschezza inserendo in ogni messaggio ADV o REQ un contatore del numero di *frame*, come illustrato in figura 4.9. In questo modo messaggi di frame precedenti saranno ignorati, mentre messaggi duplicati all'interno dello stesso frame come detto non avranno impatto sul comportamento del nodo.

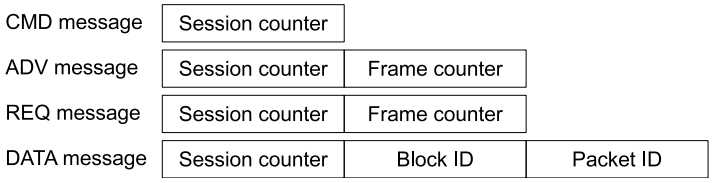


Immagine 4.10: Nonce inseriti nei vari messaggi di Synapse++

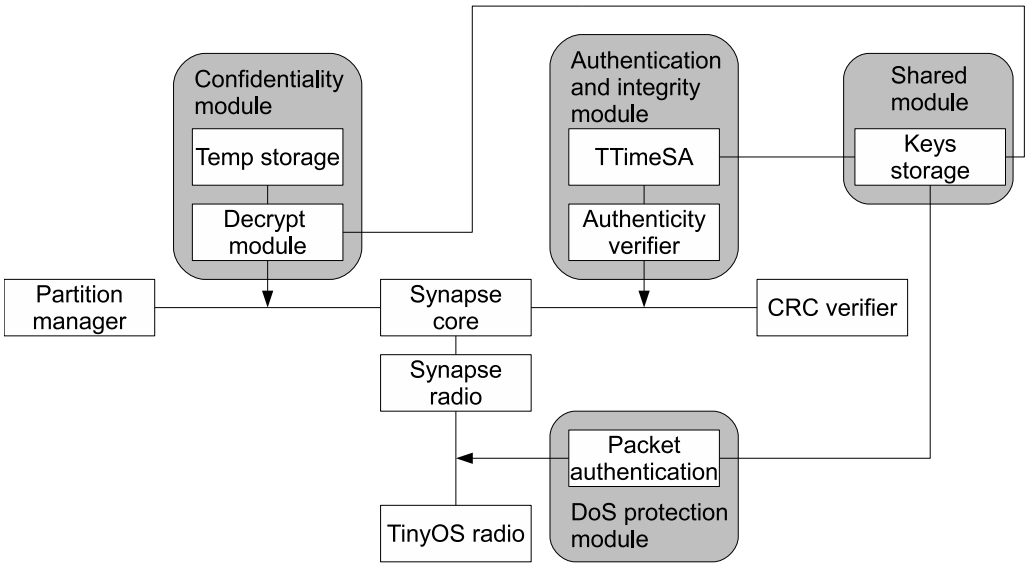


Immagine 4.11: Architettura del sistema

Riconoscimento di messaggi duplicati: messaggi DATA

Nei messaggi data sono contenuti, oltre ai dati da trasportare, anche un identificativo di blocco e un identificativo di pacchetto. Grazie a queste informazioni è possibile quindi scartare pacchetti duplicati o relativi ad un blocco diverso da quello che si sta ricevendo. Per questi pacchetti quindi non è necessario aggiungere dei nonce aggiuntivi.

4.4 Architettura del sistema

Nell'immagine 4.11 è illustrato come i vari moduli di sicurezza sono stati integrati in Synapse++ .

4.5 Risultati ottenuti

I benefici introdotti dall'inserimento di meccanismi di sicurezza nel sistema sono pagati con un degrado delle performance. Nelle prossime sezioni da un lato verrà misurato questo degrado, e dall'altro saranno valutati i benefici introdotti rispetto al sistema originale. I test sono stati condotti su una porzione di un testbed realizzato al Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, costituita da 50 nodi.

4.5.1 Performance autenticazione

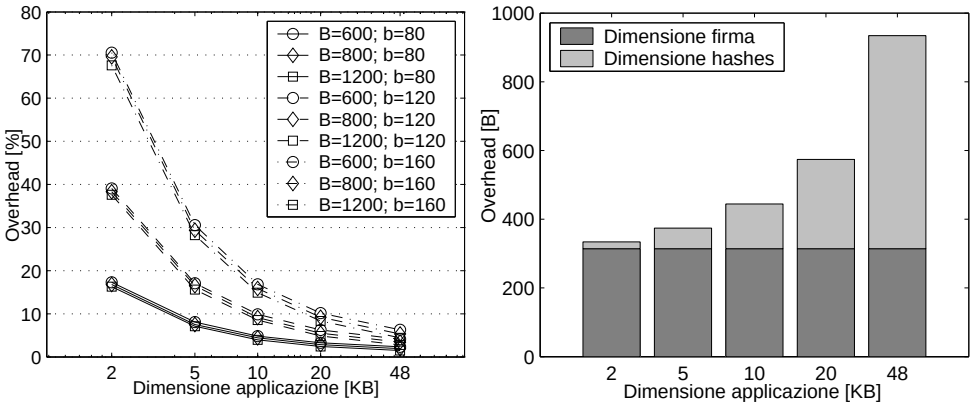
L'introduzione del modulo di autenticazione penalizza le prestazioni del sistema in quanto vengono introdotti i seguenti overhead:

- dati aggiuntivi costituiti dalla firma e dagli hash
- tempo di verifica della firma nel primo blocco
- tempo di verifica dell'hash per ogni blocco successivo al primo

L'entità dell'overhead introdotto varia in funzione dei seguenti parametri:

- dimensione del blocco di trasporto
- livello di sicurezza desiderato

In figura 4.12a possiamo vedere come varia la dimensione dei dati aggiuntivi al variare della dimensione dell'applicazione da disseminare, della dimensione del blocco e del numero di bit di sicurezza desiderati. I risultati sono stati ricavati analiticamente, in particolare la dimensione della firma è stimata con il procedimento utilizzato nella sezione 4.1.3. Dall'immagine si nota come l'overhead in percentuale diminuisca al crescere della dimensione dell'applicazione. Si vede inoltre che il numero di bit di sicurezza ha un maggior peso rispetto alla dimensione del blocco. Di fatto un raddoppio del



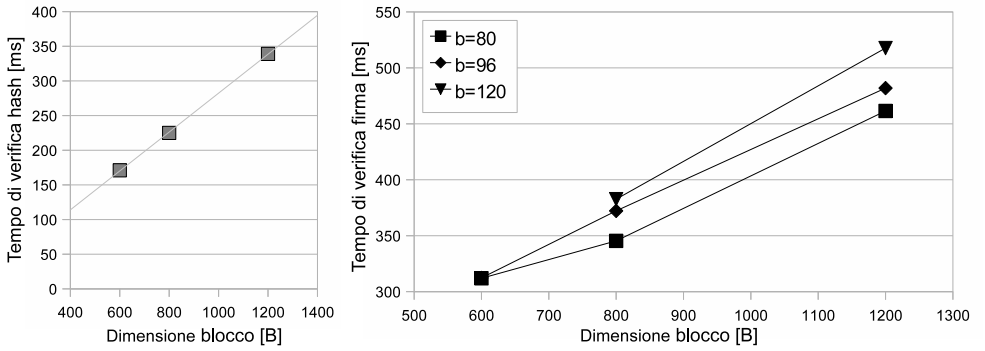
(a) (b)

Immagine 4.12: Dimensione dei dati aggiuntivi necessari all'autenticazione

numero di bit di sicurezza comporta all'incirca una quadruplicazione dello spazio aggiuntivo necessario. Questo si spiega considerando che un aumento di un fattore due è dovuto al raddoppio della dimensione di tutti gli hash, e un altro aumento di un fattore due è dovuto al raddoppio del numero di hash contenuti nella firma. Raddoppiando la dimensione del blocco invece sarà solo dimezzato il numero degli hash di autenticazione dei blocchi, mentre la dimensione della firma rimarrà invariata. In figura 4.12b invece si nota come la dimensione della firma costituisca una parte importante dell'overhead complessivo.

Ricordando che in SecureSynapse viene scelto un livello di sicurezza di 80 bit e la dimensione del blocco in Synapse++ è di 800 byte, dall'immagine 4.12a si può notare come l'overhead introdotto sia inferiore al 10% per applicazioni di soli 5KB, mentre per applicazioni di almeno 10KB risulta essere inferiore al 5%.

Per quanto riguarda l'overhead temporale in figura 4.13a è riportato il tempo necessario alla verifica della firma, al variare della dimensione del blocco e del livello di sicurezza. In figura 4.13b invece è riportato il tempo necessario alla verifica dell'hash di un blocco al variare della dimensione del



(a) (b)

Immagine 4.13: Tempi aggiuntivi necessari all'autenticazione

blocco. Il livello di sicurezza in questo caso non è influente in quanto l'hash viene sempre calcolato a 128 bit, e solo successivamente viene troncato al numero di bit richiesto. Il punto relativo al blocco di dimensione 600B con livello di sicurezza 128 bit non è presente in quanto tali parametri danno origine ad una firma di dimensione maggiore del blocco e quindi non sono applicabili nella realtà. Dalle immagini si nota come il tempo necessario alla verifica dell'hash sia lineare nella dimensione del blocco. La retta che interpola i dati ha equazione $t = 0.28B + 1.71$, dove t è il tempo e B la dimensione del blocco. Per un blocco di 800 byte, come implementato attualmente in SecureSynapse , il tempo di verifica dell'hash risulta quindi di 225ms. Il tempo necessario alla verifica della firma invece dipende sia dalla dimensione del blocco, che determina la quantità di dati da firmare, sia dal livello di sicurezza, che determina invece la dimensione della firma. Anche in questo caso comunque come ci si aspetta il tempo di verifica aumenta con il livello di sicurezza e con la dimensione del blocco. Nell'implementazione attuale di SecureSynapse il livello di sicurezza è di 80 bit e la dimensione del blocco di 800 byte, quindi il tempo necessario alla verifica della firma è mediamente (in quanto la lunghezza della firma può variare) di 350 ms.

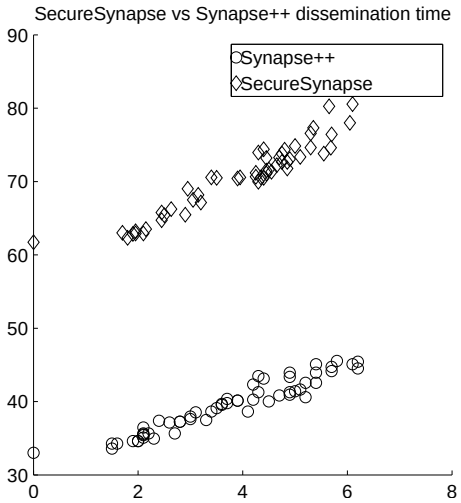


Immagine 4.14: Tempo complessivo di disseminazione in multihop

Impatto complessivo

Per integrare le funzioni di autenticazione in Synapse++ quindi sono stati tenuti in considerazione questi overhead, andando ad aumentare la durata dei frame. Considerando che la verifica della firma richiede più tempo della validazione di un blocco e che viene eseguita solo alla prima iterazione, la durata del frame è stata incrementata solamente del tempo necessario alla verifica dell'hash del blocco. In questo modo si avrà che la ricezione del primo blocco occuperà due frame, mentre la ricezione dei blocchi successivi occuperà regolarmente un solo frame.

Questa scelta tuttavia va ad impattare negativamente sul meccanismo di pipelining di Synapse++ , in quanto come illustrato nella sezione 2.3.2 un nodo acquisisce la priorità di trasmissione nel frame successivo alla ricezione di un blocco, ma dato che la validazione della firma si protrae appunto nel frame successivo la priorità acquisita non sarà sfruttata. Questo conflitto con il sistema di pipelining causa quindi un rallentamento, visibile in figura 4.14. L'implementazione delle modifiche necessarie ad un ripristino della funzionalità di pipelining è stata demandata a sviluppi futuri.

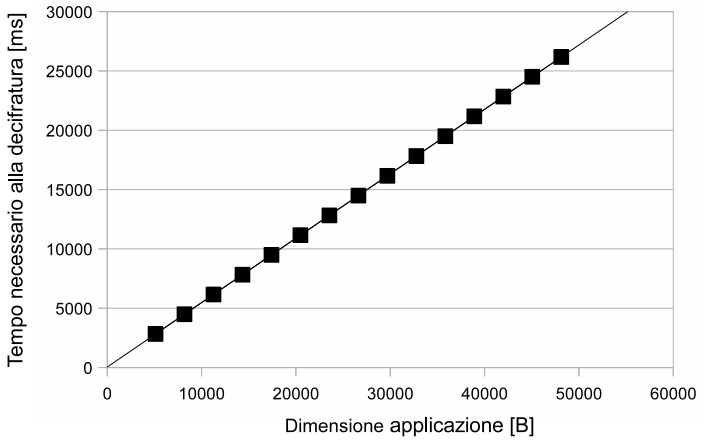


Immagine 4.15: Overhead dovuto alla cifratura

4.5.2 Performance confidenzialità

Per quanto riguarda la confidenzialità l'overhead introdotto è costituito in primo luogo dai 16 byte del vettore di inizializzazione per la modalità operativa OFB che devono essere disseminati insieme con l'applicazione per permettere ai nodi di decifrarla. In secondo luogo va considerato anche l'overhead temporale introdotto dal processo di decifratura dell'applicazione a disseminazione conclusa. In figura 4.15 si può vedere come questo tempo cresca proporzionalmente all'aumentare della dimensione dell'immagine disseminata. L'andamento ricavato dai dati è $t = 0.54d + 40$, dove t è il tempo necessario alla decifratura e d la dimensione dell'applicazione. Il tempo necessario per decifrare un'applicazione della dimensione massima per la piattaforma telosb, ovvero 48KB, è di circa 25 secondi.

4.5.3 Performance protezione da DoS

La protezione da attacchi DoS influisce sul tempo di trasmissione e ricezione di ogni singolo pacchetto. Vi è infatti un aumento del payload dato dal MAC, dai nonce e da altri byte di controllo, raffigurato nell'immagine 4.16. Inoltre vi è anche un overhead dato dal tempo necessario all'inserimento nel

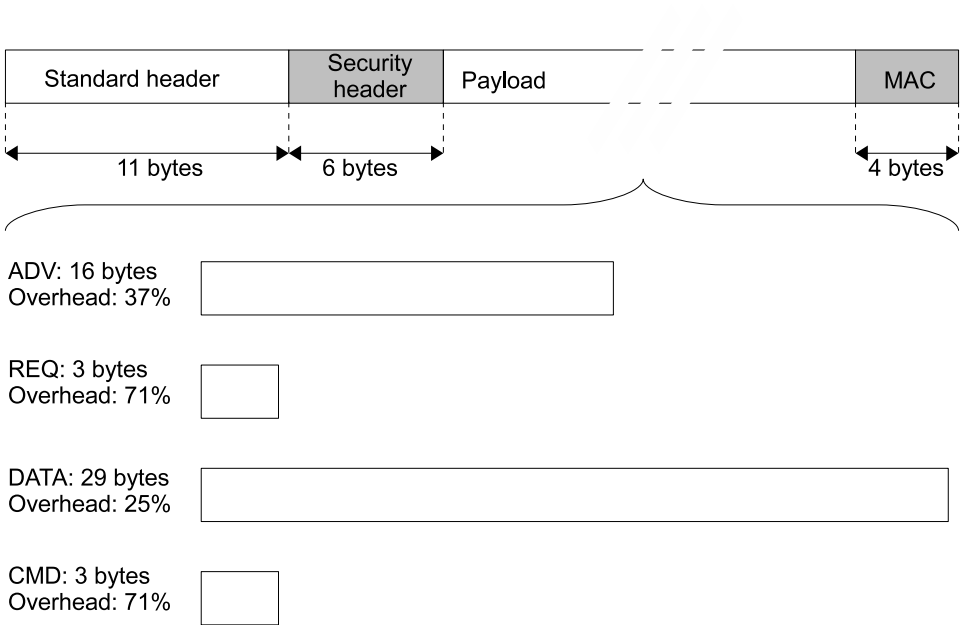


Immagine 4.16: Overhead nei pacchetti per la protezione da DoS

pacchetto dei dati necessari per l'autenticazione dal lato del trasmettitore, dato invece dal tempo necessario alla verifica dei dati di autenticazione al lato ricevitore. La valutazione di questi fattori è stata fatta sperimentalmente e l'overhead complessivo è risultato essere di circa 250ms per frame.

4.5.4 Performance complessiva

Nell'immagine 4.17 è possibile vedere il tempo di disseminazione utilizzando tutte le funzionalità di sicurezza: autenticazione, confidenzialità e protezione da DoS. Il tempo medio risulta circa il doppio di quello necessario a Synapse++ , ma come già spiegato nella sezione 4.5.1 una buona parte di questo aumento è dovuta al malfunzionamento del meccanismo di pipelining una volta introdotta la funzionalità di autenticazione.

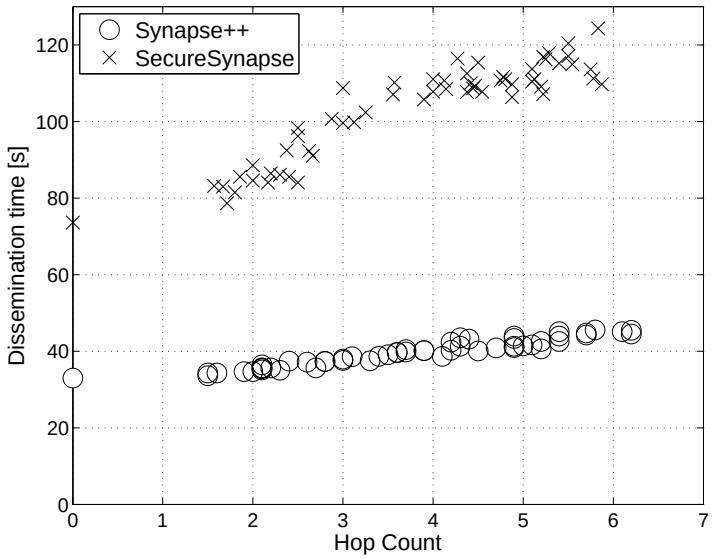


Immagine 4.17: Tempo di disseminazione con tutte le funzionalità di sicurezza

4.5.5 Dimensione del codice

In figura 4.18 sono riportate le occupazioni in ROM e RAM dei vari moduli e delle loro combinazioni. Le immagini 4.18a e 4.18c mostrano i valori assoluti di overhead, mentre le immagini 4.18b e 4.18d permettono di avere una visione più intuitiva dell'overhead introdotto, visualizzando assieme la dimensione di ROM e RAM di Synapse++ e dei moduli introdotti.

Sia in termini di ROM che di RAM il modulo più costoso risulta essere quello che garantisce l'autenticazione, seguito dal modulo di protezione da DoS, e infine dal modulo di cifratura. Si nota inoltre come la combinazione dei vari moduli comporti una combinazione praticamente additiva della RAM, mentre la dimensione della ROM di due moduli combinati è abbastanza inferiore alla somma della dimensione dei singoli moduli.

L'overhead complessivo dei tre moduli combinati risulta essere del 42% per quanto riguarda la ROM, e del 37% per quanto riguarda la RAM.

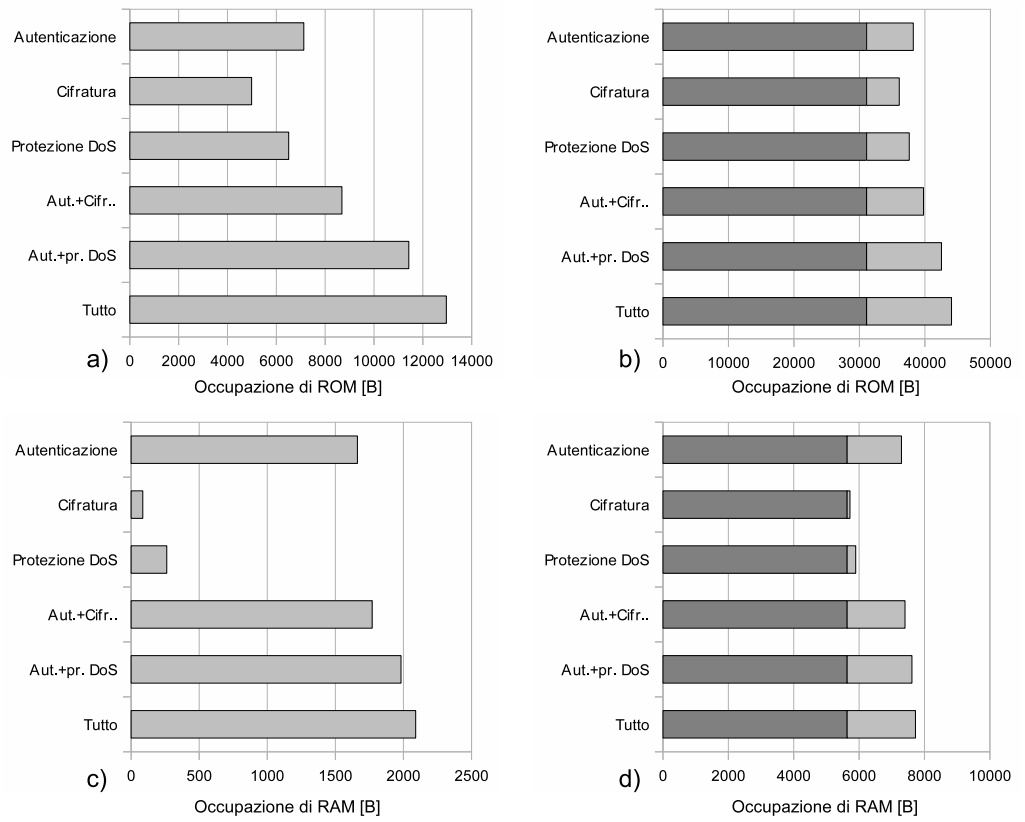


Immagine 4.18: Utilizzo delle risorse della piattaforma

4.5.6 Impatto di possibili attacchi

Per valutare il meccanismo di autenticazione è stato sperimentato l'impatto di un attacco di corruzione dell'applicazione disseminata in Synapse++ e in SecureSynapse . In Synapse++ l'esito di tale attacco è stato valutato analizzando la percentuale di disseminazioni che un certo numero di attaccanti è in grado di far fallire, corrompendo l'applicazione disseminata. Nell'immagine 4.19 è possibile vedere la distribuzione di probabilità della percentuale di disseminazioni corrotte al variare del numero di attaccanti. Si nota come all'aumentare del numero di attaccanti la probabilità di far fallire una certa percentuale di disseminazioni aumenti. In particolare la probabilità di corrompere almeno il 50% delle disseminazioni è di circa il

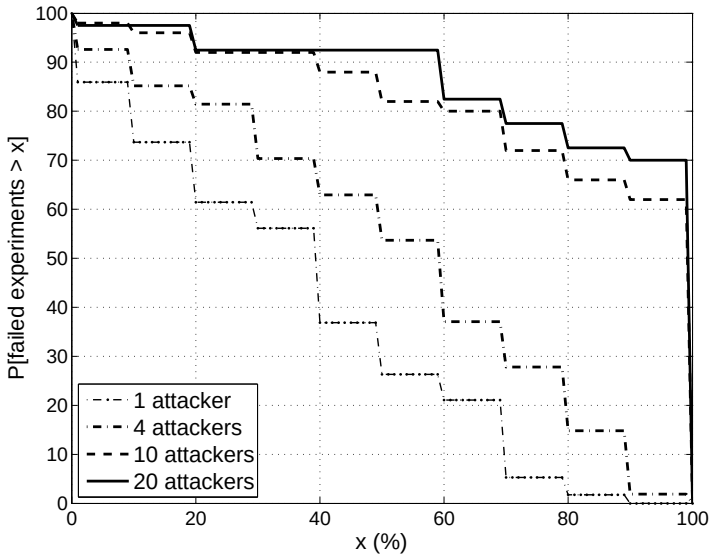


Immagine 4.19: Attacco di corruzione dell'applicazione disseminata

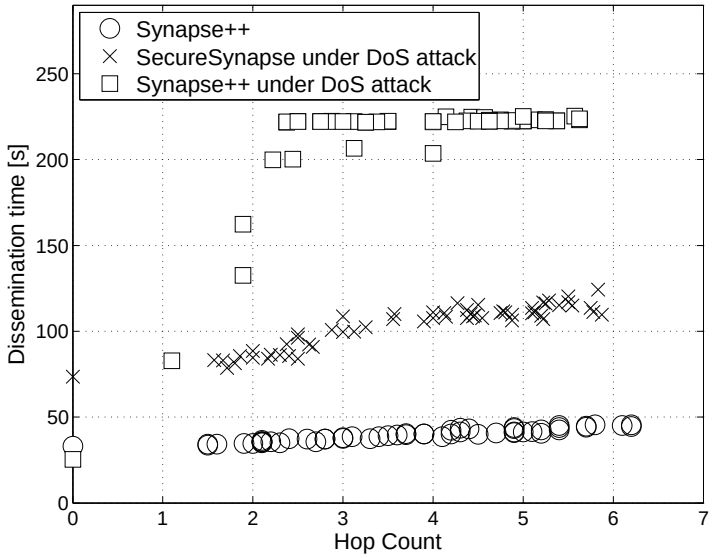


Immagine 4.20: Attacco di tipo Denial of Service

25% con un solo attaccante, e arriva ad oltre il 90% con 20 attaccanti. In SecureSynapse , non essendo possibile la corruzione dell'applicazione disseminata, un tale attacco non provoca alcun danno.

Per quanto riguarda gli attacchi di tipo DoS, in SecureSynapse il tempo di disseminazione sotto attacco è praticamente lo stesso tempo necessario

in assenza di attacchi. Un attacco perpetrato senza che l'avversario abbia avuto accesso fisicamente a qualche nodo si riduce infatti ad un attacco di jamming. I pacchetti inviati dall'avversario provocano un rallentamento nell'attività dei nodi dovuto solamente all'occupazione del canale radio e al tempo necessario a verificare la non validità del pacchetto. Nell'immagine 4.20 è riportato il tempo di disseminazione di SecureSynapse e di Synapse++ sotto un attacco di tipo DoS da parte di un solo nodo, comparato con il tempo di disseminazione di Synapse++ . Si nota come Synapse++ sia molto vulnerabile a questo attacco, dato che con un solo attaccante si riesce a far aumentare il tempo di disseminazione di più del 300%.

Per quanto riguarda la confidenzialità l'unico modo che ha un avversario per entrare a conoscenza delle informazioni trasmesse è l'accesso fisico al nodo, in modo da recuperare le chiavi segrete che permettono la decifratura.

Capitolo 5

Conclusioni e possibili sviluppi

In questa tesi sono state presentate alcune tecniche di sicurezza sfruttate nell'ambito della riprogrammazione wireless di reti di sensori, per garantire l'autenticità e la confidenzialità dell'applicazione disseminata e per impedire ad un avversario di concludere con successo attacchi di tipo Denial of Service.

I meccanismi di sicurezza sono stati integrati in Synapse++ , un sistema per la riprogrammazione wireless basato su rateless Fountain Codes, che permette un recupero efficiente degli errori anche in reti ad alta densità. Il sistema così ottenuto è il primo a combinare i benefici dati dall'utilizzo dei codici rateless con tecniche di sicurezza che garantiscono una protezione così completa.

Per raggiungere gli obiettivi preposti è stato necessario prestare una grande attenzione alle limitazioni imposte dalle piattaforme hardware utilizzate. La limitazione più restrittiva in particolare è risultata essere la memoria ROM, che contiene il codice eseguibile dell'applicazione di disseminazione. Per far fronte a questo problema è stato necessario prestare la massima attenzione alla dimensione del codice sviluppato, dovendo accettare anche alcuni compromessi tra funzionalità e codice necessario, come

nel caso del sistema di protezione da attacchi DoS. In altri casi è stato fatto uso di algoritmi progettati appositamente per piattaforme dalle risorse limitate, come l'algoritmo TTimeSA. È stato utile nell'affrontare questo problema anche la struttura modulare del sistema sviluppato, che ha permesso uno sviluppo e quindi una verifica ed un debug indipendente delle varie parti, limitando di molto la necessità di eseguire il debug sul sistema nel suo complesso. Questa operazione si è rivelata infatti molto difficoltosa in quanto il sistema completo sfruttava praticamente al massimo la disponibilità di ROM, e risultava quindi impossibile inserire il codice necessario al debug.

Nonostante l'attenzione riposta nell'ottimizzare il codice, ulteriori sviluppi possono essere in primo luogo una sua analisi approfondita ed una modularizzazione più spinta, che possono portare a riduzioni ulteriori della memoria ROM utilizzata. Un altro sviluppo proposto è l'adattamento del meccanismo di pipelining alle esigenze di autenticazione dell'applicazione, in modo da ottenere un notevole miglioramento del sistema anche in disseminazioni multi-hop.

Bibliografia

[1] Volcano sensorweb. http://ai.jpl.nasa.gov/public/projects/sensorweb/. [pag. 6]

[2] NASA Jet Propulsion Laboratory. Nasa goes inside a volcano, monitors activity. http://www.jpl.nasa.gov/news/news.cfm?release=2009-117, 08 2009. [pag. 6]

[3] GE Aviation. Reducing wiring complexity: Ge aviation supports the adoption of witnesss with its wireless sensor technology. http://www.geaviationsystems.com/News/Archive/2009/Reducing-W/index.asp, 2009. [pag. 7]

[4] Lachesi. Progetto guarini. http://www.lachesi.com/jsp/it/monitoring_project_detail_it.jsp?id=22. [pag. 7]

[5] Sensys Networks. Arterial travel time system. http://www.sensysnetworks.com/traveltime. [pag. 8]

[6] Toumaz Technology Ltd. Ultra low power intelligent sensor interface and transceiver platform. http://www.toumaz.com/public/page.php?page=sensium_intro. [pag. 9]

[7] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. AES algorithm original submission, September 1999. [pag. 13]

[8] Chipcon AS SmartRF. *CC2420 Preliminary Datasheet (rev 1.2)*. 06 2004. [pag. 13]

[9] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, New York, NY, USA, 2009. [pag. 17, 18]

[10] SJTU CIS Lab. The standalone aes encryption of cc2420. http://cis.sjtu.edu.cn/index.php/The_Standalone_AES_Encryption\discretionary{-}{}{}_of_CC2420_(TinyOS_2.10_and_MICAz). [pag. 22, 69]

[11] Michele Rossi, Nicola Bui, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, and Michele Zorzi. SYNAPSE++: Code Dissemination in Wireless Sensor Networks using Fountain Codes. *Transactions on Mobile Computing*, 2010. Accepted for publication. [pag. 26, 28]

[12] Michele Rossi, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, Albert F. Harris III, and Michele Zorzi. SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks using Fountain Codes. In *IEEE SECON*, San Francisco, CA, US, June 2008. [pag. 26]

[13] Vlastimil Klima. Tunnels in hash functions: Md5 collisions within a minute (extended abstract). Technical report, 2006. [pag. 37]

[14] Ronald Rivest Shafi Goldwasser, Silvio Micali. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281308, Apr. 1988. [pag. 39]

[15] Ralph C. Merkle. A certified digital signature. *CRYPTO '89: Proceedings on Advances in cryptology*, pages 218–238, 1989. [pag. 40]

[16] Osman Ugus, Dirk Westhoff, and Jens-Matthias Bohli. A ROM-friendly Secure Code Update Mechanism for WSNs Using a Stateful verifier T-time Signature Scheme. In *ACM WiSec*, Zurich, Switzerland, March 2009. [pag. 44, 46]

[17] Prabal K. Dutta, Jonathan W. Hui, David C. Chu, and David E. Culler. Securing the deluge Network programming system. In *IEEE IPSN*, Nashville, TN, USA, April 2006. [pag. 56, 57]

[18] Patrick E. Lanigan, Rajeev Gandhi, and Priya Narasimhan. Sluice: Secure Dissemination of Code Updates in Sensor Networks. In *IEEE ICDCS*, Reading, UK, May 2006. [pag. 56, 57]

[19] Jing Deng, Richard Han, and Shivakant Mishra. Secure code distribution in dynamically programmable wireless sensor networks. In *IEEE IPSN*, Nashville, TN, USA, April 2006. [pag. 56, 57]

[20] Sangwon Hyun, Peng Ning, An Liu, and Wenliang Du. Seluge: Secure and DoS-Resistant Code Dissemination in Wireless Sensor Networks. In *IEEE IPSN*, St. Louis, MO, USA, April 2008. [pag. 56, 58]

[21] H. Tan, D. Ostry, and S. Jha J. Zic. A Confidential and DoS-Resistant Multi-hop Code Dissemination Protocol for Wireless Sensor Networks. In *ACM WiSec*, 2009. [pag. 56, 57, 67]

[22] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM. [pag. 56]

[23] Bouncycastle. `http://www.bouncycastle.org/`. [pag. 68]

[24] Maxwell N. Krohn, Michael J. Freedman, and David Mazieres. On-the-fly verification of rateless erasure codes for efficient content distribution. In *In Proceedings of the IEEE Symposium on Security and Privacy*, pages 226–240, 2004. [pag. 71]

Appendice

Lista dei Simboli e Abbreviazioni

Abbreviazione	Descrizione	Definizione
WSN	Wireless Sensor Network	page 1
DoS	Denial of Service	page 3
ISM	Industrial, Scientific and Medical	page 12
USB	Universal Serial Bus	page 12
CTR	Counter	page 13
MAC	Message Authentication Code	page 48
CBC-MAC	Cipher Block Chaining MAC	page 13
CCM	Counter with CBC-MAC	page 13
SPI	Serial Peripheral Interface	page 14
SVN	Subversion	page 22
CCA	Clear Channel Assessment	page 25
ARQ	Automatic Repeat-reQuest	page 26
NACK	Not Acknowledge	page 26
SHA1	Secure Hash Algorithm 1	page 38
MIC	Message Integrity Code	page 49
AES	Advanced Encryption Standard	page 53

Lista delle immagini