View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Thesis and Dissertation Archive - Università di Pisa

UNIVERSITÀ DI PISA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

PROGETTO E REALIZZAZIONE DI UN PROTOCOLLO DI SICUREZZA PER LA VERIFICA REMOTA DELL'INTEGRITÀ DELLE APPLICAZIONI IN UNA RETE DI SENSORI

Candidato:

Zoran Curto ______________________

Relatori:

Prof. Gianluca Dini ______________________

Ing. Alessio Bechini ______________________

Prof. Marco Avvenuti ______________________

Anno Accademico 2005/2006

Abstract

Con riferimento alla sicurezza in una rete di sensori, gli obiettivi di questa tesi sono la progettazione e la realizzazione di un protocollo per la verifica dell'integrità delle applicazioni eseguite da un generico sensore. Le reti di sensori vengono spesso installate in ambienti vasti e non sorvegliati. I sensori posso essere catturati da un avversario che può riprogrammarli e reintrodurli nella rete allo scopo di attaccare i servizi di rete oppure i dati che transitano nella rete stessa.

Questo tipo di protocollo permette di creare delle reti di sensori composte soltanto da nodi 'affidabili', verificando l'integrità delle applicazioni installate sui sensori.

Il protocollo, di tipo *challenge-response*, si basa su di una funzione di *hash* [3], studiata appositamente per sistemi *embedded* poveri di risorse, grazie alla quale è possibile effettuare il calcolo della *hash* sul contenuto della memoria di un sensore ogni qual volta sia necessario verificarne l'integrità. Grazie a questo protocollo è possibile contrastare attacchi che prevedono la cattura, il *reverse-engineering* e la *riprogrammazione* dei sensori con una soluzione semplice, efficiente e che non necessita di hardware aggiuntivo (*software-only*).

In questo lavoro vengono analizzati i principali requisiti che deve possedere un protocollo per la verifica dell'integrità e vengono analizzati alcuni protocolli già esistenti [1],[2],[3]. Tra questi, il protocollo PIV (*Program Integrity Verification*) [3] è sicuramente il più efficiente. Purtroppo esso è vulnerabile perché soggetto ad attacchi di parallelizzazione (*parallel attack*). Dopo aver analizzato la vulnerabilità di PIV, ne viene proposta una modifica (PIVm) per aumentarne il livello di sicurezza e renderlo resistente ad attacchi di tipo *parallel attack*.

Del protocollo proposto è stato realizzato un prototipo per sensori della famiglia Moteiv Tmote Sky e ne sono state valutate le prestazioni. Con un overhead computazionale di +11,5% rispetto alla funzione PIV originale, la funzione modificata (PIVm) è in grado di calcolare una *hash* in 3,4 secondi. Dal punto di vista dell'occupazione di memoria RAM, PIVm necessita di soli 314 bytes in più rispetto a PIV (ovvero il 18% in più). Infine, grazie alle modifiche introdotte è possibile limitare drasticamente la quantità di dati scambiati tra server e sensori e risparmiare quindi il 97,8% dei costi di trasmissione.

Ai miei genitori

Sommario

Gli obiettivi di questa tesi sono la progettazione e la realizzazione di un protocollo di sicurezza per la verifica remota dell'integrità delle applicazioni in una rete di sensori. In particolare questo protocollo permette di creare delle reti di sensori composte soltanto da nodi ‘affidabili’ verificando l’integrità delle applicazioni installate sui sensori. Il cuore del protocollo è una funzione di *hashing* studiata appositamente per CPU povere di risorse, grazie alla quale è possibile effettuare il calcolo di una *hash* sul contenuto della memoria, ogni qual volta sia necessario verificare l’integrità di un sensore. La funzione di *hashing* viene inizializzata con un seme, che viene mandato dal server di verifica dell’integrità. Grazie a questo protocollo è possibile prevenire attacchi di cattura, *reverse-engineering* e riprogrammazione dei sensori con una soluzione semplice, efficiente e che non necessita di hardware aggiuntivo.

Nel primo capitolo si introduce la problematica relativa alle reti di sensori collocate in ambienti ostili e si fa una rapida panoramica sullo stato dell’arte.

Nel secondo capitolo si introducono gli aspetti fondamentali che caratterizzano le reti di sensori, facendo particolare riferimento alle problematiche che riguardano la scarsità di risorse dei sensori nonché quelle relative alla sicurezza. Innanzitutto viene fatta una breve analisi sugli ambiti di applicazione delle reti di sensori. Successivamente vengono descritti i requisiti relativi alla sicurezza in una rete di sensori e infine vengono illustrati i vincoli e le limitazioni introdotte sia dai singoli sensori che dalle reti nel loro insieme.

Nel terzo capitolo viene introdotta una tassonomia per classificare gli avversari e gli attacchi a cui può essere soggetta una rete di sensori. Successivamente viene fatta un’analisi di questi attacchi e viene stabilito che il protocollo di sicurezza che si sta progettando deve proteggere la rete da attacchi di manomissione e impersonificazione dei sensori. Successivamente vengono analizzati i protocolli di verifica dell’integrità già esistenti e vengono fatti alcuni confronti tra di essi, per mostrarne le differenze.

Nel quarto capitolo vengono analizzate in dettaglio le caratteristiche di sicurezza che deve possedere il protocollo che si vuole realizzare e vengono riassunte alcune

caratteristiche salienti del sensore Moteiv Tmote-Sky nonché del sistema operativo TinyOS, sui quali verrà realizzato il prototipo del protocollo.

Nel quinto capitolo si espone il protocollo PIV, presentato in [3], nonché la sua vulnerabilità. Successivamente viene mostrato il protocollo PIVm, che rappresenta una variante del protocollo PIV a cui sono state apportate tre modifiche. La prima modifica ne aumenta il livello di sicurezza, mentre le altre due modifiche ne aumentano l'efficienza.

Nel sesto capitolo vengono mostrati i moduli software realizzati per TinyOS, le loro interconnessioni, e le procedure per effettuare il calcolo della funzione di *hash* nonché per l'attraversamento pseudo-casuale della memoria.

Nel settimo capitolo viene illustrata l'applicazione server, realizzata in linguaggio Java, che è stata creata per implementare il server PIVSm.

Nell'ottavo capitolo vengono descritte le tecniche che sono state usate per effettuare i test delle due applicazioni.

Infine, nel nono capitolo, vengono mostrati in modo sintetico i risultati ottenuti in questo lavoro di tesi.

tamper resistance

è l’arte di creare un sistema tale che esso non possa essere modificato da un avversario senza che l’attacco venga scoperto.
E’ una proprietà simile alla *fault tolerance*, ma mentre quest’ultima è la capacità di scoprire un errore che si è creato per cause accidentali, nel caso della tamper resistance si presuppone la presenza di un avversario senziente.

1. INTRODUZIONE **11**
1.1 Descrizione del problema 11
1.2 La verifica dell'integrità delle applicazioni 12
2. LA SICUREZZA NELLE RETI DI SENSORI **14**
2.1 La missione delle reti di sensori 14
2.1.1 Protezione del perimetro di un'area *14*
2.1.2 Sorveglianza remota di un'area *15*
2.1.3 Monitoraggio ambientale *15*
2.2 I vincoli energetici 16
2.2.1 Trasmissione e ricezione di dati *16*
2.3 Ciclo di vita dei sensori 17
2.3.1 Produzione *17*
2.3.2 Stoccaggio *18*
2.3.3. Inizializzazione e configurazione *18*
2.3.4 Collocazione e funzionamento *18*
2.3.5 Completamento della missione *19*
2.4 Requisiti di sicurezza 19
2.4.1 Confidenzialità *19*
2.4.2 Autenticità/integrità *19*
2.4.3 Freschezza *20*
2.4.4 Disponibilità *20*
2.4.5 Accessibilità *20*
2.4.6 Auto-organizzazione *21*
2.5 Vincoli 21
2.5.1 Vincoli imposti dai nodi *21*
2.5.1.1 Riserve energetiche/Durata della batteria *22*
2.5.1.2 Consumo energetico indotto dagli algoritmi di sicurezza *22*
2.5.1.3 Consumo energetico dovuto alla trasmissione di dati relativi alla sicurezza *23*
2.5.1.4 Cicli sonno-veglia *23*
2.5.1.5 Raggio di trasmissione *23*
2.5.1.6 La dimensione della memoria *24*
2.5.1.7 Resistenza alla manomissione *24*
2.5.1.8 Gli orologi di sistema *25*
2.5.1.9 Collocazione in ambienti ostili *25*
2.5.2 Vincoli imposti dalla rete *26*
2.5.2.1 Topologia dinamica *26*
2.5.2.2 Dimensione dei pacchetti e velocità di trasmissione dei dati *26*
2.5.2.3 Error rate dovuto al canale di trasmissione *26*
2.5.2.4 Connettività intermittente *27*
2.5.2.5 Latenza *27*
2.5.2.6 Unicast e Multicast *27*
2.5.2.7 Comunicazioni unidirezionali *27*
2.5.2.8 Sottoreti isolate *28*
2.5.2.9 Routing dinamico *28*
2.5.2.10 Destinatari sconosciuti *28*
2.5.2.11 Mittenti sconosciuti *28*
3. TIPI DI ATTACCHI E SOLUZIONI ESISTENTI **29**
3.1 Tassonomia degli avversari e degli attacchi 29
3.1.1 Classificazione degli avversari *29*
3.1.2 Classificazione degli attacchi *30*
3.1.2.1 Manomissione (tampering) o attacco fisico *31*
3.1.2.2 Impersonificazione *32*
3.1.2.3 Attacchi di tipo DoS (Denial of Service) *32*
3.2 Obiettivi del protocollo per la verifica dell'integrità 33
3.3 Alcuni protocolli per la verifica dell'integrità 34
3.3.1 Ipotesi comuni *34*

3.3.2 SWATT 35
3.3.3 Attestation 36
3.3.4 PIV 38
3.4 OSSERVAZIONI 39

4. SPECIFICA DEI REQUISITI E CARATTERISTICHE DELLA PIATTAFORMA SCELTA PER L'IMPLEMENTAZIONE 42

4.1 REQUISITI DEL PROTOCOLLO 43
4.1.1 Integrazione con le applicazioni e i servizi della rete 43
4.1.2 Autonomia 43
4.1.3 Soluzione software-only 43
4.1.4 Verifica remota 43
4.1.5 Server di autenticazione 44
4.1.6 Paradigma request-response 44
4.1.7 Configurazioni hardware note 44
4.1.8 Immagini della memoria dei sensori 44
4.1.9 Copertura totale della memoria 45
4.1.10 Scansione pseudo-casuale della memoria 45
4.1.11 Procedura di verifica crittograficamente sicura 46
4.1.12 Procedura di verifica nota 46
4.1.13 Non parallelizzabilità 46
4.1.14 Non comprimibilità 47
4.1.15 Resistenza ad attacchi di tipo replay e attacchi con response precalcolata 47
4.1.16 Resistenza ad attacchi in cui l'informazione fresca viene prevista o indovinata dall'avversario 47
4.1.17 Resistenza ad attacchi di manomissione di livello 1 e di livello 2 48
4.1.18 Deadline temporale 48
4.1.19 Meta-informazioni 48
4.1.20 Attivazione della procedura di verifica 49
4.1.21 Consumo energetico 50
4.2 CARATTERISTICHE DEI SENSORI TMOTE SKY 51
4.2.1 Descrizione 51
4.2.2 Caratteristiche hardware 51
4.2.3 Alimentazione 53
4.2.4 Microprocessore 53
4.2.5 Programmazione 54
4.2.6 Flash esterna 55
4.3 ORGANIZZAZIONE DELLO SPAZIO DI MEMORIA DI UN SENSORE TMOTE SKY 56

5. IL PROTOCOLLO PIV, L'ATTACCO DI PARALLELIZZAZIONE E LA SOLUZIONE PROPOSTA 59

5.1 IL PROTOCOLLO PIV 59
5.1.1 La randomized hash function (RHF) 60
5.1.2 Il protocollo 62
5.1.3 Analisi del livello di sicurezza di PIV 64
5.2 L'ATTACCO DI PARALLELIZZAZIONE 67
5.2.1 La vulnerabilità della RHF 67
5.2.2 I dettagli dell'attacco 68
5.2.3 Esempio di attacco 70
5.2.4 Conseguenze dell'attacco 71
5.3 SOLUZIONE ALL'ATTACCO 72
5.3.1 La tecnica usata dagli altri protocolli 72
5.3.2 La creazione dei blocchi x_i 73
5.3.3 Alcune tecniche per comporre i blocchi x_i 75
5.3.4 L'operazione di interleaving 79
5.3.5 Misure anti-parallelizzazione 81
5.3.6 Generatore di numeri pseudo-casuali 81
5.3.7 Ingombro in memoria dell'algoritmo PIVC e dell'algoritmo PIVm 82
5.3.8 Tempi di esecuzione dell'algoritmo PIVC e dell'algoritmo PIVm 84
5.4 CONCLUSIONI 85
5.5 ULTERIORI VANTAGGI 86

5.5.1 Risparmio nelle comunicazioni ... *86*
5.5.2 Server di autenticazione AS ... *89*
5.5.3 Dati non comprimibili ... *89*
5.6 SVANTAGGI ... 90

6. IMPLEMENTAZIONE DEL CLIENT PIVCM ... 91

6.1 TINYOS ... 91
6.1.1 Caratteristiche ... *91*
6.1.2 Comunicazione in TinyOS ... *93*
6.2 PROGRAMMAZIONE IN NESC ... 95
6.2.1 Creazione dei componenti ... *95*
6.2.2 Concorrenza in nesC ... *96*
6.3 APPLICAZIONE ... 97
6.3.1 Il modulo GaloisC ... *98*
6.3.2 Il modulo RandomM ... *100*
6.3.3 Il modulo PivcM ... *101*
6.3.4 I task del modulo PivcM ... *105*
6.3.5 Le funzioni di supporto del modulo PivcM ... *106*
6.4 FORMATO DEI PACCHETTI ... 109

7. IMPLEMENTAZIONE DEL SERVER PIVSM ... 111

7.1 APPLICAZIONE ... 111
7.1.1 Tipi di dati ... *112*
7.1.2 Diagramma a stati di PIVSm ... *112*
7.1.3 Le immagini della memoria dei sensori ... *114*
7.1.4 Il package pivs e la classe pivserver ... *114*
7.1.5 Il generatore di numeri pseudo-casuali ... *114*
7.1.6 la funzione calcola() ... *115*
7.1.7 La funzione scegliMPartizioni() ... *116*
7.1.8 La funzione riempi_x(int) ... *116*
7.1.9 La funzione memoria(int, int) ... *116*
7.2 FORMATO DEI PACCHETTI ... 117

8. TEST DEL SISTEMA ... 118

8.1 LA LIBRERIA DELLE OPERAZIONI ARITMETICHE IN GF(2^8) ... 119
8.2 IL GENERATORE DI NUMERI PSEUDO-CASUALI ... 119
8.3 LA RHF, OVVERO I TASK CALCOLA(), CALCOLABLOCCO()(LATO CLIENT) E LA FUNZIONE CALCOLA() (LATO SERVER) ... 119
8.4 LE FUNZIONI MEMORIA(INT,INT) E RIEMPI_X(INT) ... 120
8.5 LA FUNZIONE SCEGLIMPARTIZIONI ... 120

9. CONCLUSIONI ... 121

9.1 COMPORTAMENTO DEL PROTOCOLLO ... 121
9.2 TEMPI DI ESECUZIONE ... 121
9.3 OVERHEAD DI COMUNICAZIONE ... 122
9.4 SICUREZZA ED EFFICIENZA ... 122

BIBLIOGRAFIA ... 123

1. Introduzione

1.1 Descrizione del problema

Le reti di sensori sono strumenti che nel prossimo futuro potrebbero avere una grande diffusione, e anche qualora non si trattasse di sensori ma di generici sistemi *embedded*, la tendenza attuale è quella di permettergli di comunicare e collaborare.

In alcuni ambiti applicativi, le reti di nodi sensori vengono installate in spazi vasti, non sorvegliati. Collocare un gran numero di sensori in un tale ambiente li espone al rischio di essere sottratti da un malintenzionato, riprogrammati e poi re-inseriti nell'ambiente. n sensore che è stato manomesso in questo modo può rientrare a far parte della rete di sensori e successivamente tentare di sabotare in qualche modo il suo funzionamento.

Un protocollo per la verifica dell'integrità permetterebbe di accorgersi in maniera preventiva delle modifiche che possono aver subito i sensori, isolandoli dal resto della rete ed evitando così eventuali danni che potrebbero causare. Inoltre, grazie a questo stesso protocollo, il server di controllo della rete potrebbe avvertire le persone responsabili affinché prendano i dovuti provvedimenti per ripristinare i nodi manomessi oppure per scoprire il responsabile dell'attacco. La prospettiva di poter disporre di un protocollo per la verifica dell'integrità diventerebbe molto allettante anche in altri ambiti applicativi, ad esempio nelle reti *wireless* di altri sistemi *embedded*. Queste reti, infatti, potrebbero essere soggette ad attacchi dovuti non a persone malintenzionate che vogliano ottenere qualcosa di concreto (introdursi in un area sorvegliata oppure causare un danno materiale ad una struttura/edificio/ambiente evitando che la rete di sensori che la sorveglia lanci un allarme), bensì dovuti a dei software maliziosi come *troiani*, *worm* o *virus*, il cui unico scopo sia quello di diffondersi e/o di causare dei piccoli, fastidiosi malfunzionamenti.

1.2 La verifica dell'integrità delle applicazioni

In letteratura sono state già fatte alcune proposte di protocolli per la verifica dell'integrità nelle reti di sensori.

Tutti questi protocolli prevedono innanzitutto l'esistenza di un server di autenticazione, che deve essere un host con potenza di calcolo e quantità di memoria superiori a quelle dei sensori. Essi prevedono l'invio da parte del server di autenticazione di una richiesta (*challenge*). Al ricevimento della *challenge*, nel sensore si attiva una routine apposita, che raccoglie determinate informazioni riguardo al contenuto della memoria del sensore ed, in alcuni casi, anche riguardo allo stato interno della CPU. Immediatamente dopo aver preparato queste informazioni, il sensore le invia al server (*response*). A quel punto il server, dopo aver ricevuto la *response* dovrebbe essere in grado di determinare se il sensore è stato manomesso oppure no, confrontando la *response* ricevuta con delle altre informazioni in suo possesso.

Allo stato attuale, nessun microprocessore di quelli utilizzati nei nodi sensori è dotato di meccanismi hardware per verificare l'autenticità del codice. Di conseguenza, i protocolli di autenticazione che sono stati studiati e proposti sinora sono di tipo *software-only*. Essi prevedono che sul nodo venga fatto girare un codice (una routine, o funzione) che, analizzando il contenuto della memoria, aiuti determinare se questa è stata modificata.

Inoltre, i protocolli finora proposti si basano sul presupposto che l'avversario (colui che manomette il sensore) si limiti a riprogrammare il sensore o a modificare solo una parte del programma contenuto nel sensore senza però effettuare delle modifiche all'hardware sul sensore stesso. Quindi, questi schemi di autenticazione non garantiscono la propria efficacia contro attacchi che prevedano l'aggiunta di memoria oppure l'installazione di un microcontrollore con potenza di calcolo superiore all'originale. Questo perché sarebbe impossibile per un protocollo *software-only* capire se l'hardware è stato modificato in tali modi. Comunque, quest'ultimo tipo di attacco potrebbe risultare molto costoso, in termini di tempo e di denaro, per l'attaccante. Soprattutto nel caso in cui dovesse manomettere molte decine di sensori in una rete che ne contiene centinaia o migliaia.

Studiando alcuni dei proposte già esistenti è venuto alla luce che molti di essi non sono stati progettati per essere applicate su sistemi con potenza di calcolo molto ridotta (come nel caso dei nodi sensori), altri sono stati progettati per la verifica di

nodi isolati (come ad esempio i personal computer) mentre gli unici protocolli che sono stati studiati appositamente per le reti di sensori sono poco efficienti oppure sono vulnerabili in altri modi. In particolare, l'unico tra questi protocolli che presenta un algoritmo veramente efficiente dal punto di vista della complessità computazionale, il protocollo PIV, è vulnerabile perchè questo algoritmo può essere parallelizzato. Ovvero, se un avversario riuscisse a manomettere due sensori, potrebbe farli lavorare insieme riuscendo così ad ingannare il server di verifica dell'integrità.

Il contributo di questa tesi è stato di analizzare le varie soluzioni esistenti e quindi di proporre un protocollo di autenticazione che abbia entrambi i requisiti più importanti: un buon livello di sicurezza e soprattutto efficienza. Il protocollo proposto è una variante del protocollo PIV [3], la cui funzione di hash è stata modificata in modo da impedirne la parallelizzazione. Inoltre, sono state effettuate delle ulteriori modifiche al protocollo che ne hanno migliorato l'efficienza, minimizzando la quantità di dati trasmessi durante lo scambio di messaggi tra il server e i sensori.

2. La sicurezza nelle reti di sensori

2.1 La missione delle reti di sensori

Lo scopo primario di una rete di sensori è rilevare gli eventi che accadono nel campo recettivo dei sensori che compongono la rete ed eventualmente fare rapporto ad un centro di controllo e comando. I nodi sensori generalmente dispongono di misurazioni grezze (e.g. luminosità, temperatura, umidità), ma attraverso la cooperazione di più sensori è possibile ottenere informazioni molto più precise e affidabili, quali ad esempio gradienti di temperatura, informazioni sul movimento di veicoli o persone, informazioni sullo stato di un paziente (nel caso di utilizzo dei sensori in ambito medico) o in generale informazioni sullo stato globale di un sistema (o ambiente). Quando un sensore rileva un evento di interesse, esso può riferirlo direttamente ad un centro di controllo oppure può confrontarsi, collaborando, con i sensori vicini in modo da ottenere informazioni più precise riguardo all'evento e successivamente inviarle al centro di controllo. Il centro di controllo avrà poi il compito di prendere una decisione riguardo all'evento appena accaduto. Un centro di controllo può essere un elaboratore con potenza di calcolo e capacità di memoria superiori ai sensori, oppure un addetto alla sorveglianza di un'area o anche un medico responsabile di un paziente.

2.1.1 Protezione del perimetro di un'area

Uno scenario di applicazione tipico per una rete di sensori è la difesa dell'integrità di un'area, oppure del suo perimetro.

Per proteggere un perimetro, tipicamente i sensori vengono dislocati in maniera unidimensionale (e.g. lungo un recinto o una staccionata) per individuare eventuali violazioni. In questo tipo di applicazione la rete è statica, e durante il suo ciclo di vita ci sono pochi inserimenti di nuovi nodi o rimozioni. E' importante osservare che, anche quando i sensori sono relativamente vicini al centro di controllo, possono rimanere incustoditi per lunghi periodi di tempo.

2.1.2 Sorveglianza remota di un’area

Per sorvegliare un’area i sensori vengono disposti su due dimensioni, più o meno uniformemente. La loro collocazione può essere casuale (ad esempio se vengono lanciati da un elicottero o da un aereo) oppure calibrata. Una rete di questo tipo può essere composta da centinaia, migliaia o anche centinaia di migliaia di sensori. La topologia della rete varia in continuazione, perché i sensori periodicamente si pongono in uno stato a basso consumo energetico (sleep). Inoltre, dei nodi scompaiono dalla rete quando le loro batterie si esauriscono oppure perché sono stati sottratti o distrutti. Le misurazioni effettuate dai sensori possono essere inviate ad un centro di controllo da dove è possibile monitorare i movimenti di persone o veicoli nell’area monitorata. Quando rilevano un evento, i sensori possono agire immediatamente avvisando il centro di controllo oppure possono ritardare l’invio per non svelare la propria presenza oppure per motivi di risparmio energetico. Inoltre, il centro di controllo può inviare dei comandi ai sensori, ad esempio per cambiare il loro ciclo sonno-veglia. Anche in questo caso i sensori rimangono incustoditi per lunghi periodi di tempo, forse anche oltre la durata delle loro batterie.

Un altro tipo di scenario, con topologia statica, potrebbe essere un’operazione militare su territorio urbano: i soldati, avanzando installano dei sensori in molte zone di un edificio. Quelle zone a quel punto divengono ‘sicure’ poiché sotto controllo da parte della rete di sensori. Anche in questo caso la rete sarebbe composta di nodi statici, e la topologia si estenderebbe man mano che nuovi sensori vengono piazzati e accesi. Questo tipo di rete avrebbe una vita più breve, perché dopo aver occupato anche l’area urbana intorno all’edificio, i sensori potrebbero essere spenti e rimossi.

2.1.3 Monitoraggio ambientale

Quando utilizzati per il monitoraggio ambientale, i sensori sono sottoposti a notevole stress fisico. Sbalzi di temperatura e di umidità possono influire sulla durata delle batterie, mentre la vegetazione e la caduta di foglie (se i sensori sono collocati sul terreno) rappresentano un ostacolo per le comunicazioni radio. Inoltre, se lasciati incustoditi, i sensori possono essere accidentalmente distrutti o rubati.

La densità con cui i sensori devono essere collocati varierà a seconda del tipo di terreno e a seconda degli ostacoli che si trovano tra sensori vicini. Anche in questo caso i sensori sono tipicamente statici. Ciononostante, la topologia della rete varia in continuazione, perché i sensori periodicamente si pongono in uno stato a basso

consumo energetico (sleep). Inoltre, dei nodi scompaiono dalla rete quando le loro batterie si esauriscono oppure perché sono stati sottratti o distrutti, e si aggiungono alla rete quando le loro batterie vengono sostituite oppure quando nuovi nodi vengono collocati in sostituzione di nodi rubati o distrutti.

2.2 I vincoli energetici

2.2.1 Trasmissione e ricezione di dati

In una rete di sensori, lo scambio di informazioni tra sensori è condizionato dalle loro disponibilità energetiche e dai vincoli che vengono imposti per limitare il consumo energetico. Come osservato in [8], la quantità di energia necessaria per computare un bit è sensibilmente inferiore a quella necessaria per trasmetterlo. Questo è tanto più vero quanto più è grande la rete, poiché in una rete di dimensioni notevoli, se il nodo mittente non riesce a comunicare direttamente con il centro di controllo, un pacchetto di dati deve essere ritrasmesso più volte per arrivare al suddetto centro (rete multi-hop). Questo è uno dei motivi per cui è necessario progettare le applicazioni in modo da minimizzare la quantità di dati trasmessi.

Per risparmiare energia, prima di inviare un rapporto al centro di controllo, è necessario che il nodo cerchi di raccogliere quante più informazioni riguardo all'evento rilevato, anche confrontandosi con i nodi vicini. Inoltre, quando il rapporto viene inviato, in nodi intermedi che si trovano a dover inoltrare il messaggio dovrebbero poter fondere le informazioni in loro possesso a quelle contenute nel rapporto. Questa operazione viene detta di *accumulazione* o *aggregazione* dei dati. Grazie ad essa è possibile minimizzare la quantità di bit che vengono complessivamente instradati sulla rete. Alcuni nodi, eventualmente collocati in posizioni strategicamente (o logisticamente) più favorevoli, ed eventualmente dotati anche di maggiori riserve energetiche, potenza di calcolo e capacità di memoria, possono effettuare l'*aggregazione* di dati per conto di altri sensori.

I nodi di una rete possono essere suddivisi in gruppi, in base alla posizione in cui sono collocati, in base alla loro funzione all'interno della rete o anche in base ad altri criteri.

Il centro di controllo può inviare dei comandi ai sensori che compongono la rete, ad esempio per richiedere dei dati specifici oppure per indurre i sensori ad entrare in uno stato di risparmio energetico (sleep).

2.3 Ciclo di vita dei sensori

Il ciclo di vita di un sensore inizia con la sua produzione, prosegue con la sua collocazione in una rete di sensori e termina con il completamento della sua missione oppure con la sua distruzione o furto.

Secondo [9] possiamo suddividere la vita di un sensore in 6 fasi: la produzione, un periodo di stoccaggio in un deposito, l'inizializzazione prima della collocazione, la collocazione in una rete di sensori, lo svolgimento della missione e il completamento della missione.

2.3.1 Produzione

Alcuni aspetti della produzione dei sensori sono determinanti per il livello di sicurezza che possono offrire i sensori stessi. Questi aspetti sono:

- I sensori vengono prodotti in grandi quantità e a basso costo;
- Gli impianti di produzione potrebbero non avere forti restrizioni per limitare l'accesso ai non addetti;
- I sensori (o parti di essi) potrebbero essere sottratti o rubati durante il processo di produzione;
- Il processo produttivo potrebbe non essere vulnerabile a sabotaggi di tipo hardware o software. Se questo accadesse, verrebbero prodotti dei sensori non corrispondenti alle specifiche;
- Errori o guasti nei macchinari di produzione potrebbero causare dei bug o in generale dei malfunzionamenti nel comportamento dei sensori prodotti. Tali bug o malfunzionamenti potrebbero poi essere sfruttati da un avversario con intenzioni maliziose.

Durante il processo di produzione è possibile pre-caricare alcune funzioni basilari sul sensore. Inoltre, se vi sono degli algoritmi crittografici implementati via hardware, si ritiene che essi vengano installati durante questa fase. Inoltre, supporre che un impianto di produzione di sensori a basso costo non sia dotato di sistemi di sicurezza molto avanzati è ragionevole, perché questo inciderebbe pesantemente sul costo di produzione dei sensori stessi.

2.3.2 Stoccaggio

Dopo la produzione, i sensori rimarranno per un periodo di tempo variabile in un deposito o magazzino. Similmente a quanto osservato per gli impianti di produzione anche in questo caso è ragionevole supporre che nei depositi non ci sarà un consistente controllo degli accessi, il che espone i sensori al rischio di essere manomessi o rubati.

2.3.3. Inizializzazione e configurazione

Prima di procedere alla collocazione dei nodi è necessario prepararli, installando le applicazioni necessarie e configurando i parametri di funzionamento. Durante la fase di inizializzazione e configurazione è necessario fissare le seguenti caratteristiche:

- Installazione del software;
- Tipo di letture da effettuare (e.g. acustiche, sismiche, all'infrarosso);
- Ruolo del sensore nella rete (nodo semplice, nodo di aggregazione oppure gateway);
- Configurazione di parametri (e.g. ID di rete, chiavi crittografiche iniziali).

E' possibile limitare questa fase alla sola installazione di un *tool* di configurazione dinamica (ad es. Deluge, nel caso del sistema operativo TinyOS), che permetterà di installare e aggiornare le applicazioni successivamente, sfruttando l'interfaccia *wireless*.

2.3.4 Collocazione e funzionamento

Una volta collocati e accesi, i sensori inizieranno a svolgere le loro regolari funzioni. Un requisito per garantire un buon funzionamento della rete di sensori è l'auto-organizzazione. I sensori devono essere in grado di identificare i loro vicini e di creare dei percorsi di *routing* in maniera autonoma, senza bisogno di interventi dall'esterno. Allo stesso modo, i sensori devono essere in grado di accorgersi delle modifiche all'interno della rete: quando gli altri sensori vanno in modalità di risparmio energetico (*sleep*) oppure quando scompaiono per altri motivi e chiaramente anche il vice-versa, cioè quando ritornano dalla fase di *sleep* oppure quando vengono aggiunti nuovi nodi. Inoltre è importante che il *routing*, quando

esistono più percorsi per instradare i pacchetti, venga fatto in maniera bilanciata in modo da uniformare il consumo energetico nella rete.
Durante questa fase della loro vita i sensori possono rimanere per molto tempo incustoditi, anche fino all'esaurimento delle loro riserve energetiche. E' in questa fase che il rischio di furto o manomissione diventa più elevato.

2.3.5 Completamento della missione

Una rete di sensori può terminare la sua missione dopo la cessazione del motivo per cui era stata installata oppure a causa della *morte* dei sensori. La *morte* dei sensori avviene quando vengono fisicamente distrutti, quando si guastano, quando rimangono isolati dal resto della rete, oppure quando esauriscono le batterie. In ogni caso, anche i sensori morti che venissero catturati da un avversario potrebbero rappresentare un problema di sicurezza. Non soltanto per il rischio che l'avversario recuperi delle chiavi crittografiche da essi, ma anche perché egli potrebbe effettuare un'analisi statica delle applicazioni e scoprire bug o malfunzionamenti da usare a suo vantaggio per soverchiare altre reti di sensori.

2.4 Requisiti di sicurezza

Vi sono alcuni requisiti che una rete di sensori deve soddisfare, per essere considerata *sicura*:

2.4.1 Confidenzialità

Le informazioni raccolte dai sensori, o le informazioni critiche per il funzionamento delle applicazioni non devono essere accessibili a soggetti non autorizzati.

2.4.2 Autenticità/integrità

Il destinatario di un messaggio deve poter verificare l'identità del mittente.

2.4.3 Freschezza

Il destinatario di un messaggio deve essere in grado di distinguere un messaggio nuovo (fresco) da un messaggio vecchio che ha transitato a lungo nella rete prima di arrivare a destinazione oppure che è stato replicato da qualcuno.

2.4.4 Disponibilità

I servizi offerti dalla rete nel suo complesso o da alcuni nodi della rete che abbiano funzioni particolari, dovrebbero essere disponibili durante tutto l'arco di vita della rete.

Durante la progettazione di una rete, bisognerebbe osservare queste linee guida:

- La rete di sensori dovrebbe proteggere le proprie risorse (i sensori) da un eccessivo uso di potenza di calcolo e di trasmissione dei dati, per minimizzare il consumo energetico.
- I meccanismi di sicurezza presenti nella rete non dovrebbero interferire con la disponibilità dei dati raccolti né impedire il normale funzionamento della rete.
- Nessun servizio né meccanismo di sicurezza dovrebbe presentare un *single point of failure* (o *punto di fallimento globale*), ovvero qualsiasi componente strategicamente importante per la rete dovrebbe essere presente in più copie, per evitare che il fallimento di un singolo componente comprometta il funzionamento dell'intera rete.

Il requisito per cui i meccanismi di sicurezza non dovrebbero interferire con la disponibilità dei dati raccolti né impedire il normale funzionamento della rete è importante, ma è ovvio che in alcuni casi bisogna raggiungere un compromesso tra sicurezza ed efficienza, soprattutto nel caso di scenari *mission-critical* in ambienti ostili (ad es. scenari di applicazione militare). Sta ai progettisti di una rete stabilire se è più importante mantenere la *disponibilità* anche a scapito di temporanee riduzioni del livello di sicurezza oppure se è più importante mantenere il livello di sicurezza, anche a scapito di temporanee riduzioni di *disponibilità* dei servizi.

2.4.5 Accessibilità

Per permettere ai nodi intermedi, cioè a quei nodi che si trovano tra il mittente di un messaggio ed il suo destinatario, di effettuare aggregazione dei dati, è sconsigliabile

implementare meccanismi di confidenzialità *end-to-end*. Infatti, per limitare la quantità di dati che transitano dal ricevente verso il mittente di un messaggio, è necessario che i nodi intermedi siano in grado di analizzare il contenuto dei messaggi che inoltrano, anche per riuscire a rilevare incongruenze dovute ad es. ad errori di trasmissione e interrompere quindi il transito del pacchetto corrotto.

2.4.6 Auto-organizzazione

I sensori che compongono una rete devono essere in grado di rilevarne la topologia, sia al momento della collocazione dei sensori che durante la vita della rete, quando la topologia cambia in maniera dinamica. Allo stesso modo, anche i meccanismi di sicurezza presenti nella rete devono essere in grado di accorgersi dei cambiamenti nella sua topologia ed adattarsi di conseguenza.

3.7 Flessibilità:
Le reti di sensori possono essere usate in una grande varietà di situazioni. In alcune di queste, in particolare, degli aspetti quali condizioni ambientali, pericoli, e scopo della missione possono variare molto rapidamente. Cambiare lo scopo della rete può comportare l'aggiunta o la rimozione di sensori. Inoltre, potrebbe essere necessario fondere due reti in una, oppure partizionare una rete in due reti distinte. I meccanismi di sicurezza utilizzati devono essere abbastanza flessibili da adattarsi a queste modifiche. Per questo motivo è preferibile utilizzare i protocolli di sicurezza molto flessibili, che possano adattarsi a modifiche del numero di sensori, delle loro collocazioni e delle loro funzioni all'interno della rete.

2.5 Vincoli

Dopo aver definito i requisiti per la sicurezza di una rete di sensori, è necessario specificare alcuni vincoli imposti dai nodi sensori e dalla rete di sensori che sicuramente incideranno sulla progettazione di un protocollo di sicurezza.

2.5.1 Vincoli imposti dai nodi

Prima di iniziare a progettare un protocollo di sicurezza per reti di sensori è necessario analizzare le possibilità e le limitazioni offerte dall'hardware. Solitamente

si presuppone che i sensori siano economici, di tipo generico e con poca potenza di calcolo e capacità di memoria. Per svolgere operazioni più complesse (ad es. aggregazione di dati, o routing), in una rete possono essere presenti anche un minor numero di nodi più potenti, con maggior potenza di calcolo e quantità di memoria.

2.5.1.1 Riserve energetiche/Durata della batteria

L'energia a disposizione di un sensore è l'aspetto che pone il maggior limite all'uso che se ne può fare. Si presuppone che una volta collocati i sensori non possano essere ricaricati. Di conseguenza, la carica della batteria deve essere conservata il più a lungo possibile, per prolungare sia la vita del sensore che quella della rete. Nell'implementare un protocollo di sicurezza per reti di sensori questo aspetto deve essere valutato attentamente.

La presenza di protocolli di sicurezza comporterà un consumo di energia ulteriore, in una rete. Per valutare questo consumo di energia bisogna considerare il tempo impiegato per svolgere operazioni relative alla sicurezza (crittografare, de-crittografare, firmare dati, verificare firme, valutare l'integrità delle applicazioni), l'energia necessaria a trasmettere i dati relativi alla sicurezza (incluso l'*overhead* sui dati dovuto ai protocolli di sicurezza), e l'energia necessaria per memorizzare i dati necessari a implementare questi protocolli (ad es. chiavi crittografiche).

2.5.1.2 Consumo energetico indotto dagli algoritmi di sicurezza

Il consumo energetico indotto dall'esecuzione degli algoritmi di sicurezza dipende primariamente dal modello di CPU, dalla sua frequenza di funzionamento, e dal numero di cicli di clock necessari per svolgere queste operazioni. Il numero di cicli di clock necessari dipende dalla scelta degli algoritmi nonché dall'efficienza della loro implementazione. Un'idea per ridurre questo costo potrebbe essere ridurre la frequenza di clock e la tensione di alimentazione della CPU, ma i processori utilizzati attualmente nei nodi sensori non possiedono queste capacità. Inoltre essendo questi algoritmi molto costosi dal punto di vista della complessità computazionale, non sarebbe opportuno ridurre ulteriormente la frequenza di clock, poiché i tempi di esecuzione su queste CPU sono già notevolmente lunghi, tanto da rappresentare un problema per la disponibilità di altri servizi. Infatti, mentre un sensore è impegnato

nell’esecuzione di un algoritmo di sicurezza, esso generalmente non può rispondere ad altre richieste.

2.5.1.3 Consumo energetico dovuto alla trasmissione di dati relativi alla sicurezza

Oltre al consumo dovuto alla computazione, gli algoritmi di sicurezza causano un ulteriore consumo dovuto alla trasmissione dei loro dati. Tra queste comunicazioni troviamo lo scambio di chiavi crittografiche e di certificati, l’invio di checksum o hash, firme digitali, vettori di inizializzazione di alcuni algoritmi nonché il padding sui dati causato da alcuni algoritmi di crittografia.

Per ridurre questi costi è preferibile scegliere algoritmi più semplici, che scambino chiavi soltanto con i nodi vicini, che permettano l’aggregazione o l’accumulazione dei dati provenienti da più nodi e che, in generale, limitino le comunicazioni tra nodi.

2.5.1.4 Cicli sonno-veglia

Per prolungare la durata della batteria, i nodi devono trascorrere molto tempo in una modalità di risparmio energetico. Quando si trovano in questa modalità i sensori non sono in grado di ricevere comunicazioni né esiste modo di risvegliarli, quindi sono di fatto isolati dal resto della rete. Un protocollo di sicurezza per reti di sensori deve tener conto di questo aspetto per evitare che, se dei nodi si trovano in stato di *sleep* durante un aggiornamento, questi rimangano poi completamente isolati dalla rete anche quando si risveglieranno, non essendo stati aggiornati con i nuovi parametri.

2.5.1.5 Raggio di trasmissione

Il raggio di trasmissione dei nodi sensori è limitato, per ridurne il consumo energetico. Inoltre vi altri due pregi che si ottengono mantenendo un raggio di trasmissione limitato: si riduce la probabilità che i sensori vengano scoperti (nel caso di reti di sensori utilizzate a scopo militare o bellico) e si riduce la possibilità di collisioni tra pacchetti quando nodi diversi cercano di trasmettere contemporaneamente. I vari modelli di sensori hanno potenze di trasmissione diverse e portate diverse. Tipicamente la potenza di trasmissione è dell’ordine di qualche

decina di mW e i segnali trasmessi su terreno aperto sono intelleggibili fino a distenze di 100m.

2.5.1.6 La dimensione della memoria

I sensori sono dotati di due tipi di memoria. Una memoria Flash per immagazzinare il sistema operativo e le applicazioni, ed una memoria RAM per immagazzinare dati, variabili e per contenere lo stack (uno o più d'uno, nel caso di sistemi multitasking) di programma. Inoltre, alcuni modelli sono dotati di un'ulteriore memoria Flash, di capacità maggiore, che serve per effettuare backup dei dati, per immagazzinare altre applicazioni (nel caso in si voglia poter riprogrammare dinamicamente un sensore emanando un apposito comando dall'esterno) oppure per implementare un piccolo *file system*.

Queste memorie sono generalmente molto piccole, specialmente la memoria RAM. La RAM, infatti, è ciò che influisce maggiormente sul consumo energetico "a riposo". Per conservare i dati presenti in RAM è necessario mantenere la sua alimentazione anche quando in nodo va in regime di basso consumo energetico. Ed è per questo motivo i produttori cercano di limitare al massimo la quantità di memoria RAM presente sui sensori. Un parametro di interesse, per esporre meglio questa scelta progettuale, è il rapporto tra la dimensione della memoria Flash e la dimensione della RAM. Tra i sensori più utilizzati con TinyOS questo rapporto rimane tra 4:1 (Moteiv Tmote-Sky) e 16:1 (Crossbow Mica2 e MicaZ), e nel futuro questo rapporto probabilmente crescerà ancora.

Le memorie Flash installate sui nodi sensori sono dell'ordine di qualche decina di KBytes, mentre le RAM sono dell'ordine di qualche KBytes. Per questo motivo devono essere considerate una risorsa molto preziosa. Specialmente nel caso in cui vi siano dei meccanismi di sicurezza che devono coesistere insieme all'applicazione di rilevamento dei dati. La quantità di memoria disponibile per l'applicazione diventa quindi un fattore molto critico, e va riservata molta cura all'implementazione degli algoritmi, in modo che le routine siano quanto più brevi e concise.

2.5.1.7 Resistenza alla manomissione

I sensori vengono costruiti seguendo criteri di minimizzazione del costo, per cui essi sono solitamente sprovvisti di meccanismi di protezione hardware. I meccanismi anti manomissione (*tamper-protection*) di tipo hardware si dividono in due categorie:

attivi e passivi. I meccanismi attivi sono parti di circuiteria ideate appositamente per proteggere dati sensibili. I meccanismi passivi, invece, sono quelli che non richiedono energia per espletare la loro funzione; tra questi troviamo sigilli anti intrusione sugli involucri e gusci in resine epossiliche per ricoprire i chip.

I meccanismi di protezione attiva richiederebbero ulteriori costi di progettazione e di produzione, e quindi non vengono mai implementati dai produttori di sensori, mentre i meccanismi passivi sono ritenuti inaffidabili, come mostrato in [10] e quindi vengono tralasciati anch'essi.

In ogni caso, i meccanismi anti manomissione non possono resistere a tutti i tipi di attacchi. Un avversario abbastanza motivato potrebbe comunque risalire alle informazioni segrete contenute nei nodi catturati, proprio a causa delle debolezze dei meccanismi di protezione hardware. Per questo motivo alcuni autori (ad esempio [9]) consigliano di progettare le reti di sensori preventivando che uno o più nodi sensori potrebbero essere catturati e manomessi. E quindi la priorità diventa la protezione della rete e non la protezione dei singoli sensori.

2.5.1.8 Gli orologi di sistema

I nodi che compongono una rete di sensori hanno bisogno di sincronizzare i propri orologi. In alcune applicazioni questo aspetto è di importanza cruciale. I protocolli di sincronizzazione temporale tra i sensori devono poter resistere ad attacchi di modifica e replica dei pacchetti. Il metodo più sicuro per ottenere la sincronizzazione è usare il sistema GPS come sorgente alla quale tutti i nodi si agganciano, perché esso non è soggetto ad attacchi di '*spoofing*'; ma non sempre questo sistema è adottabile (ad es. in ambienti chiusi) o economicamente accettabile. Tuttavia, esistono altri protocolli affidabili che fanno uso di messaggi di sincronizzazione o di sensori che dispongono di orologi più accurati.

2.5.1.9 Collocazione in ambienti ostili

Ancora una volta è necessario far notare che spesso i sensori vengono collocati in ambienti dove non possono essere sorvegliati. E più lunghi sono i periodi nei quali i sensori rimangono incustoditi, più cresce la probabilità che essi vengano compromessi da un avversario.

2.5.2 Vincoli imposti dalla rete

Le reti di sensori sono reti poco convenzionali e devono affrontare delle problematiche molto diverse rispetto alle reti tradizionali.

2.5.2.1 Topologia dinamica

La topologia di una rete di sensori si determina al momento della collocazione dei nodi. Durante la vita della rete questa topologia può variare. Questo vincolo costringe chi progetta una rete di sensori a non stabilire dei ruoli fissi per i sensori all'interno della rete. I nodi dovrebbero essere in grado di cambiare ruolo, all'occorrenza, per mantenere l'affidabilità e la disponibilità dei servizi offerti dalla rete.

2.5.2.2 Dimensione dei pacchetti e velocità di trasmissione dei dati

La dimensione dei pacchetti e il *data rate* sono entrambi molto limitati, rispetto alle reti tradizionali. In TinyOS, ad esempio, la dimensione dei pacchetti è fissata ad un massimo di 29 Bytes per il *payload* più 7 Bytes per l'*header*, ed il data rate di trasmissione dichiarato è di 250 kbps (anche se nella pratica il *throughput* non supera i 15 kbps). Questi dati evidenziano innanzitutto il fatto che l'header rappresenta una grossa percentuale sulla dimensione totale del pacchetto (sempre superiore al 19,4%), ed è quindi nuovamente necessario ribadire che i protocolli di sicurezza devono cercare di minimizzare le comunicazioni tra nodi, al fine di minimizzare il consumo energetico e prolungare la vita dei sensori nonché della rete.

2.5.2.3 *Error rate* dovuto al canale di trasmissione

In questa sede daremo per assunto che i protocolli di trasmissione di basso livello siano in grado di riconoscere e scartare i pacchetti contenenti degli errori. Ciononostante, nell'eventualità che un errore si propagasse fino a un livello più alto, ad esempio a livelli dove vengono instaurate relazioni di confidenzialità o autenticità dei messaggi, le conseguenze sarebbero imprevedibili e potrebbero causare l'impossibilità di uno o più nodi di scambiare dati di livello applicativo anche nel futuro.

2.5.2.4 Connettività intermittente

In una rete di sensori la connettività tra nodi cambia in continuazione, sia a causa dei periodi di *sleep* (basso consumo energetico) dei nodi che a causa di problemi che riguardano il canale di trasmissione. I periodi di *sleep* dei nodi possono cambiare durante la vita della rete, in funzione degli eventi rilevati e delle riserve di energia. I problemi di trasmissione dovuti al canale, invece, dipenderanno dalle condizioni atmosferiche e da altri fattori che potrebbero interessare l'area in cui sono dislocati i sensori.

I protocolli di sicurezza dovrebbero essere in grado di adattarsi a condizioni di scarsa connettività, per evitare che alcuni nodi rimangano isolati dal resto della rete se, durante un aggiornamento, non sono riusciti ad ottenere le chiavi crittografiche necessarie a comunicare.

2.5.2.5 Latenza

La latenza nella consegna dei pacchetti può dipendere da congestioni del canale e dal sovraccarico della CPU, soprattutto nel caso di routing di tipo *multi-hop*. Soprattutto nel casi di alcuni protocolli di sicurezza, in cui i tempi di risposta sono un fattore determinante, è necessario apportare delle modifiche in modo da tollerare un margine di ritardo.

2.5.2.6 Unicast e Multicast

Molti tra i protocolli utilizzati nelle reti di sensori presuppongono il paradigma di comunicazione unicast. Ciononostante il tipo di rete favorisce il paradigma multicast, tanto che riuscire ad utilizzare quest'ultimo potrebbe comportare un significativo vantaggio.

2.5.2.7 Comunicazioni unidirezionali

In alcuni casi i nodi sono costretti ad utilizzare comunicazioni unidirezionali. Un esempio di un tale scenario potrebbe essere una rete di sensori posizionati su territorio nemico. In determinati momenti i sensori rischiano di essere individuati e quindi rimangono in silenzio radio finché il pericolo non cessa, continuando però a svolgere le loro funzioni di rilvamento dei dati. I protocolli di sicurezza dovrebbe

essere in grado di affrontare queste situazioni adattandosi di conseguenza, ad esempio ritardando gli aggiornamenti o le interrogazioni fino a quando non vengano ripristinate le normali condizioni di funzionamento.

2.5.2.8 Sottoreti isolate

In una rete di sensori, a causa dei periodi di *sleep*, soprattutto quando la densità dei sensori scende a causa della morte di alcuni (o molti) di essi, un gruppo (o più gruppi) di nodi potrebbe rimanere isolato dal resto della rete. Nel momento in cui la connettività venisse ripristinata la rete dovrebbe riuscire a riorganizzarsi attuando una fusione tra le sottoreti che si sono create.

2.5.2.9 Routing dinamico

Al diminuire delle riserve energetiche dei nodi, diventa necessario modificare le tabelle di routing per distribuire uniformemente il consumo energetico. Un protocollo di sicurezza non dovrebbe far troppo affidamento su dei percorsi di routing, perché questi possono cambiare in corso d'opera.

2.5.2.10 Destinatari sconosciuti

Quando un nodo trasmette un pacchetto, se il destinatario non è nella sua portata, questo pacchetto deve essere instradato attraverso altri nodi. Questi nodi intermedi sono sconosciuti e potrebbero essere maliziosi. Sarà quindi necessario effettuare una scelta: se implementare i protocolli di sicurezza su base end-to-end oppure su base hop-by-hop.

2.5.2.11 Mittenti sconosciuti

Quando un nodo riceve un pacchetto, non può stabilire con certezza la sua provenienza, quindi l'autenticazione del mittente dovrà essere effettuata a livello dei protocolli di sicurezza. Inoltre, può essere utile far notare che nell'implementazione di TinyOS non è previsto l'invio dell'indirizzo del mittente nell'header dei pacchetti (il protocollo di comunicazione IEEE 802.15.4, quello utilizzato dai sensori Moteiv Telos e Tmote-Sky, può includere l'indirizzo del mittente oppure no), per risparmiare alcuni Bytes nei pacchetti inviati.

3. Tipi di attacchi e soluzioni esistenti

3.1 Tassonomia degli avversari e degli attacchi

3.1.1 Classificazione degli avversari

Esistono molti modi di classificare gli attacchi ad un sistema informatico. Il lavoro [11] dà una valida descrizione di molti di essi. Ciononostante si rende necessario dare alcune delucidazioni su quali sono i pericoli da cui bisogna proteggere una rete di sensori.

Come è stato già detto, una rete di sensori è composta da centinaia o migliaia di sensori, ognuno dei quali è uno strumento di valore molto contenuto. E' per questo motivo che, quando una rete di sensori viene collocata, generalmente vengono installati più sensori di quanti ne sarebbero strettamente necessari per svolgere quel compito. Questa ridondanza, pur non comportando un aumento insostenibile dei costi, permette alla rete di avere una certa tolleranza ai guasti ed inoltre, se i compiti (soprattutto il routing) sono distribuiti su più sensori la rete avrà vita più lunga.

Date queste premesse, diventa plausibile che di fronte al pericolo di attacchi da avversario malizioso, la priorità sia di proteggere la rete e i suoi servizi piuttosto che i singoli sensori. Ugualmente, un avversario sarà più interessato a impadronirsi o disturbare il funzionamento dell'intera rete piuttosto che di un singolo sensore.

In [12] viene data una panoramica dei tipi di avversari che possono attaccare un sistema informatico. Questa suddivisione è stata molto usata in ambito di sicurezza perché suddivide gli avversari in base alle loro conoscenze e alle disponibilità di risorse. In ordine di pericolosità crescente, essi vengono suddivisi in:

- ***Clever outsiders***: letteralmente, forestieri intelligenti. Sono persone molto capaci, ma con poca conoscenza dei sistemi che vogliono attaccare. Inoltre, non hanno molte risorse economiche a disposizione. Questi avversari probabilmente cercheranno di trovare un punto debole del sistema per sfruttarlo a proprio vantaggio.
- ***Knowledgeable insiders***: sono sostanzialmente degli addetti ai lavori. Hanno ottime conoscenze tecniche riguardo al sistema e molta esperienza. Forse non conoscono benissimo alcuni aspetti del sistema, ma possono avere accesso a

quasi tutti i suoi componenti. Spesso dispongono anche di apparecchiature molto sofisticate.

- ***Funded organizations***: sono gruppi organizzati che dispongono di molto denaro, sostanzialmente in grado di assoldare delle vere e proprie *task-force*: esperti in molti campi, con grande esperienza e moltissime risorse a disposizione. Sono in grado di analizzare un sistema molto in profondità e sono in grado di progettare attacchi molto sofisticati. A volte possono servirsi di 'addetti ai lavori' per raccogliere più informazioni.

3.1.2 Classificazione degli attacchi

In [10] viene fatta un'analisi molto realistica sui sistemi usati per proteggere le informazioni nei sistemi informatici ed elettronici, e da questa emerge che un avversario molto determinato con libero accesso ad un sistema può recuperarne qualsiasi informazione 'segreta'. In [17] vengono analizzati degli attacchi, di cui alcuni molto semplici ed economici, che possono essere usati per recuperare informazioni 'segrete' da congegni quali smart-card, EEPROM, banchi di RAM o microprocessori, usati in ambiti dove è necessario proteggere qualche segreto o qualche servizio da un avversario che abbia accesso al congegno. In aggiunta, bisogna tener conto è che negli scenari in cui i sensori vengono lasciati incustoditi, qualsiasi avversario può avere fisicamente accesso ai sensori e quindi può catturarli e analizzarli con tranquillità.

La conclusione che se ne può trarre è che sarebbe quasi impossibile proteggere i singoli sensori, soprattutto se si considera che, per motivi di riduzione dei costi, sui sensori non è presente alcun meccanismo di protezione. In questa sede quindi daremo per assunto che i sensori che cadono in mani avversarie verranno sicuramente compromessi, e l'avversario riuscirà a recuperare qualsiasi chiave crittografica o altro segreto ivi contenuto, così come daremo per scontato che l'avversario possa modificare l'applicazione presente sul sensore, oppure riprogrammarlo interamente per poi utilizzarlo in un successivo attacco alla rete.

Gli attacchi che un avversario può portare ad una rete di sensori sono di due tipi: attacchi ai servizi o attacchi ai dati. Gli attacchi ai servizi sono focalizzati all'interruzione o al disturbo dei servizi quali ad es. il routing, la localizzazione dei nodi, la sincronizzazione temporale oppure i servizi di cifratura dei messaggi. Gli

attacchi ai dati sono focalizzati a interrompere la trasmissione dei dati, a sottrarre dati oppure a inserire dati falsi nella rete.

Noi distingueremo tre metodi che l'avversario può utilizzare per orchestrare un attacco alla rete: attacchi per manomissione, impersonificazione e attacchi di tipo DoS (Denial of Service).

3.1.2.1 Manomissione (*tampering*) o attacco fisico

Un attacco di manomissione avviene quando un avversario riesce a manomettere uno o più sensori della rete in modo da modificarne il comportamento. Il sensore così modificato deve sembrare genuino agli occhi del centro di controllo della rete, ma in realtà i dati che hanno origine in quel sensore o che transitano attraverso di esso vengono modificati oppure inoltrati anche ad un altro destinatario. Questo tipo di attacco è detto anche "attacco fisico", perché l'avversario ha bisogno di accedere fisicamente al sensore per poterlo manipolare.
Gli attacchi fisici sono generalmente suddivisi in tre livelli di difficoltà, in base allo sforzo che deve fare l'avversario per manomettere uno o più sensori:

- ***Attacco di livello 1***

L'avversario può semplicemente riprogrammare il sensore e rimetterlo nella sua collocazione originaria. Nel caso in cui l'avversario abbia bisogno di più sensori, egli può semplicemente installare la stessa applicazione anche su altri sensori che ha catturato.

- ***Attacco di livello 2***

Per riuscire a manomettere un sensore l'avversario deve analizzare il comportamento dell'applicazione presente su di esso e deve escogitare un modo per modificarla senza comprometterne il funzionamento. Nel caso in cui l'avversario abbia bisogno di più sensori, egli deve ripetere l'analisi per ogni sensore che intende manomettere. In sostanza, non è possibile sfruttare l'esperienza acquisita manomettendo un sensore per manometterne altri.

- ***Attacco di livello 3***

Per riuscire a manomettere un sensore l'avversario deve modificarne la configurazione hardware. Ad esempio potrebbe installare una RAM più capiente oppure una CPU più potente.

3.1.2.2 Impersonificazione

Un attacco di impersonificazione avviene quando un avversario prepara (programma) un sensore in modo che esso emuli il comportamento di un sensore genuino. Successivamente lo colloca nell'area della rete, e questo sensore cerca di entrare nella rete per poi immettere dei dati falsi nella rete, per rubare dati oppure per ingannare in qualche modo il centro di controllo o gli altri sensori.

3.1.2.3 Attacchi di tipo DoS (Denial of Service)

Un attacco di tipo DoS è un attacco di sabotaggio. Il suo scopo è quello di interrompere la disponibilità dei servizi offerti dalla rete. Un avversario potrebbe distruggere dei nodi in modo da lasciare isolate intere porzioni di rete oppure potrebbe interrompere le comunicazioni saturando il canale di trasmissione utilizzando dei sensori appositamente programmati o con delle apparecchiature di *jamming*.

3.2 *Obiettivi del protocollo per la verifica dell'integrità*

Gli attacchi per manomissione o impersonificazione producono sintomi che non sono facilmente riconoscibili. Se l'avversario ha acquisito conoscenze sul funzionamento della rete e dei sensori, può fare in modo che i suoi sensori si comportino in modo del tutto credibile agli occhi del centro di controllo. Di conseguenza, dei sensori malizioni potrebbero infliggere molti danni alla rete o ai dati prima di essere scoperti.

L'obiettivo che vorremmo raggiungere, per evitare che un avversario possa nuocere ad una rete di sensori, è di impedire ai sensori manomessi di accedere alla rete. Il protocollo che vedremo, infatti, cerca di costruire una rete composta soltanto da sensori 'affidabili'. Per stabilire la loro affidabilità, i sensori devono essere sottoposti ad un test in cui viene verificata l'integrità dell'applicazione installata su di essi. Se non riescono a superarlo gli viene impedito di comunicare con gli altri sensori. In pratica, visto che non è possibile impedire ad un avversario di catturare un sensore e manipolare il programma che questo contiene, è invece possibile impedire a questo sensore di rientrare nella rete dopo essere stato manipolato. Uno schema di questo tipo può riuscire a respingere attacchi per manomissione o impersonificazione.

In particolare si vorrebbe ottenere resistenza agli attacchi per manomissione di livello 1 e 2 e resistenza agli attacchi per impersonificazione. La resistenza agli attacchi per manomissione di livello 3 è impossibile da ottenere utilizzando soluzioni *software-only*. La spiegazione di questa affermazione risiede nel fatto che è impossibile, per una procedura software, avere informazioni complete riguardo all'architettura del sistema su cui essa viene eseguita.

Inoltre, gli attacchi di tipo DoS non verranno trattati in questa sede. Questi attacchi producono dei sintomi molto particolari e facilmente riconoscibili, e richiedono tecniche particolari per poter essere contrastati.

3.3 Alcuni protocolli per la verifica dell'integrità

Di seguito verranno descritti e analizzati alcuni protocolli per la verifica dell'integrità delle applicazioni.

3.3.1 Ipotesi comuni

I protocolli di verifica dell'integrità che verranno mostrati fanno uso di alcune ipotesi semplificative comuni:

- L'obiettivo che si vuole ottenere è riuscire a contrastare attacchi fisici di livello 2 e attacchi di impersonificazione. Questi protocolli utilizzano soluzioni *software-only*, e per questo motivo non avrebbero modo di accorgersi che un avversario ha modificato l'hardware di un sensore. In ogni caso, riuscire a contrastare attacchi di livello 1 e 2 significa rendere molto complesso il compito di un avversario. Infatti, se questi volesse effettuare un attacco su una rete di sensori sarebbe costretto a effettuare modifiche su molti sensori, il che comporterebbe costi molto elevati.
- Questo tipo di protocolli non può contrastare attacchi di tipo DoS (*Denial of Service*), poiché questi possono essere portati anche senza che il sensore manomesso rientri legalmente tra i sensori ritenuti affidabili.
- I protocolli che vedremo prevedono l'esistenza di un server di fiducia, che si suppone non possa essere manomesso e che interroga i sensori per verificarne l'integrità. Esistono anche altri schemi di verifica dell'integrità noti come algoritmi di *self-checking* o *self-hashing* (alcuni esempi di questi algoritmi vengono presentati in [13],[14],[15] e ne viene data una descrizione più generica in [16]) che non fanno uso di un server di verifica, ma questi sono stati studiati per proteggere il software nel caso venga fatto funzionare su elaboratori isolati (ad es. applicazioni per personal computer). Usare un server esterno di fiducia ci permette di prendere decisioni dall'esterno senza dover demandare al nodo stesso la decisione di escludersi dalla rete oppure no. Questo implica un aumento del livello di sicurezza, perché quando un sensore viene manomesso non è più possibile aver fiducia in esso per qualsiasi operazione gli venga richiesta.
- Il server possiede una copia del contenuto della memoria flash di ogni sensore, ovvero, quando l'applicazione viene installata, viene fatta un'immagine della

memoria programma e questa immagine viene conservata nel server di autenticazione.

- Una richiesta di autenticazione è una sorta di chiamata a procedura remota. Quando deve effettuare la verifica di un sensore, il server gli invia una richiesta di esecuzione. Il sensore a quel punto esegue di una funzione di *hash* sul contenuto della propria memoria e successivamente inoltra al server il risultato della computazione. Per evitare attacchi di tipo *replay* (il sensore manomesso contiene una risposta vecchia e la manda al server quando è interrogato), le funzioni di *hash* sono studiate in modo tale da includere nella computazione anche un'informazione fresca (fresh), diversa ad ogni richiesta, che il server invia al sensore unitamente alla richiesta. Il risultato della computazione dipende anche da questo valore (o insieme di valori), e quindi ci si assicura che il nodo debba effettivamente eseguire la funzione quando gli viene richiesto.
- L'autenticazione di un nodo può essere richiesta in tre casi:
  - periodicamente
  - se il nodo esibisce un comportamento ritenuto "strano"; nel caso sia attivo un servizio di IDS (Intrusion Detection System) progettato per rilevare anomalie nella rete
  - se rimane inattivo per un periodo di tempo ritenuto troppo lungo (abbastanza lungo da giustificare il sospetto che sia stato sottratto, modificato e riportato nella sua sede originaria).

3.3.2 SWATT

SWATT [1] prevede che la scansione della memoria sia effettuata in maniera casuale, accedendo ad indirizzi prodotti da un generatore di numeri pseudo-casuali che implementa l'algoritmo RC4. Il risultato finale della computazione dipende dall'ordine in cui le locazioni di memoria sono accedute, e la *challenge* contiene un seme per inizializzare il generatore RC4. In questo modo il protocollo diviene resistente agli attacchi di tipo *replay*.

Poiché il risultato della computazione dipende da tutto il contenuto della memoria, se un avversario ne modificasse anche soltanto una piccola parte, dovrebbe comunque mantenere una copia dei dati originali (backup), e dovrebbe modificare la procedura di calcolo della *hash* in modo che questa (quando se ne presenta la necessità) prelevi i dati originali dalla loro nuova posizione anziché dalla posizione

che occupavano originariamente. Al fine di trovare una zona di memoria in cui conservare i dati originali, l'avversario potrebbe prendere in considerazione delle aree di memoria il cui contenuto sia prevedibile (una zona di memoria contenente bytes impostati a zero oppure una zona di memoria non utilizzata o anche una zona di memoria il cui contenuto sia comprimibile), dove sarebbe possibile conservare quei dati senza ulteriore perdita di contenuto informativo della memoria.

Secondo gli autori, però, questa modifica della procedura di calcolo da parte dell'avversario comporterebbe un cospicuo incremento del tempo di esecuzione, tale che il nodo non riuscirebbe più a rispondere al server entro una deadline temporale assegnata. Ovviamente, questa deadline dovrà tener conto anche dei tempi di trasmissione dei pacchetti e degli eventuali ritardi introdotti dai nodi intermedi, cioè quei nodi che si trovano tra il server ed il sensore sotto esame.

Per impedire attacchi di parallelizzazione, ovvero quegli attacchi in cui l'avversario manomette due sensori in maniera tale da fargli calcolare la *hash* unendo la potenza di calcolo delle due CPU, il generatore pseudo-casuale di indirizzi è costruito in modo tale che gli indirizzi generati dipendano sia dallo stato interno del generatore sia dal valore istantaneo del checksum. Il valore del checksum ovviamente varia durante l'esecuzione della procedura ed in questo modo, anche se un avversario copiasse una vasta porzione di memoria su un altro sensore con l'intento di fargli calcolare solo le parti di checksum relative ai dati da esso posseduti, questo sarebbe impossibile perché il secondo sensore non sarebbe in grado di produrre la stessa sequenza di numeri.

Infine, gli autori mostrano un'implementazione del codice ottimizzata per CPU a 8 bit.

3.3.3 Attestation

La soluzione proposta da [2] prevede che la procedura che calcola la funzione di *hash* non risieda nel nodo, bensì che venga inviata al nodo unitamente alla *challenge*. Il protocollo prevede che la procedura di *hash* inviata sia sempre diversa. Questo si può ottenere mediante le seguenti tecniche:

1. *randomizzazione* del codice (il codice è sempre diverso, in maniera casuale)
2. cifratura (alcune parti del codice sono cifrate con metodi semplici in modo tale che vengano decifrate dal nodo stesso prima di essere eseguite)
3. inserimento di porzioni di codice automodificante

4. utilizzo di espressioni “opache” (*opaque predicates*) e pesante utilizzo di puntatori, anche a più livelli (es. “puntatori a puntatori” o “puntatori a puntatori a puntatori”)
5. inserimento di istruzioni fasulle e fuorvianti, per rendere più complessa un’eventuale operazione di *reverse-engineering* che faccia uso di un disassemblatore.

In ogni caso, la procedura in sé dovrebbe comunque calcolare una *hash* del contenuto della memoria, e gli autori propongono di utilizzare una soluzione simile a quella di SWATT [1]. Le locazioni di memoria vengono lette in sequenza pseudo-casuale, e la procedura continua a fare accessi in memoria finchè tutte le locazioni non sono state lette *almeno una volta*.

Attestation prevede un vincolo temporale che il nodo deve rispettare per fornire una risposta alla *challenge*. Questo vincolo temporale T_{wait} è pari a:

$$T_{wait} = (2*r) + e + \Delta,$$

dove r è il tempo in cui un pacchetto di dati attraversa la rete per arrivare dal server al sensore, e è il tempo di esecuzione previsto per la procedura e Δ rappresenta una stima della latenza introdotta dai sensori intermedi che devono instradare il pacchetto.

Gli autori spiegano che usare queste cinque tecniche di offuscamento del codice, unite al vincolo di dover fornire una risposta entro un tempo prefissato renderebbero inviolabile il protocollo, poiché per far entrare un sensore nella rete un avversario sarebbe costretto ad analizzare la procedura, comprenderla e ingannarla rimanendo comunque nei limiti della *deadline* temporale. E secondo gli autori nessun avversario sarebbe in grado di fare queste operazioni in tempi così brevi.

3.3.4 PIV

La soluzione [3] è apparentemente simile alla [1], ma prevede che la memoria venga scandita in maniera sequenziale, ottenendo così una procedura semplice la cui esecuzione da parte di un sensore sia veloce e poco dispendiosa dal punto di vista energetico. Infatti, questa procedura riesce a raggiungere la copertura totale della memoria (ogni locazione viene letta almeno una volta) con il minimo numero di accessi: ovvero, se la memoria è composta da n locazioni, la procedura di PIV farà esattamente n accessi. Inoltre, nel caso di PIV, non è necessario avere un generatore di numeri pseudo-casuali a bordo del sensore, perché la procedura non ne fa uso. Questo velocizza ulteriormente l'esecuzione della procedura. Il protocollo prevede che il server invii dei dati freschi (*fresh*) all'interno della *challenge*. Questi dati sono un vettore di interi e una matrice di interi. Inoltre la procedura di calcolo viene inviata unitamente alla *challenge*. La procedura di calcolo della *hash* è costruita in maniera tale da includere i dati freschi nella computazione, in modo che il risultato non possa essere riprodotto (o precalcolato) senza preventivamente conoscere le informazioni stesse. Questo impedisce ad un avversario di fare attacchi di tipo *replay* o attacchi con *hash* precalcolata.

La procedura per il calcolo della *hash* si appoggia ad una funzione detta RHF (*Randomized Hash Function*). Quest'ultima possiede due interessanti proprietà:

1. il risultato della funzione può essere calcolato in due modi: il primo (quello che viene fatto eseguire dai sensori), pur dovendo fare molte operazioni aritmetiche tra matrici, necessita di poca memoria RAM, rispettando quindi il vincoli imposti dall'architettura dei sensori. Il secondo modo (fatto eseguire dal server) necessita di più memoria ma risulta più semplice dal punto di vista computazionale. In questo modo il server, dato che dispone di molta memoria, può effettuare la verifica di più sensori contemporaneamente senza sovraccarico di CPU.
2. questa funzione sfrutta dei polinomi quadratici multivariati definiti su un campo finito di Rijndael-Galois di 256 elementi (detto GF(2^8)). Definire questa funzione su GF(2^8) ha il pregio di semplificare moltissimo le operazioni aritmetiche; infatti le somme e le sottrazioni in questo campo si effettuano con degli OR bit-a-bit mentre le moltiplicazioni e le divisioni si effettuando mediante due tabelle di lookup di 2^8 Bytes ognuna.

Infine, anche il protocollo PIV prevede l'esistenza di un vincolo temporale tra richiesta e risposta. Ovvero quando un sensore riceve una richiesta (*challenge*) deve inviare una risposta (*response*) entro un tempo prestabilito.

3.4 Osservazioni

I protocolli SWATT e Attestation hanno un difetto molto consistente: per garantire che ogni locazione di memoria sia letta almeno una volta, siccome gli indirizzi vengono generati casualmente, bisogna effettuare molti più accessi di quanti sarebbero necessari con una scansione lineare. In sostanza, siccome la memoria RAM è molto limitata non è possibile mantenere un vettore di bit nel quale indicare quali locazioni sono già state lette, è necessario continuare a generare indirizzi fino a raggiungere una determinata probabilità che ogni indirizzo sia stato letto almeno una volta.

Secondo quanto stabilito dal "teorema del collezionista" [6], se si fanno delle estrazioni casuali da un insieme di n elementi, per fare in modo che ogni elemento venga estratto almeno una volta, bisognerà effettuare $O(n \cdot \ln(n))$ estrazioni. In particolare, il teorema dice che se X è il numero di estrazioni che bisogna fare per riuscire ad estrarre ogni elemento almeno una volta, allora:

$$\Pr[X > c \cdot n \cdot \ln(n)] \leq n^{-c+1}.$$

Ovvero, la probabilità che X (numero di estrazioni che bisogna effettuare) sia maggiore di $c \cdot n \cdot \ln(n)$ è inferiore ad n^{-c+1}. Volendo fare una valutazione pratica, considerando che un sensore della famiglia *Berkeley Motes* possiede circa 58-132 KBytes di memoria, se gli accessi vengono effettuati con granularità di bytes, il numero di accessi da effettuare per avere una buona copertura della memoria sarà dell'ordine di:

$$n \cdot \ln(n) \approx 11 \cdot n \qquad (\text{per } n = 58 \text{ KB} = 59'392)$$

oppure:

$$n \cdot \ln(n) \approx 11.8 \cdot n \quad (\text{per n} = 132 \text{ KB} = 135'168).$$

Quindi bisognerà effettuare almeno 10 volte gli accessi che sarebbero necessari se la scansione della memoria fosse sequenziale. In aggiunta, con questo metodo non è

comunque possibile avere una garanzia di copertura totale della memoria, ma soltanto una ragionevole probabilità.
Inoltre, considerando l'aumento del tempo di esecuzione delle procedure e soprattutto considerando i vincoli imposti dai sensori riguardo al consumo energetico, risulta chiaro che adottare questo tipo di procedura implica una perdita di efficienza rispetto a procedure che effettuano la scansione in maniera sequenziale come, ad esempio, PIV.

La soluzione proposta da 'Attestation' [2], cioè di inviare una procedura diversa ad ogni richiesta, sembra offrire le maggiori garanzie di sicurezza. Tuttavia, la progettazione e realizzazione di un server capace di generare autonomamente delle procedure che soddisfino i criteri di cui sopra, è un obbiettivo molto complesso, come mostrato in [7]. Inoltre, l'implementazione delle tecniche sopra esposte nelle reti di sensori sarebbe resa più difficoltosa dalla semplicità degli stessi e del loro microprocessore (che può contare su un *instruction-set* molto ridotto). Nonchè, dovendo inviare la procedura unitamente alla richiesta e considerando i vincoli riguardanti il consumo energetico relativo alle trasmissioni, significherebbe dover fare un compromesso tra consumo energetico ed efficacia del protocollo. Ovvero, per limitare il consumo energetico bisognerebbe minimizzare la dimensione della procedura, ma è altresì ovvio che sarebbe molto difficile occultare il codice utile e aggiungere codice fasullo e meccanismi di automodifica del codice, se la procedura risultante dovesse comunque essere di dimensioni molto contenute. Infine, gli autori di Attestation affermano che la procedura genera indirizzi casuali finché *tutte* le locazioni di memoria siano state verificate almeno una volta. Ma ciò non risulta possibile nel casi dei nodi sensori, come verrà illustrato nel capitolo 5, perché i sensori non possiedono abbastanza memoria RAM per tenere traccia di quali locazioni sono state verificate e quali invece no.

La soluzione più ragionevole sembra essere quella proposta da PIV [3], sia perché è semplice da realizzare sia perché è quella che effettua la computazione nel minor tempo, dimostrandosi quindi la più rispettosa nei confronti di una condotta di risparmio energetico. Ciononostante, nella soluzione proposta dagli autori immaginando di applicare il protocollo alle reti di sensori, la dimensione della *challenge* è di circa 800-1200 Bytes, perché il protocollo prevede che il server invii il codice della procedura e i dati freschi da inserire nel calcolo della *hash* (questi ultimi constano di diverse centinaia di bytes) unitamente alla challenge. Gli autori però

specificano che la verifica di un sensore deve avvenire con frequenza molto bassa, in modo che questo costo complessivo durante la vita della rete non risulti troppo elevato.

Nonostante la procedura proposta da PIV sia probabilmente la più efficiente dal punto di vista della complessità computazionale e del tempo di esecuzione, è stato dimostrato che la procedura è parallelizzabile, come verrà mostrato anche in seguito. Quindi, se un avversario riuscisse a impadronirsi di due sensori potrebbe riprogrammarli in modo che il calcolo della funzione di *hash* per uno dei due sensori fosse fatto sfruttando l'ausilio dell'altro sensore. Il secondo sensore verrebbe usato sia per contenere porzioni di memoria che originariamente stavano sul primo sensore, sia per effettuare una parte della computazione.

4. Specifica dei requisiti e caratteristiche della piattaforma scelta per l’implementazione

Di seguito verranno analizzati in dettaglio tutti i requisiti che e le caratteristiche che dovrà avere il protocollo per la verifica dell’integrità. Questo elenco ragionato è stato ottenuto ponderando i vincoli imposti dai sensori e dalle reti di sensori, le caratteristiche già offerte da altri protocolli, nonché le loro debolezze.

Le reti di sensori sono strumenti che potenzialmente potrebbero avere una grande diffusione, e anche qualora non si trattasse di sensori ma di altri, generici, sistemi *embedded*, la tendenza attuale è quella di permettergli di comunicare per collaborare. Quindi, in ogni caso, nel prossimo futuro assisteremo ad una proliferazione di reti di piccoli agenti. La prospettiva di poter progettare un protocollo per la verifica dell’integrità diventa quindi molto allettante, anche considerando che non sempre gli avversari che si possono incontrare saranno persone malintenzionate i cui attacchi siano finalizzati ad ottenere qualcosa di concreto (introdursi in un area sorvegliata dalla rete di sensori oppure causare un danno materiale ad una struttura/edificio/ambiente evitando che la rete di sensori che la sorveglia lanci un allarme), ma gli avversari potrebbero anche essere dei software maliziosi come *troiani*, *worm* o *virus*, il cui unico scopo sia quello di diffondersi e/o di causare dei piccoli fastidiosi malfunzionamenti. Un protocollo per la verifica dell’integrità, in questi ambiti, permetterebbe di accorgersi in maniera preventiva delle modifiche che possono aver subito i sensori, escludendoli dalla rete per evitare che causino danni ed eventualmente avvertendo le persone responsabili della rete, affinché prendano i dovuti provvedimenti.

4.1 Requisiti del protocollo

4.1.1 Integrazione con le applicazioni e i servizi della rete

Il protocollo di verifica dell'integrità deve essere parte integrante della rete di sensori, e deve coesistere con le applicazioni e i servizi offerti dalla rete. Ciononostante, durante la verifica di uno o più sensori è possibile, se necessario, che vi sia una breve interruzione o un degrado dei servizi. Questo perché di fronte al pericolo di una manomissione, è stato scelto che il mantenimento di un buon livello di sicurezza ha la priorità sul livello dei servizi.

4.1.2 Autonomia

Oltre a coesistere con in servizi della rete, il protocollo di verifica dell'integrità deve avere un certo grado di autonomia. In particolare, il sistema deve essere in grado di verificare uno o più sensori ed eventualmente isolarli dal resto della rete senza necessitare di un intervento umano.

4.1.3 Soluzione *software-only*

Come è gia stato discusso, nella progettazione dei nodi sensori e in generale di *sistemi embedded*, la massima priorità è la riduzione dei costi di produzione. Per questo motivo i produttori dei sensori di tipo *Berkeley Motes* non utilizzano microprocessori in grado di offrire un supporto hardware per autenticare il software. Di conseguenza, il protocollo per la verifica dell'integrità che vogliamo progettare non deve necessitare di alcun supporto hardware aggiuntivo.

4.1.4 Verifica remota

E necessario poter verificare l'integrità dei sensori anche senza poter avere fisicamente accesso ad essi. Questa caratteristica non solo è di particolare importanza negli scenari in cui i sensori vengano collocati in ambiente nemico, ma risulta fondamentale anche in virtù del requisito di autonomia. Ovvero, trattandosi di un

problema di sicurezza relativo alle reti di sensori, è ragionevole costruire un protocollo che sfrutti le comunicazioni tra sensori per poter inviare/ricevere dati.

4.1.5 Server di autenticazione

Il protocollo che si vuole progettare deve includere l'esistenza di un server di verifica dell'integrità, che possa essere considerato un server di fiducia, nel senso che non possa essere compromesso da un avversario. Quindi il protocollo dev'essere di tipo *request-response*, e non di tipo *self-checking* o *self-hashing*. Ma questi ultimi sono ritenuti meno sicuri, come è già stato visto nel capitolo 3.

4.1.6 Paradigma *request-response*

Il server invia al sensore una richiesta, il sensore effettua una computazione e prepara la risposta, che poi inoltra al server. Nello stesso tempo il server effettua la stessa computazione in base alle informazioni in suo possesso (il server deve possedere un'immagine, o copia, della memoria del sensore). Dopo aver ricevuto la *response*, il server la confronta con il risultato ottenuto dalla propria computazione e valuta, anche in base al tempo che è trascorso tra l'invio della *request* e la ricezione della *response*, se il nodo è stato manomesso oppure no.

4.1.7 Configurazioni hardware note

Il server di verifica deve conoscere l'hardware dei sensori. In particolare deve conoscere la loro potenza di calcolo, la quantità di memoria che possiedono, il set di istruzioni del microprocessore e anche l'organizzazione dello spazio di memoria.

4.1.8 Immagini della memoria dei sensori

Il server di verifica deve conoscere l'esatto contenuto della memoria di tutti i sensori per cui è responsabile.

4.1.9 Copertura totale della memoria

La procedura di verifica deve poter individuare anche piccolissime modifiche al contenuto della memoria con certezza assoluta. Per fare un esempio, si immagini uno scenario in cui una rete di sensori venga collocata lungo il perimetro di un'area, per proteggerne l'integrità. I nodi disposti lungo il perimetro saranno dotati di sensori sensibili alle radiazioni infrarosse oppure ai rumori. In questo contesto, i sensori probabilmente saranno programmati per lanciare un allarme qualora una delle grandezze rilevate superi una certa soglia, ad esempio se il rumore nell'ambiente supera una certa potenza in dB oppure se viene rilevato un oggetto che emette radiazioni infrarosse. Un avversario che voglia entrare ed uscire con tranquillità da questa area, potrebbe limitarsi a modificare il valore di questa soglia, facendo in modo che, di fatto, i sensori non vadano mai in allarme.

Quindi, in un caso limite come quello appena esposto, un avversario potrebbe attaccare una rete di sensori modificando il valore di un singolo byte in un singolo sensore. Ed è per questo motivo è necessario che ogni locazione di memoria venga verificata, ovvero, che la procedura implementi la *copertura totale della memoria*.

4.1.10 Scansione pseudo-casuale della memoria

La procedura di verifica dovrà accedere alle locazioni di memoria in maniera pseudo-casuale. Infatti, se l'avversario non può sapere, a priori, in quale ordine verranno effettuati gli accessi alle locazioni di memoria, non potrà implementare una procedura di verifica parallelizzata che sia anche efficiente.

Effettuare gli accessi in memoria in maniera pseudo-casuale ha anche un altro pregio, che si manifesterebbe nel momento in cui l'avversario riuscisse a comprimere la memoria creando un piccolo spazio dove inserire i dati originali. In quel caso, essendo gli accessi in memoria pseudo-casuali, per evitare che la procedura acceda alle locazioni modificate, egli dovrebbe modificare la procedura stessa, inserendo almeno una istruzione di controllo del flusso (*if*). La procedura modificata, ad ogni accesso in memoria, dovrà controllare se l'indirizzo generato corrisponde ad una delle locazioni modificate, nel qual caso dovrà dirottare la lettura verso un altro indirizzo oppure prendere qualche altro provvedimento. In ogni caso, anche nel caso limite in cui l'avversario abbia modificato una sola locazione di memoria, questa istruzione *if* dovrà essere eseguita ad ogni accesso in memoria, rallentando

sensibilmente la procedura di verifica. Eventualmente, il sensore non sarà in grado di mandare la risposta al server entro la deadline temporale assegnata.

4.1.11 Procedura di verifica crittograficamente sicura

La procedura di verifica dev'essere progettata in modo tale che il sensore possa fornire la risposta corretta soltanto se il contenuto della sua memoria è uguale all'immagine contenuta nel server. Una procedura di verifica che possiede questa proprietà è detta *procedura di verifica sicura (secure verification procedure)*.

4.1.12 Procedura di verifica nota

La procedura di verifica che viene eseguita sui sensori deve essere nota ed il protocollo di verifica deve essere progettato in modo che non sia necessario avere la certezza che sui sensori venga eseguita la procedura *originale*, cioè la procedura di fiducia. Date queste premesse, un avversario che catturi un sensore tenterà di modificare anche la procedura che computa la hash.

Di conseguenza, la procedura che computa la hash deve essere progettata in maniera tale che, se il contenuto della memoria del sensore è stato modificato, il sensore non sia in grado di produrre la *response* corretta qualsiasi sia la procedura di verifica che risiede su di esso. Questa caratteristica può essere indicata anche con in nome di *resistenza ad attacchi di analisi statica del codice*. Se la procedura di verifica possiede questa caratteristica l'unico modo in cui un avversario potrebbe sovvertire il protocollo di autenticazione sarebbe aggiungere hardware al sensore (un attacco fisico di livello 3). In questo modo è possibile ottenere la resistenza ad attacchi fisici di livello 1 e 2.

4.1.13 Non parallelizzabilità

La funzione che calcola la *hash* non deve poter essere parallelizzata. Se ciò fosse possibile, un avversario potrebbe far collaborare più nodi per accelerare il calcolo della *hash*.

4.1.14 Non comprimibilità

Questo requisito, che forse è uno dei vincoli più forti, stabilisce che utilizzando uno schema software-only, affinché un protocollo per la verifica dell'integrità delle applicazioni possa funzionare correttamente l'avversario non deve poter comprimere in alcun modo i dati o il codice presenti nella memoria del sensore. Se ciò accadesse, l'avversario potrebbe modificare delle parti di memoria conservando però i dati originali nello spazio ricavato comprimendo la memoria. Successivamente, egli potrebbe far eseguire al nodo una procedura modificata in modo da accedere ad indirizzi errati, ed effettuare la *hash* accedendo alla copia dei dati originali che l'avversario ha alterato. Quindi, è necessario che la procedura di verifica sia incomprimibile, e purtroppo è necessario che anche l'applicazione lo sia. E questo è un requisito che quindi si trasmette anche alle altre librerie, routine e applicazioni installate sul sensore.

4.1.15 Resistenza ad attacchi di tipo *replay* e attacchi con *response* precalcolata

Il protocollo deve essere in grado di resistere ad attacchi di tipo *replay* ed attacchi con *response* precalcolata. Questa caratteristica si può ottenere facendo in modo che il server invii dei dati freschi (fresh) che abbiano una collocazione nel calcolo della *hash*. In questo modo, se il server invia dei dati sempre diversi insieme ad ogni richiesta, è impossibile che un nodo sfrutti un vecchio messaggio di *response* prodotto da un altro sensore oppure che sfrutti una *response* calcolata prima che il sensore venisse manipolato.

4.1.16 Resistenza ad attacchi in cui l'informazione fresca viene prevista o indovinata dall'avversario

L'informazione fresca che viene inviata al sensore al momento della richiesta di verifica dell'integrità non deve essere prevedibile. In sostanza, se un avversario facesse *spoofing* delle richieste del server, e venisse quindi a conoscenza di una o più chiavi *fresh*, egli non deve poter prevedere (o indovinare) le prossime chiavi che il server invierà unitamente alle prossime richieste.

4.1.17 Resistenza ad attacchi di manomissione di livello 1 e di livello 2

Per ingannare il protocollo di sicurezza, l'avversario deve essere costretto ad effettuare un attacco fisico di livello 3, ovvero deve apportare modifiche hardware ad uno o più sensori. Questo significa che il costo in tempo e risorse economiche per effettuare un attacco aumenterà, e la speranza è che l'avversario rinunci all'attacco perché quest'ultimo diverrebbe sconveniente.

4.1.18 Deadline temporale

Il server di verifica dovrà possedere un meccanismo per misurare il tempo trascorso tra l'invio della *challenge* e la ricezione della *response*. Questo meccanismo dovrà però tener conto del fatto che in una rete con molti nodi, il nodo potrebbe non avere un collegamento diretto col server, e quindi i pacchetti di dati che viaggiano dal server al nodo e viceversa potrebbero dover essere ricevuti e inoltrati da dei nodi intermedi (multi-hop). Inoltre, al momento di trasmettere un nodo potrebbe trovare il canale di trasmissione occupato oppure potrebbero verificarsi errori di trasmissione. Il server di verifica dovrà essere in grado di tener conto di queste eventualità. Ad esempio, se la *response* di un nodo dovesse arrivare con troppo ritardo, prima di negare l'accesso alla rete a quel nodo, il server dovrebbe sottoporlo a verifica ancora una o più volte.

Per avere una stima dell'intervallo di tempo da usare come limite temporale, possiamo considerare

$$T = RTT + e + \Delta,$$

dove RTT è il *Round Trip Time* (tempo impiegato da un pacchetto partito dal server, per attraversare la rete, giungere al nodo da esaminare, essere rispedito indietro e ritornare al server), e è una stima del tempo di esecuzione della procedura e Δ è una stima della latenza introdotta dai nodi intermedi che dovessero trovarsi in sovraccarico oppure causata da errori di trasmissione o da situazioni in cui il canale di trasmissione fosse occupato.

4.1.19 Meta-informazioni

La procedura potrebbe includere nel calcolo della hash anche delle informazioni relative allo stato interno del microprocessore. In questo modo si avrebbe un ulteriore

garanzia del fatto che, una determinata *response* può essere ottenuta soltanto eseguendo una determinata sequenza di istruzioni in un determinato ordine. Tuttavia, bisogna tener conto che introdurre questo tipo di proprietà nella procedura di verifica significa che il server, per calcolare a sua volta una *response* in base all'immagine del contenuto della memoria del sensore, dovrebbe simulare il comportamento del microprocessore del sensore. Questo significa che per il lato server sarebbe necessario costruire un emulatore o un simulatore. In ogni caso, i sensori "Berkeley Motes" non dispongono di memoria virtuale, di cache, o di *performance registers*, che potrebbero essere usati per questo scopo. Di conseguenza in un'implementazione per questo tipo di sensori non è possibile utilizzare meta-informazioni.

4.1.20 Attivazione della procedura di verifica

Il server di verifica dovrà procedere alla verifica di un sensore in tre casi:

- periodicamente
- quando un sensore nuovo viene aggiunto alla rete
- se rimane inattivo per un periodo di tempo ritenuto troppo lungo (abbastanza lungo da giustificare il sospetto che sia stato sottratto, modificato e riportato nella sua sede originaria)

inoltre, se nella rete di sensori è presente un sistema IDS (*Intrusion Detection System*), il server di verifica dovrà richiedere la verifica di un sensore se il nodo esibisce un comportamento ritenuto "strano".

E' doveroso osservare che, a seconda di come è stata progettata la rete di sensori, o meglio, a seconda della durata dei periodi di *sleep* dei sensori nonché dell'intervallo di tempo che può trascorrere senza che un sensore abbia bisogno di comunicare con altri sensori oppure con il centro di controllo (o di raccolta dati), potrebbe presentarsi uno scomodo problema. Ovvero, se un avversario catturasse il sensore, lo riprogrammasse e lo reinserisse nella sua pozione originale all'interno della rete in un tempo molto breve la rete non si accorgerebbe dell'assenza del sensore e quindi il meccanismo di verifica non verrebbe attivato. Ovviamente, questo problema si potrebbe risolvere diminuendo la lunghezza dei periodi di *sleep* dei sensori oppure dei periodi che loro possono trascorrere senza dover comunicare. In alternativa si potrebbero forzare delle verifiche periodiche a tutti i sensori.

4.1.21 Consumo energetico

La procedura di verifica che viene eseguita sui sensori deve essere breve, efficiente, non parallelizzabile e con basso consumo di memoria (si ricorda che i sensori hanno a disposizione pochissima memoria RAM). Ugualmente, la quantità di informazioni fresche che il server invia al sensore deve essere ridotta al minimo indispensabile.

E' utile osservare che nel caso in cui si utilizzi una procedura nota, questa può essere presente su tutti i sensori ed in questo modo non sarà necessario che il server la invii insieme ad ogni richiesta.

4.2 *Caratteristiche dei sensori Tmote Sky*

La piattaforma hardware che sarà utilizzata per lo sviluppo del protocollo di verifica remota dell'integrità sono i sensori Tmote Sky della MoteIv. Di seguito sarà presentata una panoramica delle caratteristiche hardware e software di tali sensori, evidenziando gli aspetti utili per questa applicazione.

4.2.1 Descrizione

Il Tmote sky è un modulo *ultra low power* progettato per essere impiegato nelle reti di sensori. In particolare, nelle applicazioni di monitoraggio. Il punto di forza è l'utilizzo di standard quali USB e IEEE 802.15.4, in modo da poter interagire con altri dispositivi. È dotato di sensori di umidità, temperatura e luminosità, ed è inoltre possibile aggiungervi delle schede esterne contenenti altri tipi di sensori. È completamente compatibile con i sensori di tipo Telos Revision B.

4.2.2 Caratteristiche hardware

- 250kbps 2.4GHz IEEE 802.15.4 Chipcon Wireless Transceiver.
- Interoperability with other IEEE 802.15.4 devices.
- 8MHz Texas Instruments MSP430 microcontroller (10k RAM, 48k Flash).
- Integrated ADC, DAC, Supply Voltage Supervisor, and DMA Controller.
- Integrated onboard antenna with 50m range indoors / 125m range outdoors.
- Integrated Humidity, Temperature, and Light sensors.
- Ultra low current consumption.
- Fast wakeup from sleep (<6μs).
- Hardware link-layer encryption and authentication.
- Programming and data collection via USB.
- 16-pin expansion support and optional SMA antenna connector.
- TinyOS support: mesh networking and communication implementation.

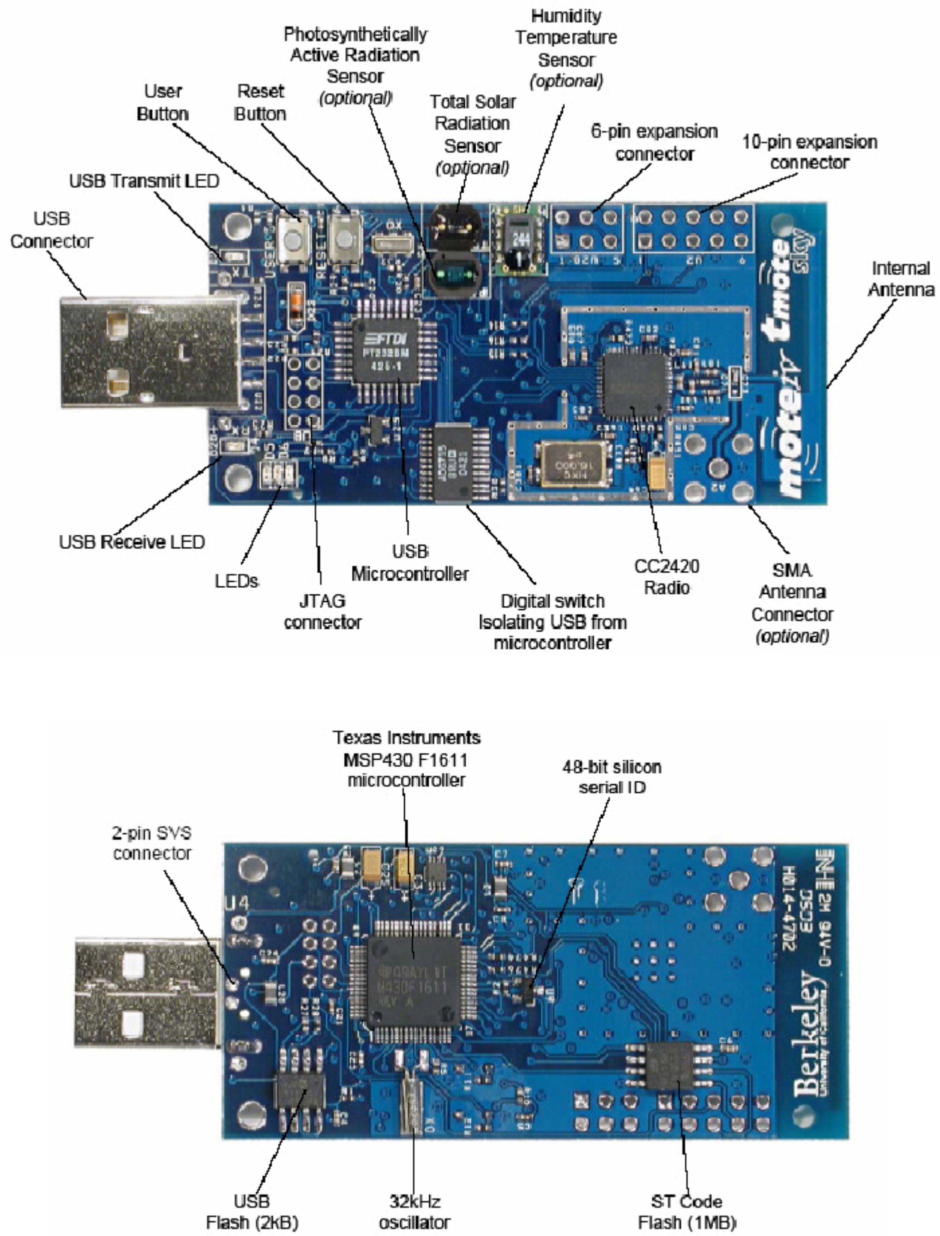


Figura 4.1: Fronte e retro del sensore Tmote sky.

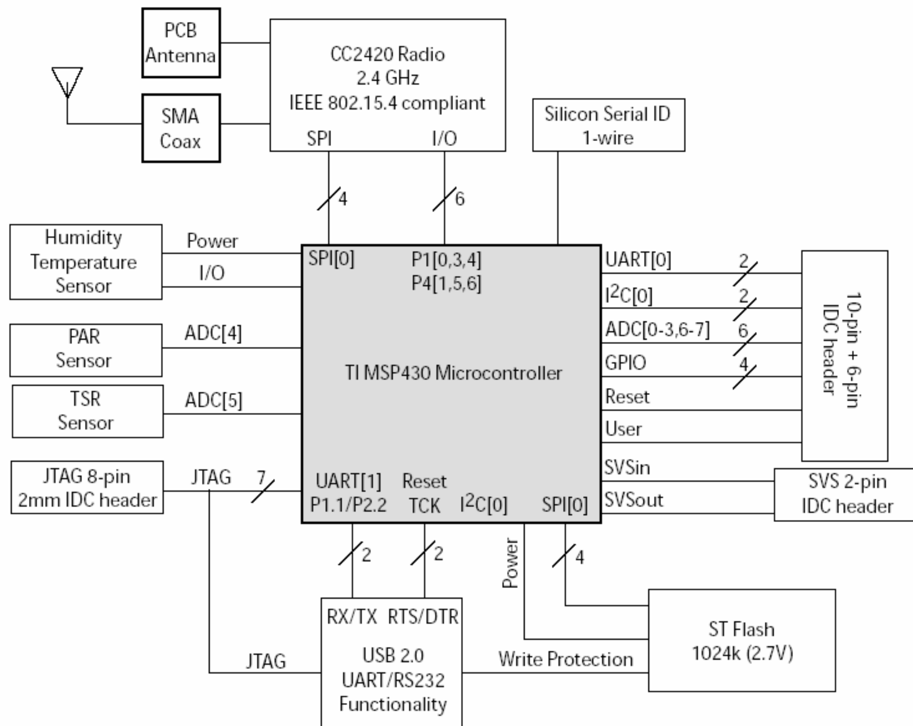


Figura 4.2: Diagramma a blocchi funzionali del modulo Tmote, i suoi componenti ed i bus.

4.2.3 Alimentazione

Il Tmote sky è alimentato da due batterie tipo AA. Il modulo è stato progettato della stessa dimensione di tali batterie e funziona con una tensione di alimentazione compresa tra 2,1V e 3,6V. E' importante far notare che durante la programmazione della flash del microcontrollore o della flash esterna la tensione deve essere superiore a 2,7V. Se collegato ad una porta USB riceve una tensione di 3,0V direttamente dalla porta e non è quindi necessaria alcuna batteria.

4.2.4 Microprocessore

Il basso consumo del Tmote sky è dovuto al bassissimo consumo del microcontrollore della Texas Instruments MSP430 F1611. Le sue caratteristiche sono: 10 KBytes di RAM, 48 KBytes di flash, 256 bytes di *information memory*. Questo microprocessore è caratterizzato da un bassissimo consumo sia durante il

funzionamento attivo che in modalità sleep. In riferimento alla dimensione della information memory, in alcuni testi è riportata la dimensione di 128 bytes. Questo perché la information memory è divisa in due segmenti da 128 bytes nei quali sono contenute le stesse informazioni. Sia per motivi di *fault tolerance* che per permettere di effettuare aggiornamenti. Infatti, essendo collocata nella memoria flash, come si vedrà più avanti, per aggiornarla è necessario cancellare un intero segmento e successivamente riscriverlo. Avendo a disposizione due segmenti uguali è possibile cancellarne uno e successivamente riscriverlo integrando i dati 'vecchi', contenuti nell'altro segmento, con i dati nuovi.

Il microcontrollore ha al suo interno un oscillatore digitale (DCO) che permette un funzionamento a 8MHz. Il DCO può commutare dalla modalità di sleep in 6µs, anche se, a temperatura ambiente necessita soltanto di 292ns. Quando il DCO è spento il microcontrollore sfrutta un oscillatore esterno a 32768Hz. Benché il DCO cambi la sua frequenza in base alla temperatura, è possibile calibrarlo sfruttando l'oscillatore esterno a 32kHz.

L' MSP430 ha 8 ADC esterni ed 8 ADC interni. Gli ADC interni possono essere utilizzati per monitorare la temperatura interna o la tensione di alimentazione.

Altre periferiche incluse sono: SPI, UART, digital I/O ports, Watchdog timer, Timers, 2 porte 12-bit DAC modle, Supply Voltage Supervisor, e 3 porte DMA controller.

4.2.5 Programmazione

Il Tmote sky viene programmato tramite porta USB utilizzando una versione modificata del MSP430 Bootstrap Loader, l'msp430-bsl. Questo perché il Tmote ha un unico circuito che previene il mote da falsi reset e proprio questo circuito rende necessario inviare una speciale sequenza al modulo in modo da programmarlo. In TinyOS, si utilizza l' utility "make tmote" per compilare il codice per il Tmote e l'utility "make tmote reinstall , *x*" per inserire l'eseguibile compilato nel sensore impostando un id pari a *x*.

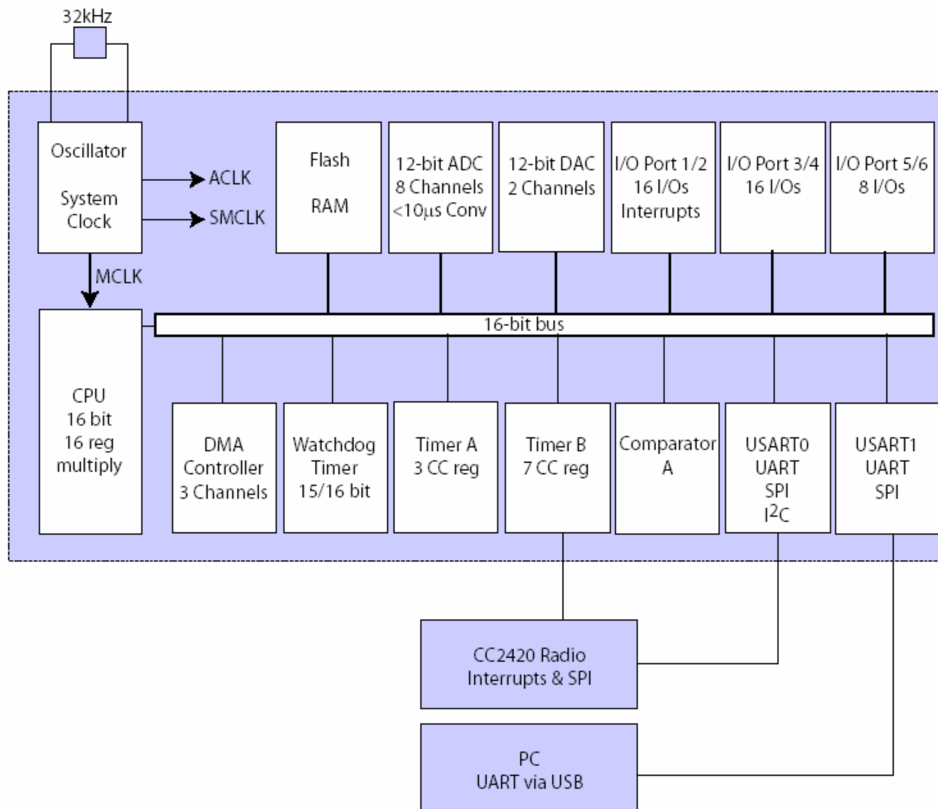


Figura 4.3: Diagramma a blocchi del TI MSP430 e sue connessioni.

4.2.6 Flash esterna

Il Tmote Sky monta un chip di memoria flash del tipo ST M25P80 40MHz. Questa memoria è comunemente detta 'flash esterna' ed ha una capacità di 1024KBytes suddivisi in 16 segmenti ciascuno di dimensione 64KBytes. La flash condivide le linee di comunicazione SPI con il CC2420, quindi deve essere prestata molta attenzione nella scrittura e lettura della flash tale da poter essere intervallata dalla comunicazione radio.

4.3 Organizzazione dello spazio di memoria di un sensore Tmote Sky

I protocolli per la verifica dell'integrità si basano solitamente su una procedura che, in qualche modo, preleva dei dati dalla memoria del sensore per ricavarne una *hash* o MAC (Message Authentication Code). Per progettare una tale procedura è indubbiamente necessario avere una chiara visione del modo in cui è organizzato lo spazio d'indirizzamento della memoria.

I sensori Tmote Sky sono dotati di un microprocessore Texas Instruments Mixed Signal Microcontroller MSP430F1611, il cui spazio di indirizzamento ha ampiezza di 16bit. I microprocessori della famiglia MSP430 implementano un'architettura Harvard, ovvero i dati e le istruzioni risiedono in due memorie separate. Questa architettura, tipicamente utilizzata nei DSP, garantisce migliori prestazioni poiché le due memorie possono lavorare in parallelo ma è più complessa da gestire. Inoltre, in un microprocessore progettato per avere un consumo molto basso, come nel caso degli MSP430, utilizzare una memoria flash per i programmi ed una memoria RAM per i dati implica dover alimentare soltanto la memoria RAM durante i periodi di sleep. E questo comporta un notevole risparmio energetico. Tuttavia, pur possedendo fisicamente un'architettura Harvard, l'MSP430 non prevede le classiche restrizioni tipiche di questa architettura. Ovvero, nell'MSP430 è possibile eseguire codice situato nella memoria dati, così come è possibile scrivere (anche se con particolari accorgimenti) nella memoria programma.

Il modello MSP430F1611 possiede, internamente, una memoria Flash di 48 KBytes ed una memoria RAM di 10 KBytes. Il suo spazio d'indirizzamento è organizzato secondo la tabella seguente 4.1.

MSP430F1611		
Memory Main: interrupt vector Main: code memory	Size Flash Flash	48KB 0FFFFh - 0FFE0h 0FFFFh - 04000h
RAM (Total) Extended Mirrored	Size Size Size	10KB 038FFh - 01100h 8KB 038FFh - 01900h 2KB 018FFh - 01100h
Information memory Boot memory	Size Flash Size ROM	256 Byte 010FFh - 01000h 1KB 0FFFh - 0C00h
RAM (mirrored at 018FFh - 01100h)	Size	2KB 09FFh - 0200h
Peripherals	16-bit 8-bit 8-bit SFR	01FFh - 0100h 0FFh - 010h 0Fh - 00h

Tabella 4.1: Organizzazione dello spazio d'indirizzamento del microprocessore MSP430F1611

La memoria Flash di 48 KBytes è accessibile tramite l'intervallo di indirizzi 0x4000h – 0xFFFFh. I 32 Bytes con indirizzi più alti della memoria Flash (0xFFE0h – 0xFFFFh) sono riservati per contenere la tabella degli interrupt e l'indirizzo di partenza (entry-point) dal quale iniziare l'esecuzione all'accensione del sensore. La memoria Flash è suddivisa in segmenti di 512 Bytes. Per una corretta gestione della memoria Flash, prima di accedere in scrittura ad una locazione di memoria, è necessario cancellare il suo contenuto. La cancellazione, però, può essere effettuata soltanto su interi segmenti oppure su tutto il contenuto della memoria.

Esiste poi una piccola porzione di memoria Flash di 256 Bytes, denominata Information memory, che è accessibile tramite l'intervallo di indirizzi 0x1000h-0x1100. Questa porzione di memoria è suddivisa in 2 segmenti da 128 Bytes. Questi due segmenti possono essere cancellati individualmente, oppure tramite l'operazione di cancellazione di tutta la memoria Flash. Questi due segmenti di memoria vengono utilizzati dal sistema operativo per contenere informazioni di configurazione che devono essere accessibili anche dopo eventuali operazioni di reset del sensore.

TinyOS utilizza tali settori, ad esempio, per custodire l'identificativo (ID) di rete del sensore.

La CPU può effettuare accessi in memoria Flash con granularità di byte o di word.

La memoria RAM, infine, è di 10KBytes ed è accessibile tramite l'intervallo di indirizzi 0x1100h-0x38FFh.

NOTA: Le informazioni utili della information memory si estendono soltanto da 0x1000 a 0x107e. In 0x107f è contenuto il numero di volte in cui la information memory è stata modificata, mentre in 0x1080-0x10ff c'è una copia del settore precedente (0x1000-0x107f).

5. Il protocollo PIV, l’attacco di parallelizzazione e la soluzione proposta

5.1 Il protocollo PIV

Il protocollo PIV, proposto in [3], prevede l’utilizzo di due server: un server di verifica dell’integrità, denominato PIVS ed un server di autenticazione denominato AS. Il server di autenticazione AS è necessario per proteggere i nodi da eventuali server di verifica falsi, che potrebbero inviare ai nodi una procedura maliziosa che, quando eseguita, potrebbe compromettere i nodi.

Il funzionamento del protocollo PIV può essere riassunto in questo modo: ogni sensore della rete è stato programmato con un’applicazione. Dopo l’installazione dell’applicazione la sua memoria ha un contenuto **x**. Da questa **x** viene ottenuto un *digest* $\mathbf{d}_v$ e quest’ultimo viene immagazzinato nel server PIVS. Quando si presenta la necessità di verificare un sensore, il PIVS invia al sensore una procedura grazie alla quale esso può calcolare una *signature* (o hash) $\mathbf{s}_p$. Questa *hash* viene calcolata dal sensore in base al contenuto della sua memoria **x**. Dopo aver calcolato la *hash*, il sensore la invia al server. Il server, a sua volta può calcolare $\mathbf{s}_p$ a partire dal *digest* $\mathbf{d}_v$ che esso già possiede. Confrontando la risposta ricevuta dal sensore con il valore calcolato dal server stesso, il server può decidere se il sensore è stato in qualche modo compromesso. In particolare, se le due *hash* coincidono il sensore è ancora integro, se invece esse sono diverse, il contenuto della memoria programma del sensore è cambiato, e quindi esso potrebbe essere stato compromesso.

Un sensore che cada nelle mani di un avversario può essere modificato a piacere. In particolare l’avversario può fare in modo che il sensore non esegua la procedura di *hash* genuina quando questa gli viene inviata. In questo modo l’avversario potrebbe fare in modo che il sensore restituisca al server una *hash* $\mathbf{s}_p$ precalcolata oppure una *hash* vecchia. Per evitare questo tipo di attacchi, gli autori di PIV hanno scelto di utilizzare una procedura per il calcolo della *hash* detta MAC o *keyed hash function.* Questa categoria di funzioni è progettata per prendere come parametro, oltre al messaggio di cui deve calcolare la hash, una chiave che ne modifica il risultato.

Una *keyed hash function* è una funzione $h_k(\mathbf{x})$ che possiede le seguenti proprietà:

- ***complessità***: data una chiave k ed un input $\mathbf{x}$, $h_k(\mathbf{x})$ è facile da calcolare
- ***compressione***: h_k trasforma un input $\mathbf{x}$ di lunghezza arbitraria in un output $h_k(\mathbf{x})$ di lunghezza prefissata.
- ***computation-resistance***: siano note un certo numero di coppie messaggio-hash $(\mathbf{x}_i, h_k(\mathbf{x}_i))$, è computazionalmente sconveniente riuscire a calcolare $(\mathbf{x}, h_k(\mathbf{x}))$ per qualsiasi $\mathbf{x}$ diverso da $\mathbf{x}_i$.

la proprietà di *computation-resistance* implica un'altra proprietà, quella della ***key-non-recovery***:

- ***key-non-recovery***: siano note un certo numero di coppie messaggio-hash $(\mathbf{x}_i, h_k(\mathbf{x}_i))$ calcolate con la stessa chiave k, è computazionalmente sconveniente riuscire a calcolare k.

La funzione di *hash* che gli autori hanno deciso di utilizzare appartiene ad una categoria speciale di funzioni MAC. Queste vengono chiamate *randomized hash functions* (o RHF). Queste funzioni, oltre a possedere le proprietà delle funzioni MAC viste sopra, hanno anche la particolarità di offrire due modi per calcolare $\mathbf{s}_p$, ovvero a partire da $\mathbf{x}$ oppure a partire da $\mathbf{d}_v$, come è stato visto sopra.
Sfruttando questa funzione, PIV può far eseguire ai sensori un codice che implementa la RHF, inviando loro il codice e una chiave casuale. Poi il server confronta il risultato ottenuto dal sensore con un risultato calcolato a partire da $\mathbf{d}_v$, utilizzando la stessa chiave.

5.1.1 La randomized hash function (RHF)

La RHF viene implementata applicando dei polinomi quadratici multivariati su dei valori appartenenti ad un campo chiuso di Galois $\mathcal{F} = GF(2^n)$ dove n viene generalmente scelto uguale a 8 per poi utilizzare operazioni aritmetiche tra bytes. Questa è una delle scelte dalle quali deriva l'elevate efficienza di questo protocollo. Questi campi chiusi sono già stati usati per implementare dei protocolli di cifratura a chiave pubblica. Inoltre, anche i polinomi quadratici multivariati sono già stati usati per realizzare funzioni *one-way*, quindi è plausibile utilizzarli in questo ambito proprio come funzioni *one-way*.

Sia **x** l'applicazione installata su un sensore, sia Λ la dimensione dell'applicazione espressa in bytes, e sia η la dimensione degli elementi di $\mathcal{F}$ espressa in bytes, $\eta = \lceil n/8 \rceil$. L'applicazione **x** viene partizionata in un insieme di B blocchi $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_B$, dove $\mathbf{x}_i = [x_{i,1}, x_{i,2}, \ldots x_{i,m}]^T$ è un vettore di dimensione $m \times 1$ ed $x_{i,j} \in \mathcal{F}$. Possiamo quindi esprimere la relazione $\Lambda = \eta \cdot m \cdot B$. Inoltre, PIV definisce il *digest* di un blocco $\mathbf{x}_i$ come una matrice $m \times m$ composta di tutti termini quadratici $x_{i,h} \cdot x_{i,k}$. Quindi $\mathbf{X}_i = \mathbf{x}_i \cdot \mathbf{x}_i^T = (x_{i,h} \cdot x_{i,k})$.

La RHF ha la seguente struttura algebrica: è definita nello spazio dei blocchi di dati $\mathcal{P} = \mathcal{F}^m$, dei digest $\mathcal{D} = \mathcal{F}^{m \times m}$, delle chiavi $\mathcal{G} = \mathcal{F}^B$ ed $\mathcal{H} = \mathcal{F}^{k \times m}$, e produce delle hash $\mathcal{Y} = \mathcal{F}^{k \times k}$ (con $k << m$).

Sia $\tilde{x}$ il programma da verificare, che consiste di B blocchi $\tilde{x}_1, \tilde{x}_2 \ldots \tilde{x}_B$.

L'algoritmo di *hash* è **Hash:** $\mathcal{G} \times \mathcal{H} \times \mathcal{P}^B \rightarrow \mathcal{Y}$ mentre l'algoritmo di verifica è **Vrfy:** $\mathcal{G} \times \mathcal{H} \times \mathcal{D}^B \times \mathcal{Y} \rightarrow$ { **pass**, **fail** }, tali che **Vrfy**(G,H,{$\mathbf{X}_i$}, **Hash**(G,H,{$\tilde{x}_i$})) = **pass** se $\mathbf{x}_i = \tilde{x}_i$, per ogni i=1,2, …B, e $G \in \mathcal{G}$ ed $H \in \mathcal{H}$ sono chiavi casuali (*random*) diverse ad ogni verifica.

Come è stato già detto, vi sono due modi diversi per calcolare la hash $Y = (y_{ij}) \in \mathcal{Y}$ e $y_{ij} \in \mathcal{F}$.

L'algoritmo **Hash** calcola Y da $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_B$:

$$Y = \sum_{i=1}^{B} g_i (Hx_i)(Hx_i)^T . \tag{5.1}$$

L'algoritmo **Vrfy** invece calcola Y da $\mathbf{X}_1, \mathbf{X}_2, \ldots \mathbf{X}_B$:

$$Y = H \left[\sum_{l=1}^{B} g_l X_l \right] H^T . \tag{5.2}$$

Per effettuare la verifica il sever PIVS ed il sensore S si scambiano i seguenti messaggi:

M1. S → PIVS: ID_S
M2. PIVS → S: G, H, codice (PIVC)
M3. S → PIVS: **Hash**(G, H, $\{\tilde{x}_i\}$)
M4. PIVS → S: **pass** / **fail**

A questo punto, il sensore S ha a disposizione la propria memoria $\tilde{x} = \{\tilde{x}_1, \tilde{x}_2 \ldots \tilde{x}_B\}$ mentre il server PIVS ha a disposizione i digest $\mathbf{X}_1$, $\mathbf{X}_2$, … $\mathbf{X}_B$. Di conseguenza, **Hash** e **Vrfy** effettuano I seguenti calcoli:
Hash inizializza Y' a 0 e quindi calcola Y' da $\tilde{x}_1, \tilde{x}_2 \ldots \tilde{x}_B$, cioè per ogni $l \in [1,B]$ calcola:

1. $\tilde{z}_l = H\tilde{x}_l$,
2. $Y_l' = \tilde{z}_l \tilde{z}_l^T$,
3. $Y' = Y' + g_l Y_l'$.

Vrfy che invece possiede $\mathbf{X}_1$, $\mathbf{X}_2$, … $\mathbf{X}_B$ per quel determinato sensore, calcola Y come segue:

1. $X = \sum_{l=1}^{B} g_l X_l$ per $1 \le l \le B$
2. $Y = HXH^T$.

Infine, dopo aver ricevuto Y', PIVS lo confronta con Y e decide se il sensore ha superato il test (Y'=Y) oppure se ha fallito (Y'≠Y).

5.1.2 Il protocollo

L'applicazione **x** presente su ogni sensore consiste di un insieme di *chiavi crittografiche*, un i*dentificatore di rete* (ID), il *codice*, ed una *porzione della memoria dati* (quella porzione che non viene utilizzata da PIVC). Immediatamente prima della verifica, questa porzione di memoria dati verrà inizializzata con dei valori casuali non comprimibili. Questo è necessario affinché l'avversario non possa

usare quello spazio per conservare le parti di codice che ha modificato, in forma originale.

Il PIVS mantiene due database: il primo, chiamato Digest_DB contiene tutti i digest $\mathbf{X}_i$ di tutti i sensori che sono stati collocati[1]; il secondo, chiamato PIV_DB contiene gli identificatori (ID) di tutti i sensori la cui integrità è stata verificata con successo. Il server deve aggiornare Digest_DB ogni volta che un nuovo sensore viene aggiunto alla rete, mentre deve aggiornare PIV_DB ogni volta che un sensore supera (o fallisce) il test di verifica dell'integrità.

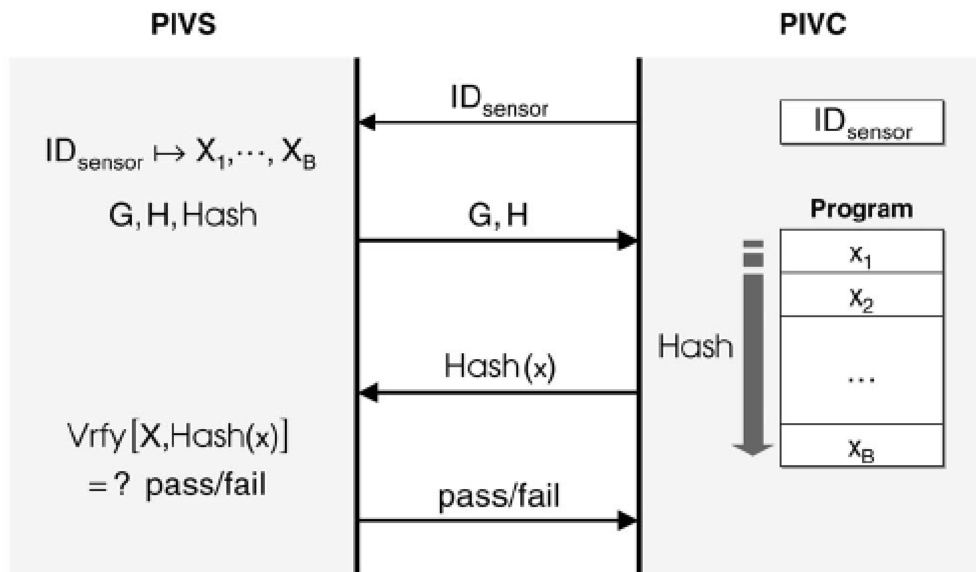


Figura 5.1: Una visione generale del protocollo PIV

All'atto della verifica di un sensore, il PIVS ed il sensore si scambiano i seguenti messaggi. Inizialmente il sensore deve mandare al PIVS il suo identificatore all'interno del messaggio M1. Dopo aver ricevuto questo messaggio, il PIVS, dopo aver scelto casualmente le chiavi G ed H, confeziona il messaggio M2 contenente le suddette chiavi ed il codice PIVC che implementa la funzione di *hash*. Dopo aver ricevuto il messaggio M2, il nodo S calcola Y' = **Hash**(G, H, $\{\tilde{x}_i\}$) secondo l'equazione (5.1) ed invia Y'al server (messaggio M3). Infine, dopo aver ricevuto il messaggio M3, il server accede al database Digest_DB, recupera i *digest* relativi al sensore S e calcola Y secondo l'equazione (5.2) e verifica se Y' = Y. Se queste sono uguali il server inserisce l'ID del sensore in PIV_DB ad invia ad S il messaggio **pass**.

[1] Questa affermazione è vera soltanto in teoria. In realtà qualora dei sensori avessero dei blocchi $\mathbf{x}_i$ a comune, cosa che è ragionevole supporre considerando che molti sensori avranno compiti simili, il server potrà contenere soltanto una copia dei digest $\mathbf{X}_i$ comuni.

Altrimenti, il PIVS cancella l'ID del sensore da PIV_DB, gli invia il messaggio **fail** e successivamente gli impedisce di accedere alla rete.

Un'altra caratteristica di PIV è che il server, dopo aver inviato M2, misura il tempo passato nell'attesa del messaggio M3 da parte del sensore. Se questo tempo supera un valore di soglia, il server termina la verifica. Questo vincolo è stato introdotto per rendere difficoltoso qualsiasi tentativo di fare in modo che il sensore si faccia aiutare da un altro nodo con potenza di calcolo superiore oppure per evitare che una procedura di calcolo alternativa possa produrre un risultato corretto pur avendo compromesso il contenuto della memoria.

Gli autori di PIV spiegano che la *hash* deve essere calcolata su tutto il contenuto della memoria di programma e della memoria dati del sensore. In particolare, le aree dati dell'applicazione devono essere riempite con dei dati incomprimibili in modo tale che sul sensore non rimanga spazio vuoto, nel quale un avversario potrebbe nascondere del codice malizioso oppure una versione originale della procedura **Hash**, qualora avesse intenzione di far eseguire una procedura modificata.

5.1.3 Analisi del livello di sicurezza di PIV

In riferimento a quanto è stato detto nel Capitolo 4, PIV possiede molti dei requisiti di sicurezza che deve soddisfare un protocollo di verifica dell'integrità *software-only*. Tra i più importanti, è necessario far notare che PIV garantisce la copertura totale della memoria, possiede una procedura di verifica crittograficamente sicura e nota. Inoltre, resiste ad attacchi di tipo *replay* o con *response* precalcolata perché il server invia delle informazioni *fresche* insieme ad ogni richiesta, resiste ad attacchi in cui l'informazione fresca viene prevista dall'avversario, presumibilmente perché il server contiene un generatore di numeri casuali che non è noto all'avversario. PIV prevede una deadline temporale ed è rispettoso dei requisiti sul consumo energetico, poiché, nonostante la richiesta di verifica preveda l'invio di molti bytes e la procedura sia *CPU-consuming*, la verifica dei sensori viene effettuata molto raramente.

PIV comprende il requisito di non comprimibilità visto nel Capitolo 4, anche se inteso limitatamente alla memoria dati. Questo potrebbe causare una falla di sicurezza, qualora l'avversario riuscisse a comprimere una parte dell'eseguibile dell'applicazione. Ma purtroppo questo è un problema che non ha soluzione. Vi sono dei *tool* di ottimizzazione del codice, che permettono di creare codice molto

compatto (ad es. *Gnu Superopt*), ma anche questi non possono fornire delle garanzie a riguardo. Ciononostante, è certo che questo tipo di attacco sarebbe molto difficile da attuare, e quindi anche in questo caso si fa affidamento al fatto che molti avversari non avranno le risorse sufficienti ad effettuare uno studio tanto approfondito della materia, esattamente come per gli attacchi che prevedano modifiche hardware dei sensori.

E' necessario notare, inoltre, che PIV non implementa una procedura di scansione pseudo-casuale della memoria.

Gli autori di PIV fanno un'analisi della sicurezza del protocollo in cui mostrano che l'unico di attaccare PIV consiste nel riuscire a ricavare dello spazio libero sul sensore in cui mantenere una copia originale dei dati che sono stati modificati per fornirli alla procedura **Hash** e poter così calcolare la *hash* corretta. Innanzitutto fanno vedere che, se un avversario modificasse alcuni blocchi di memoria e se modificasse la procedura **Hash** in modo da tener conto di queste modifiche, dovrebbe poi trovare spazio per conservare i blocchi a cui essi appartengono oppure i loro *digest*, nonché il blocco (o il *digest* del blocco) contenente la procedura **Hash** originale. Questa affermazione è lievemente incongruente, poiché, essendo i *digest* molto più ingombranti dei blocchi originali (si ricorda che i digest sono stati introdotti per permettere a PIVS di calcolare la *hash* in tempi più rapidi, pur comportando un maggiore uso di spazio su disco) in realtà basterebbe trovare spazio per salvare i dati stessi.

Infatti, gli autori mostrano che, se un avversario modificasse 3 blocchi, e dovesse trovare lo spazio per salvare i loro *digest*, dovrebbe trovare $(3+1)\cdot m^2\cdot\eta$ bytes, supponendo che la procedura **Hash** originale occupi 1 blocco di memoria. Se invece l'avversario volesse conservare soltanto i blocchi, piuttosto che i *digest*, gli sarebbe sufficiente trovare $(3+1)\cdot m\cdot\eta$ bytes di memoria. Per sventare questo tipo di attacco gli avversari consigliano di usare una tecnica che chiamano *interleaving*, che andrebbe utilizzata per comporre i blocchi. Ad esempio, se una porzione di programma di dimensione $B_i\cdot m\cdot\eta$ bytes venisse distribuita su B_i blocchi di dati (con $B_i = \frac{B}{N_c + 1}$, dove N_c è il numero di blocchi che sono stati modificati), anche un piccola modifica dell'applicazione comporterebbe cambiamenti su *almeno* B_i blocchi. Quindi l'avversario dovrebbe riuscire a ricavare almeno $B_i\cdot m\cdot\eta$ bytes (ad es. 36 Kbytes nel caso in cui B_i=384, m=96 ed η=1).

Quest'ultima affermazione risulta molto oscura. Gli autori non spiegano il funzionamento di questa tecnica di *interleaving*, e nuovamente ipotizzano che

l'avversario dovrebbe conservare gli interi blocchi che venissero così creati, invece dei singoli dati che sono stati modificati.

Quindi, in realtà, basterebbe che l'avversario salvasse i dati originali, e quindi lo spazio si ridurrebbe a $(N_c+1)\cdot m\cdot\eta$ (ovvero 4·96=384 bytes, nel caso dell'esempio visto sopra) bytes nel caso peggiore, cioè nel caso in cui l'avversario avesse modificato *tutti* i bytes contenuti in quegli N_c blocchi e nel blocco contenente la procedura **Hash**. E soprattutto, se, come è già stato fatto vedere, l'avversario si limitasse a modificare un singolo byte (ad esempio per cambiare una soglia di attivazione di un allarme), basterebbe poter salvare anche soltanto $(1+m)\cdot\eta$ bytes (ovvero 97 bytes, nel caso dell'esempio visto sopra), dove m è la dimensione del blocco contenente la procedura **Hash** originale.

Ciononostante, come verrà mostrato nel paragrafo successivo, se l'avversario manomettesse più di un sensore per organizzare un *parallel attack*, dovrebbe effettivamente copiare interi blocchi di dati sul secondo sensore, in modo da permettere a quest'ultimo di contribuire attivamente al calcolo della *hash*, occupandosi lui (il secondo sensore) di calcolare le sotto-*hash* Y' relative ai blocchi da esso posseduti.

Anche in questo caso, però, questa tecnica di *interleaving* non sarebbe efficace, perché avendo a disposizione un secondo sensore, non sarebbe affatto problematico conservare i $B_i\cdot m\cdot\eta$ bytes (36 KBytes, nell'esempio visto sopra) necessari ad effettuare l'attacco.

5.2 L'attacco di parallelizzazione

Il protocollo PIV nella sua versione esposta in [3] è vulnerabile ad un attacco di parallelizzazione (*parallel attack*). In questo attacco, un avversario manomette due sensori e li programma in maniera tale che soltanto uno dei sensori cerchi di rientrare nella rete, mentre l'altro servirà soltanto come nodo di supporto. Per comodità, chiameremo S_A il sensore che si candida per l'accesso alla rete e chiameremo S_B il sensore programmato per dare supporto durante il test. Nell'attacco che verrà esposto, durante il test di verifica dell'integrità del sensore S_A, il sensore S_B potrà svolgere una parte dei calcoli necessari per ottenere *Y'*, ottenendo un doppio beneficio: sobbarcandosi una parte del peso della computazione e soprattutto liberando una porzione della memoria di S_A, sul quale quindi, l'avversario potrà sia installare una procedura di *hash* modificata, ed eventualmente altre procedure maliziose che serviranno successivamente per portare attacchi ai dati o ai servizi della rete.

Tutto questo può essere ottenuto soltanto riprogrammando due sensori. Quindi con questo attacco si dimostra la vulnerabilità di PIV ad attacchi fisici di livello 2, cioè quella categoria di attacchi che non necessitano di effettuare modifiche dell'hardware dei sensori.

5.2.1 La vulnerabilità della RHF

Se un'applicazione **x** è composta da *B* blocchi di memoria $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_B$,sia $\mathcal{B} = \{1, 2, \ldots B\}$ l'insieme degli indici dei suddetti blocchi. Sia $\wp(\mathcal{B})$ l'insieme potenza di $\mathcal{B}$, ovvero l'insieme di tutti i sottoinsiemi di $\mathcal{B}$. Siano poi $A_1 \in \wp(\mathcal{B})$ e $A_2 \in \wp(\mathcal{B})$ tali che $A_2 = A_1^C$, ovvero A_2 sia l'insieme complemento di A_1 in $\mathcal{B}$. Quindi avremo che $A_1 \cup A_2 = B$ e $A_1 \cap A_2 = \varnothing$. A questo punto è facile vedere che l'equazione (5.1) può essere scomposta in questo modo:

$$Y' = \sum_{i=1}^{B} g_i (Hx_i)(Hx_i)^T = \sum_{i \in A_1} g_i (Hx_i)(Hx_i)^T + \sum_{i \in A_2} g_i (Hx_i)(Hx_i)^T . \tag{5.3}$$

Di conseguenza, se indichiamo Y'_{A_1} = **Hash**(G, H, $\{\tilde{x}_i \mid i \in A_1\}$) = $\sum_{i \in A_1} g_i (Hx_i)(Hx_i)^T$ e Y'_{A_2} = **Hash**(G, H, $\{\tilde{x}_i \mid i \in A_2\}$) = $\sum_{i \in A_2} g_i (Hx_i)(Hx_i)^T$ allora avremo che $Y' = Y'_{A_1} + Y'_{A_2}$.

5.2.2 I dettagli dell'attacco

Supponiamo che un avversario catturi due nodi S_A ed S_B. Supponiamo che egli modifichi l'applicazione del sensore S_A, ed in particolare che modifichi un certo numero di blocchi di memoria. Lo scopo dell'avversario è quello di permettere al sensore S_A di entrare nella rete anche se il contenuto della sua memoria **x**" è stato modificato rispetto al contenuto originale **x**. La difficoltà che l'avversario deve superare è che avendo modificato l'applicazione del sensore, questo non sarà in grado di superare il test di verifica dell'integrità. Una soluzione a questo problema potrebbe sfruttare la vulnerabilità della RHF vista nel paragrafo 5.2.1. La possibilità di spezzare il calcolo della *hash* in più parti indipendenti potrebbe essere utilizzata per distribuire il calcolo della *hash* su più nodi. Dopo aver stabilito quali blocchi di memoria saranno interessati dalla modifica che vuole effettuare, l'avversario potrebbe copiare i blocchi di memoria originali su un secondo sensore S_B e su questo secondo sensore egli potrebbe installare una versione della RHF modificata in modo tale da calcolare la parte di *hash* relativa soltanto a quei blocchi.

Ad esempio, supponiamo che l'avversario voglia manomettere l'applicazione installata su un sensore e supponiamo che questa modifica vada ad interessare i blocchi di memoria $\mathbf{x}_j$ con $j \in A_1$. Sia poi $A_2 = A_1^C$ l'insieme complemento di A_1 in $\mathcal{B}$, ovvero l'insieme dei blocchi che non sono stati modificati. Supponiamo che l'avversario modifichi la procedura RHF in modo da farle calcolare soltanto Y'_{A_2} e successivamente copi sul sensore S_B i blocchi originali (prima che venissero modificati) ed installi su S_B una procedura RHF che calcoli Y'_{A_1} e che la invii al sensore S_A. Durante il test di verifica dell'integrità, il sensore S_A dovrebbe calcolare Y'_{A_2}, cioè la *hash* relativa ai blocchi in suo possesso, mentre il sensore S_B, dopo aver in qualche modo ricevuto G ed H, dovrebbe calcolare Y'_{A_1}, cioè la *hash* relativa ai blocchi che sono stati copiati sulla sua memoria. Dopo aver calcolato Y'_{A_1}, il sensore S_B dovrebbe inviarla al sensore S_A, il quale, infine, dovrebbe semplicemente sommare Y'_{A_1} ed Y'_{A_2} per ottenere la Y' corretta da inviare al PIVS.

Chiaramente, per poter calcolare Y'_{A_1} il sensore S_B avrà bisogno di G ed H. Questi possono essere ottenuti in due modi, o posizionando il sensore S_B vicino al sensore S_A in maniera tale che esso possa ottenere il messaggio contenente G ed H per eavesdropping, oppure l'avversario potrebbe implementare in S_A una modifica che gli faccia inviare G ed H non appena li riceve.
I messaggi scambiati tra S_A ed S_B durante il test sarebbero i seguenti:

A1[2]. $S_A \rightarrow S_B$: G, H
A2. $S_B \rightarrow S_A$: Y'_{A_1}

E' ovvio che effettuare queste comunicazioni comporterebbe un'aumento del tempo passato per elaborare una risposta da mandare al PIVS. E questo, considerando il limite temporale imposto dal PIVS potrebbe rappresentare un problema. Ciononostante, bisogna considerare che il sensore S_A viene alleggerito del peso di dover calcolare Y'_{A_2}. Infatti, uno dei pregi di questo attacco è che i due sensori possono calcolare due porzioni distinte di *Y'* ottenendo così un risparmio nel tempo di esecuzione. Inoltre, l'avversario potrebbe copiare nel sensore S_B più blocchi $\mathbf{x}_i$ di quanti ne abbia modificati. Anzi, supponiamo che egli prepari il sensore S_B in modo che esso contenga esattamente metà dei *B* blocchi di cui l'applicazione è composta. E' evidente che in questo caso egli potrebbe sfruttare il secondo sensore per fargli effettuare esattamente metà dei calcoli necessari per ottenere *Y'*, e questo comporterebbe un grandissimo risparmio di tempo.

Ovviamente sarebbe necessario fare delle valutazioni pratiche per stabilire se sarebbe maggiore il risparmio nel tempo di esecuzione oppure il ritardo dovuto allo scambio di messaggi tra due sensori, ma considerando il caso in cui S_B possa fare eavesdropping del messaggio M2 inviato dal PIVS per far iniziare il test al sensore S_A e considerando che il messaggio A2 è di dimensioni molto ridotte ($k \times k$ bytes), è ovvio che i due sensori riuscirebbero sicuramente a produrre Y' entro i tempi prestabiliti.

In [3], nella parte dove viene descritta l'implementazione di PIV, viene riportato che la procedura **Hash** necessita di ~40s per essere eseguita e viene riportato che i tempi di trasmissione dei pacchetti comportano un ritardo di ~1s per ogni *hop*. In questo contesto, è facile vedere che se un avversario effettuasse un

[2] Questo messaggio deve essere spedito da S_A soltanto se S_B non può fare eavesdropping del messaggio M2 inviato dal server PIVS

attacco parallelo, potrebbe portare il tempo di esecuzione a poco più di 20s, avendo quindi un margine di tempo molto ampio durante il quale i due sensori potrebbero comunicare. Oltretutto, visto il risparmio di tempo di computazione, un sensore di supporto potrebbe servire, aiutandoli, anche sensori distanti alcuni *hop*.

Inoltre, anche se venisse usata la tecnica di *interleaving* proposta in [3], questo non influirebbe sull'attacco perché per l'avversario non rappresenterebbe un problema, ma anzi, sarebbe un vantaggio (come è stato appena mostrato), dover copiare sul secondo sensore più blocchi di dati.

5.2.3 Esempio di attacco

Nelle figure 5.2 e 5.3 sono schematicamente mostrati i due sensori utilizzati dall'avversario per effettuare l'attacco.

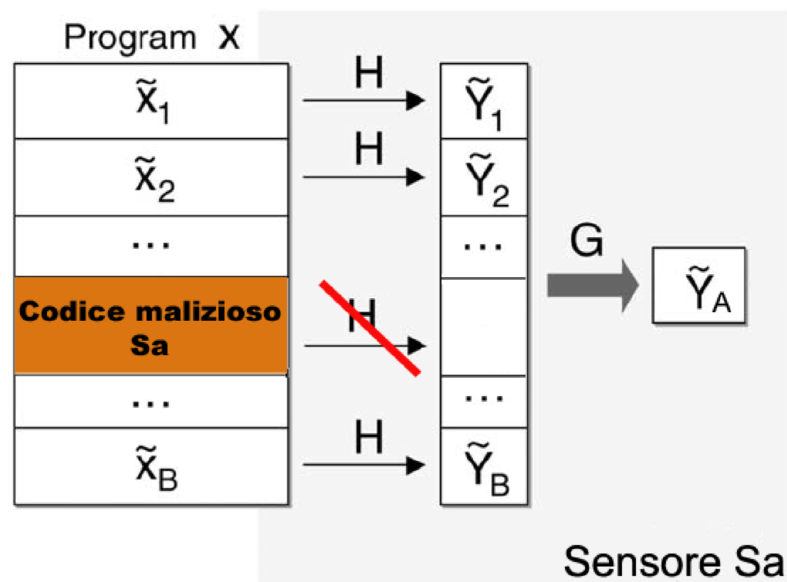


Figura 5.2: Il sensore S_A nell'attacco di parallelizzazione

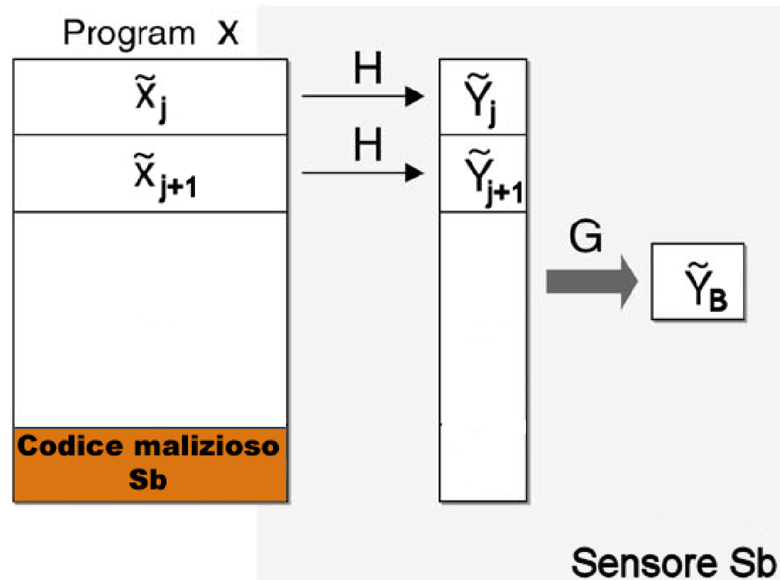


Figura 5.3: Il sensore S_B nell'attacco di parallelizzazione

E' possibile vedere che nel sensore S_A sono stati modificati i due blocchi $\tilde{\mathbf{x}}_j$ e $\tilde{\mathbf{x}}_{j+1}$. Il sensore è stato dotato di un codice malizioso che, effettuando il calcolo della *hash*, evita di calcolare $\tilde{Y}_j$ e $\tilde{Y}_{j+1}$. Nel sensore S_B, invece, sono stati copiati i due blocchi $\tilde{\mathbf{x}}_j$ ed $\tilde{\mathbf{x}}_{j+1}$ e vi è stata installata una versione di PIVC che effettua il calcolo delle sotto-*hash* $\tilde{Y}_j$ e $\tilde{Y}_{j+1}$ relative a questi due blocchi. Infine, se il sensore S_B invia al sensore S_A la *hash* $\tilde{Y}_B$ da esso ottenuta, ed il sensore S_A effettua la somma delle *hash* $\tilde{Y} = \tilde{Y}_A + \tilde{Y}_B$, può successivamente inviare questa somma al server PIVS e superare il test di verifica.

5.2.4 Conseguenze dell'attacco

Se un avversario effettuasse un attacco di parallelizzazione ad una rete di sensori che implementi il protocollo PIV, riuscirebbe a violare il suddetto protocollo e potrebbe inserire nella rete un numero arbitrario di sensori. Grazie a questi sensori egli potrebbe poi orchestrare, indisturbato, degli altri attacchi ai dati o ai servizi della rete.

5.3 Soluzione all'attacco

Di seguito verrà mostrata una variante del protocollo PIV, che è stata pensata per ovviare alla vulnerabilità che è stata esposta in 5.2. Questa vulnerabilità ha origine in una caratteristica della funzione di *hash* utilizzata, ovvero nel fatto che il calcolo effettuato nell'equazione (5.1) può essere separato in due o più porzioni indipendenti come mostrato nell'equazione (5.3).

A livello implementativo, questa debolezza compare perché un avversario può copiare uno o più blocchi di memoria $\mathbf{x}_i$ su un secondo sensore S_B, che serve soltanto come nodo di supporto, e che deve aiutare il sensore S_A nell'effettuare i calcoli delle Y'_i relative ai blocchi contenuti in esso.

5.3.1 La tecnica usata dagli altri protocolli

A questo punto è necessario trovare un metodo per impedire che la RHF possa essere parallelizzata. Per conseguire questo obiettivo è stata tratta ispirazione dagli altri due protocolli analizzati, SWATT [1] e Attestation [2], che dedicano alcune attenzioni allo sventare attacchi paralleli, come illustrato nel par.3.3.2. In particolare:

- La memoria non viene attraversata in modo sequenziale bensì in modo pseudo-casuale. In questo modo, se l'avversario copia una porzione di memoria su un altro sensore, questo avrebbe difficoltà a calcolare una parte della procedura, perché non è possibile prevedere a priori la sequenza degli indirizzi che verranno verificati.
- Il generatore pseudo-casuale di indirizzi è costruito in modo tale che gli indirizzi generati dipendano sia dallo stato interno del generatore sia dal valore istantaneo del checksum. Quest'ultimo ovviamente varia man mano che il calcolo prosegue. In questo modo, anche se un avversario copiasse una vasta porzione di memoria su un altro sensore con l'intento di fargli calcolare solo le parti di checksum relative agli indirizzi da esso posseduti, quest'ultimo non sarebbe in grado di riprodurre la stessa sequenza di indirizzi perché lo stato interno del generatore sarebbe diverso da quello del primo sensore.

5.3.2 La creazione dei blocchi $\mathbf{x}_i$

Di fatto, il motivo per cui la procedura PIVC risulta così facilmente parallelizzabile è che nella sua concezione originaria essa effettua il calcolo della *hash* sui blocchi di memoria $\mathbf{x}_i$, separatamente, ed infine effettua la somma di queste sotto-*hash*. I blocchi di memoria, come spiegato nel paragrafo 5.1.1 sono vettori $\mathbf{x}_i = [\mathbf{x}_{i,1}, \mathbf{x}_{i,2}, \ldots \mathbf{x}_{i,m}]^T$ di dimensione $m \times 1$ bytes, e sono composti da indirizzi di memoria contigui.

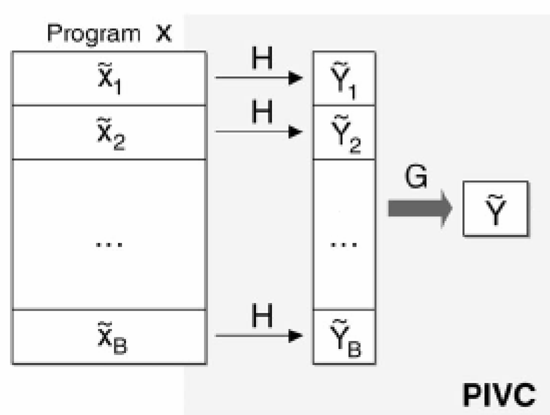


Figura 5.4: La creazione dei blocchi $\mathbf{x}_i$ secondo l'algoritmo PIVC originale.

Uno schema ideale, in termini di efficienza e di livello di sicurezza sarebbe riuscire ad unire i pregi di SWATT e PIV, ovvero effettuare una modifica a PIV tale che esso attraversi la memoria in modo pseudo-casuale. Essenzialmente questo sarebbe possibile se si riuscisse a modificare la composizione dei blocchi su cui PIVC effettua la *hash*. In questo modo si manterrebbero tutte le proprietà di efficienza e sicurezza della RHF.

In questo lavoro di tesi viene proposto un metodo alternativo per la creazione dei blocchi di dati $\mathbf{x}_i$ sui quali viene calcolata la RHF. In particolare, il metodo proposto crea i blocchi di dati a partire da indirizzi di memoria selezionati in maniera pseudo-casuale. Questa operazione può essere vista come una *virtualizzazione* dei blocchi di dati. La RHF agisce esattamente nello stesso modo in cui agiva nella versione originale di PIV, ma in questo caso, i blocchi che le vengono passati vengono confezionati al momento, prelevando dati da locazioni sparse all'interno della memoria.

Questa virtualizzazione verrà effettuata dinamicamente e sarà sempre diversa poiché dipenderà anch'essa dalle chiavi di inizializzazione che PIVS invia al sensore nel messaggio M2, ovvero nel messaggio che fa iniziare il test di verifica.

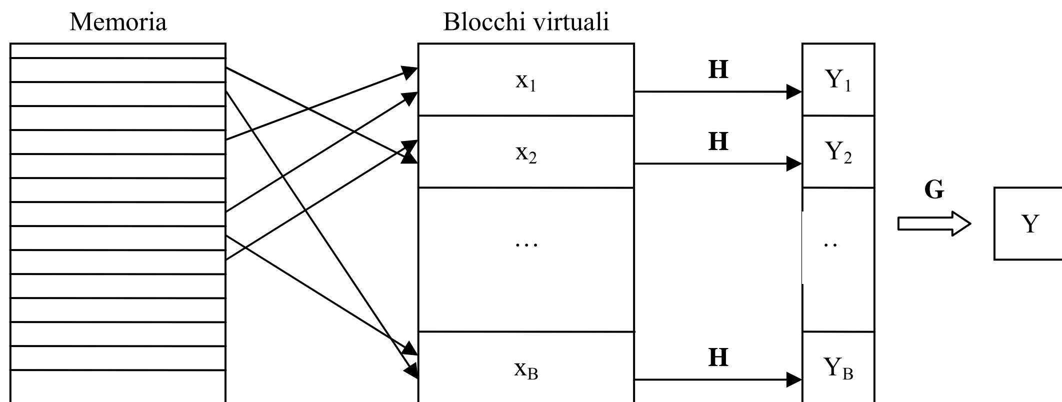


Figura 5.5: Virtualizzazione dei blocchi $\mathbf{x}_i$ e introduzione della casualità

Implementando questo accorgimento, per un avversario diviene impossibile implementare una variante di PIV parallelizzata che sfrutti due o più sensori come mostrato nella sezione 5.2. Infatti, se l'avversario modificasse il contenuto di alcune locazioni di memoria in un sensore S_A, sarebbe impossibile sapere a priori a quale blocco $\mathbf{x}_j$ apparterranno queste locazioni, e di conseguenza sarebbe impossibile trasferire tale blocco (cioè i bytes non modificati) sul sensore di supporto S_B.

Lo schema per la creazione dei blocchi $\mathbf{x}_i$ dovrà soddisfare tre requisiti:

1. gli accessi in memoria dovranno essere casuali o pseudo-casuali, quindi non prevedibili a priori.
2. dovrà dare la garanzia che ogni locazione di memoria sarà letta almeno una volta.
3. dovrà limitare il numero di accessi totali in memoria ad un numero accettabile dal punto di vista del tempo di esecuzione. Questo perché, nonostante la verifica dell'integrità di un sensore sia un'operazione che verrà eseguita molto sporadicamente, per evitare che influisca troppo sulla durata della batteria dei sensori nonché sulla qualità dei servizi, la durata di un'operazione di verifica dovrebbe rimanere dell'ordine di qualche decina di secondi.

5.3.3 Alcune tecniche per comporre i blocchi $\mathbf{x}_i$

Il metodo più semplice sarebbe comporre ogni blocco $\mathbf{x}_i$ con locazioni di memoria scelte in modo completamente casuale. Ma così facendo ci sarebbero sicuramente ripetizioni ed omissioni: ovvero, una singola locazione di memoria potrebbe apparire in più di un blocco ed inoltre, altre locazioni di memoria potrebbero non essere mai selezionata. Questo violerebbe il requisito di copertura totale della memoria. Anche aumentando artificiosamente il numero di blocchi, cioè modificando la procedura di calcolo della *hash* in modo da includere un numero di blocchi B' (con $B' >> B$, dove $B = \frac{\Lambda}{m \cdot \eta}$), se questi blocchi venissero creati in maniera completamente casuale, la probabilità che ogni locazione venga acceduta almeno una volta aumenterebbe secondo la legge descritta dal teorema del collezionista [6], ma non diventerebbe mai unitaria. Quindi, come per SWATT ed Attestation, non si avrebbe una garanzia della copertura totale della memoria e, soprattutto, l'aumentare drasticamente il numero di accessi alla memoria comporterebbe un notevole aumento del tempo di esecuzione e un conseguente aumento del consumo energetico.

Un secondo metodo potrebbe sfruttare una classica implementazione algoritmo che genera numeri casuali appartenenti ad un intervallo, senza ripetizioni. In questa implementazione viene utilizzato un vettore di bit ove vengono indicati i numeri che sono stati già estratti. Ad ogni estrazione, si verifica se quel numero è già uscito. Se il numero non è ancora stato estratto, lo si considera 'buono' e nel vettore di bit viene marcato il bit corrispondente; se invece, il numero era già stato estratto (cioè se il bit corrispondente era già stato marcato) si procede a selezionare un altro numero. Analogamente, se si utilizzasse questo algoritmo, si potrebbe allocare un vettore di bit in cui indicare quali locazioni di memoria sono già state analizzate e quali, invece, ancora non sono state analizzate. Purtroppo questa tecnica non è implementabile sui sensori che sono stati scelti come *target* per questo protocollo. Infatti, vi sono dei problemi di ordine pratico. In un sensore quale il Tmote-Sky, con 48 KBytes di memoria flash e 10 KBytes di memoria RAM, ci sono 59'392 locazioni di memoria totali. Un vettore di bit che riservasse 1 bit per ogni locazione di memoria dovrebbe occupare 7'424 bytes di memoria RAM. Nel caso del Tmote-Sky, se si riuscisse ad ottenere una procedura **Hash** sufficientemente compatta la memoria RAM del sensore sarebbe sufficiente a contenere il vettore di bit. Ma su altri modelli questo schema non potrebbe funzionare, come ad esempio nel Mica2, che possiede soltanto 4 KBytes di memoria RAM e 128 KBytes di memoria flash. E' stata considerata

anche l'ipotesi di collocare questo vettore di bit in memoria flash, ma questo non sarebbe possibile perché comporterebbe una perdita di efficienza enorme, sia dal punto di vista della complessità computazionale che dal punto di vista del consumo energetico. Infatti la politica di gestione di questa memoria prevede che prima di ogni scrittura (quindi nel nostro caso, prima di dover scrivere ogni singolo bit) è necessario cancellare (*erase*) l'intero segmento in cui è contenuto il byte che si vuole scrivere. Quindi, per aggiornare un bit all'interno del vettore di bit, sarebbe necessario innanzitutto copiare l'intero segmento in un'altra zona di memoria, per poi cancellarne il contenuto ed infine riscriverlo interamente, a partire dalla copia che ne era stata fatta ed aggiungendo anche il bit che si voleva scrivere *in primis*. Questa soluzione comporterebbe un overhead computazionale ingente, ma è possibile pensare anche ad un'altra soluzione, ovvero mantenere costantemente una copia dell'intero vettore di bit ed utilizzare la copia per integrarla con i nuovi dati. Questa seconda tecnica viene utilizzata dal sistema operativo TinyOS per gestire l'*information memory*.[3] In ogni caso, questa seconda soluzione risulterebbe molto inefficiente dal punto di vista dell'ingombro, perché dovendo mantenere una copia dell'intero vettore di bit l'occupazione di memoria sarebbe raddoppiata.

In ogni caso, ciò che renderebbe questa tecnica completamente impraticabile è il fatto che, almeno nei Tmote Sky, la scrittura in memoria flash necessita di una tensione di alimentazione di almeno 2.7V come esposto in [19], e questo impedirebbe di poter effettuare la verifica dell'integrità di quei sensori la cui batteria non sia più al massimo della carica. Di conseguenza è necessario scartare questo schema perché non sarebbe sufficientemente applicabile.

Esistono poi delle tecniche che, dato ad esempio un insieme di q elementi, sono in grado di generare tutte le permutazioni di elementi appartenenti all'insieme. Questo metodo potrebbe essere sfruttato per generare i blocchi di dati $\mathbf{x}_i$ da utilizzare con l'algoritmo **Hash**. Ovvero, una volta scelta e determinata una permutazione, i primi m elementi della permutazione verrebbero usati per comporre $\mathbf{x}_1$, i successivi m elementi verrebbero usati per comporre $\mathbf{x}_2$ e così via. Purtroppo, però, anche in questo caso un'implementazione reale di questo metodo avrebbe bisogno di molta più memoria RAM di quanta ne possiedono i sensori che si vogliono utilizzare, come mostrato in [18].

[3] La procedura per effettuare scritture in memoria flash è esposta anche nel Capitolo 4, nella descrizione dello spazio di indirizzamento del microprocessore MSP430.

Per soddisfare i tre requisiti visti nel paragrafo 5.3.2 e soprattutto per limitare la quantità di memoria RAM necessaria alla funzione di hash, è possibile pensare ad un approccio ibrido tra i primi due schemi visti in precedenza.
L'algoritmo PIVC originale mette in atto la verifica effettuando i calcoli su blocchi di dati $\mathbf{x}_i$ di *m* bytes ognuno. I blocchi sono costituiti da locazioni di memoria consecutive. La versione modificata dell'algoritmo PIVC che viene qui proposta e che chiameremo **PIVm**, suddivide la memoria in *partizioni* uguali, la cui dimensione è esattamente quella dei blocchi utilizzati da PIVC (cioè di *m* bytes ognuna). Supponendo che la dimensione totale della memoria sia di Λ bytes, questa può essere suddivisa in $B = \Lambda/m$ partizioni di m bytes. Le partizioni sono composte da locazioni di memoria con indirizzi contigui. In figura 5.6 è possibile vedere la suddivisione della memoria in partizioni.

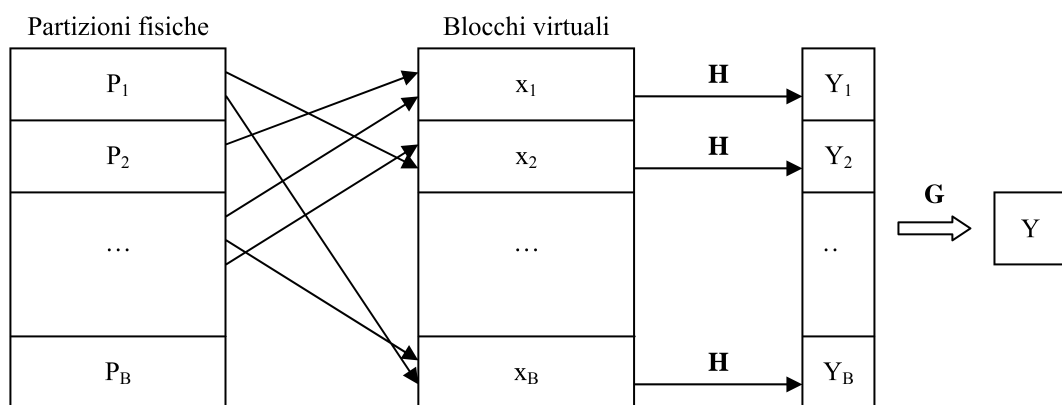


Figura 5.6: La suddivisione della memoria in partizioni

Per calcolare la *hash*, ciclicamente, PIVm seleziona (in maniera pseudo-casuale) *m* partizioni sempre diverse. Da queste partizioni, effettuando un'operazione che chiameremo *interleaving*, PIVm crea *m* blocchi di dati. Questa tecnica di *interleaving* è simile a quella proposta dagli autori di PIV [3], ma si differenzia da quest'ultima perché viene fatta in modo dinamico, ovvero, una porzione di programma qualsiasi, verrà distribuita su molti blocchi di dati, ma questi blocchi saranno diversi ad ogni esecuzione di PIVm. Di fatto, data una porzione di programma, un avversario non potrà sapere in anticipo dentro quali blocchi di dati finiranno i bytes che la compongono.

Per spiegare in maniera dettagliata questa procedura, diciamo che PIVm seleziona *m* partizioni con le quali comporrà *m* blocchi di dati. Ogni blocco di dati così ottenuto contiene una e soltanto una locazione proveniente da ognuna delle *m* partizioni. Inoltre, ogni locazione appartenente ad una delle *m* partizioni verrà inclusa in uno e soltanto uno blocco di dati $\mathbf{x}_i$. Questo avviene ciclicamente nel senso che, ad ogni ciclo di PIVm vengono selezionate *m* partizioni grazie alle quali vengono confezionati *m* blocchi di dati. Successivamente vengono calcolate le Y'_l relative a questi blocchi. Al termine di queste operazioni vengono selezionate le successive *m* partizioni e così via.

Nel caso in cui *B* non sia perfettamente divisibile per *m*, dopo aver selezionato tutte le partizioni sarà necessario selezionarne ancora *n* (con $n=B-m\cdot\lfloor B/m \rfloor$), scelte a caso fra tutte le partizioni, per completare l'ultimo gruppo di *m* partizioni.

Per esplicare meglio questa procedura, vediamola brevemente in pseudocodice. B è il numero totale di partizioni, m è il numero di bytes che compongono ogni blocco di dati ed è anche il numero di partizioni che vengono selezionate ad ogni ciclo:

Algoritmo PIVm

```
int         index:  (0 < index < B-1)
int         currentPartitions[m]
boolean     usedPartitions [B]

for (questo ciclo viene eseguito ⌈B/m⌉ volte) {
        <seleziona pseudo-casualmente m partizioni che non
        sono ancora state usate, e nel caso in cui durante l'esecuzione
        si esauriscano le partizioni non usate, ne seleziona ancora n
                        in modo da completare l'ultimo gruppo>;
        <inserisce gli indici delle partizioni selezionate
                                        in un array currentPartitions>;
        <marca le partizioni selezionate in un vettore usedPartitions>;

        for (i=0; i<m; i++) {
                < compone il blocco di dati x_i prelevando 1 byte
                  da ognuna delle partizioni indicate in currentPartitions >[4]
                <calcola Y_i = (H x_i)(H x_i)^T >;
                <aggiorna Y += g_i * Y_i >;
        }
}
```

[4] Questa operazione, chiamata operazione di *interleaving*, verrà esplicata in seguito.

OSSERVAZIONE:
- *currentPartitions* è un vettore di interi, che viene usato per contenere gli indici delle partizioni.
- *usedPartitions* è un vettore di booleani (o di bit, nell'implementazione effettiva), che riserva un booleano (o bit) per ognuna delle *B* partizioni. Questo vettore viene usato per marcare le partizioni che sono già state analizzate

I blocchi di dati così ottenuti sono costituiti da locazioni non prevedibili a priori, poiché le partizioni da cui sono stati creati, vengono scelte in maniera pseudo-casuale. Inoltre, su questi blocchi di dati vengono effettuate le stesse operazioni algebriche che vengono effettuate nel caso di PIV. Questo ci permette di non perdere tutte le proprietà crittografiche della RHF.

5.3.4 L'operazione di interleaving

La composizione di un blocco $\mathbf{x}_i$ effettuata prelevando bytes dalle partizioni contenute in *partitionIndex* viene chiamata operazione di *interleaving*. Questa operazione, il cui scopo è quello di creare il blocco *i*-esimo a partire dalle *m* partizioni selezionate, inizia prelevando il primo byte della *i*-esima partizione selezionata, e successivamente prelevando i byte successivi, dalle partizioni che seguono la *i*-esima, all'interno del vettore *partitionIndex*.

Operazione di interleaving

```
int          currentPartitions[m]

for (k=0; k<m; k++) {
        xi[k]=k-esimo byte della part. Numero currentPartitions[(i+k)mod m];
                // componiamo il blocco di dati xi prelevando
                // un byte da ognuna delle partizioni
                // indicate in currentPartitions[]
}
```

Per fare un esempio, supponiamo, in un'ipotesi semplicistica, che la dimensione dei blocchi per il PIVC e quindi anche la dimensione delle partizioni siano di m=4 bytes. Supponiamo inoltre che la dimensione totale della memoria sia di Λ=32 bytes e che quindi essa venga suddivisa in $B = \Lambda/m = 32/4 = 8$ partizioni. Supponiamo che ad un certo punto l'algoritmo abbia selezionato le partizioni 4, 1, 3 e 7.

Queste 4 partizioni verranno usate per creare 4 blocchi di dati, come illustrato anche in figura 5.7. In particolare, il primo blocco ($\mathbf{x}_1$) verrà creato prendendo il primo byte della partizione 4, il secondo byte della partizione 1, il terzo byte della partizione 3 ed il quarto byte della partizione 7.

Il secondo blocco ($\mathbf{x}_2$) verrà creato prendendo il primo byte della partizione 1, il secondo byte della partizione 3, il terzo byte della partizione 7 ed il quarto byte della partizione 4.

Analogamente accadrà per il terzo e il quarto blocco ma i bytes verranno prelevati a partire dalle partizioni 3 e 7 rispettivamente.

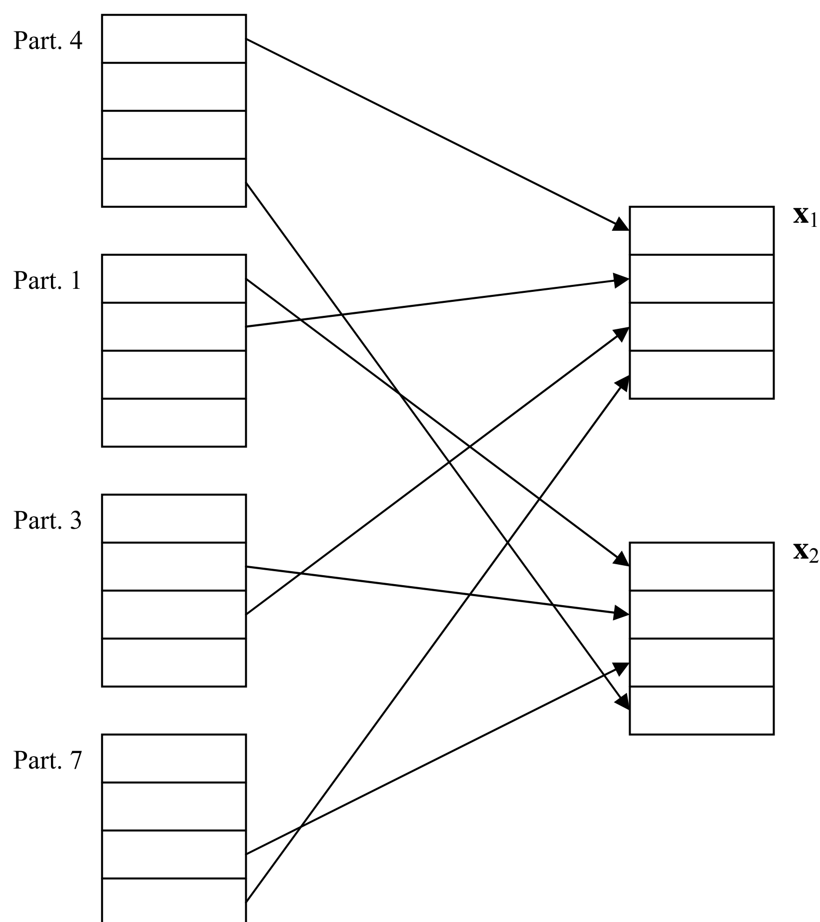


Figura 5.7: L'operazione di interleaving

5.3.5 Misure anti-parallelizzazione

Le misure all'algoritmo di PIVC che abbiamo visto finora servono a creare in modo dinamico e pseudo-casuale i blocchi di dati $\mathbf{x}_i$ sui quali viene effettuata la *hash*. Ciononostante, se un avversario che abbia catturato due sensori copiasse sul secondo sensore S_B una porzione molto vasta della memoria di S_A, questo avversario potrebbe sperare che durante l'esecuzione dell'algoritmo, vengano generati uno o più blocchi di dati i cui bytes siano tutti presenti nella memoria di S_B. Se questo accadesse, S_B sarebbe in grado di calcolare qualcuna delle sotto-*hash* Y'_i e, con un po di fortuna, considerando che prima di marchiare un sensore come irrimediabilmente compromesso il PIVS lo sottopone a più di un test, S_B potrebbe riuscire a calcolare proprio le Y'_i relative ai dati che su S_A sono stati modificati ma che su S_B sono ancora presenti in forma originale. In questo modo i due sensori S_A ed S_B potrebbero riuscire ad ingannare il server PIVS.

Per evitare attacchi di questo tipo, è stato deciso di utilizzare un'ulteriore misura di sicurezza, ovvero di modificare il generatore di numeri pseudo-casuali in modo da includere nella generazione della sequenza anche il valore istantaneo di Y (oppure di uno degli ultimi Y'_i) che sono stati calcolati. In questo modo è possibile impedire al sensore S_B di riprodurre l'esatta sequenza di numeri utilizzati per selezionare le partizioni, e quindi di aiutare S_A nell'effettuare il calcolo della *hash*.

5.3.6 Generatore di numeri pseudo-casuali

Per poter funzionare, l'algoritmo PIVm avrà bisogno anche di un generatore di numeri pseudo-casuali *crittograficamente sicuro*. Un algoritmo candidato a questa mansione è sicuramente RC4, che si presta ad essere implementato su CPU a 8 bit.

Il codice PIVC che veniva eseguito sui sensori nella versione originale del protocollo PIV non prevedeva un tale generatore.

5.3.7 Ingombro in memoria dell'algoritmo PIVC e dell'algoritmo PIVm

L'algoritmo PIV originale necessita delle seguenti variabili (o costanti) in memoria RAM. Tra parentesi è indicata la loro dimensione:

- $\mathbf{x}_i$ [ ] (*m* bytes)
- $\mathbf{z}_i$ [ ] = H · $\mathbf{x}_i$ (*m* bytes)
- *g*[ ] (*B* bytes)
- H [ ] [ ] (*k***m* bytes)
- Y'_i [ ] [ ] (*k***k* bytes)
- *Y'*[ ] [ ] (*k***k* bytes)
- Tabelle per le moltiplicazioni in GF(2^8) (512 bytes)

L'algoritmo PIV modificato, invece, oltre al codice aggiuntivo necessario per implementare l'interleaving, la selezione delle partizioni e il generatore di numeri pseudo-casuali, ha bisogno delle seguenti variabili:

- *currentPartitions* [*m*] (vettore di m interi a 16 bit)
- *usedPartitions* [*B*/8] (vettore di ***bit*** di dimensione B/8 bytes)

E' facile notare come la modifica introdotta all'algoritmo PIV comporti un ingombro ulteriore molto ridotto. Ad esempio, nel caso dei sensori Tmote-Sky per i quali, come abbiamo visto, la dimensione totale della memoria è di Λ=58 KByes, supponendo di utilizzare blocchi di memoria da *m*=128 bytes, considerando che *B*= Λ/*m*=464 l'ingombro dovuto al PIV modificato ammonta a:

- $\mathbf{x}_i$ [ ] (*m*= 128 bytes)
- $\mathbf{z}_i$ [ ] = H · $\mathbf{x}_i$ (*m*= 128 bytes)
- *g*[ ] (*B*=464 bytes)
- H [ ] [ ] (*k***m*= 4*128= 512 bytes)
- Y'_i [ ] [ ] (*k***k*= 4*4= 16 bytes)
- *Y'* [ ] [ ] (*k***k*= 4*4= 16 bytes)
- Tabelle per le moltiplicazioni in GF(2^8) (512 bytes)

Sub-totale: (1776 bytes)

Più:

- *currentPartitions* [ ] (m*2= 256 bytes)
- *usedPartitions* [ ] (58 bytes)

Sub-totale: (314 bytes)

Per un totale di: 2090 bytes. Quindi l'ingombro ulteriore dovuto alla modifica di PIV è di soli 314 bytes, cioè soltanto il 18% di memoria in più, rispetto a PIV originale.

OSSERVAZIONE: Nel caso in cui si volesse includere nella *hash* anche la memoria flash di backup da 1024 KBytes, siccome a quel punto L= 58+1024 KBytes e B=L/m=8656 il totale diventerebbe:

- $\mathbf{x}_i$ [ ] (m= 128 bytes)
- $\mathbf{z}_i$ [ ] = H · $\mathbf{x}_i$ (m= 128 bytes)
- g[ ] (B=8656 bytes)
- H [ ] [ ] (k*m= 4*128= 512 bytes)
- Y'_i [ ] [ ] (k*k= 4*4= 16 bytes)
- Y' [ ] [ ] (k*k= 4*4= 16 bytes)
- Tabelle per le moltiplicazioni in GF(2^8) (512 bytes)

Sub-totale: (9968 bytes)

Più:

- *currentPartitions* [ ] (m*2= 256 bytes)
- *usedPartitions* [ ] (B/8=1082 bytes)

Sub-totale: (1338 bytes)

Per un totale di: 11306 bytes, dei quali *currentPartitions* e *usedPartitions* rappresentano soltanto il 12%. Ovviamente un sensore Tmote Sky non possiede così tanta memoria RAM, ma l'ingombro totale ridurrebbe a soli 2650 bytes se si

utilizzasse il generatore di numeri pseudocasuali per generare i valori di *g*[] in maniera dinamica.

5.3.8 Tempi di esecuzione dell'algoritmo PIVC e dell'algoritmo PIVm

Avendo realizzato un prototipo di PIVC (visto che non è stato possibile reperire il prototipo utilizzato in [3]) ed un prototipo di PIVm, sono state effettuate delle misurazioni dei tempi di esecuzione. Queste verranno illustrate nel paragrafo 9.3, ma vengono per comodità riportate anche qui, nella tabella 5.1.

	Tempo di esecuzione
PIVC	3,05s
PIVm	3,4s

Tabella 5.1: Tempi di esecuzione di PIVC e PIVCm

Dalla tabella dei tempi di esecuzione si evince che il rallentamento dovuto alla modifica che è stata effettuata ammonta a soltanto 0,35s, ovvero +11,5% in più rispetto al tempo di esecuzione di PIVC.

5.4 Conclusioni

E' possibile vedere che l'algoritmo PIV modificato riesce ad ottenere la copertura totale della memoria nonché la non parallelizzabilità del codice con un ridottissimo ingombro ulteriore (314 bytes più pochi altri bytes per il codice aggiuntivo) rispetto all'algoritmo PIV originale.

Come è stato già visto, la copertura totale della memoria si ottiene facendo in modo che il generatore di numeri pseudo-casuali selezioni le partizioni fisiche a gruppi di m alla volta, e dalle m partizioni selezionate vengono creati esattamente m blocchi di dati $\mathbf{x}_i$. Invece, la non parallelizzabilità si ottiene grazie all'attraversamento pseudo-casuale della memoria, a conseguenza del quale l'avversario non può conoscere a priori quali dati trasferire su un secondo sensore per poterlo utilizzare attivamente nel calcolo della *hash* e si ottiene anche grazie all'aver modificato il generatore di numeri pseudo-casuali in modo tale che le partizioni selezionate in un certo istante dipendano anche dai valori di *hash* calcolati fino a quell'istante. In questo modo anche se l'avversario copiasse gran parte dei dati sul sensore di supporto sperando che quest'ultimo possa calcolare qualcuna delle sotto-*hash* Y'_i, questo non sarebbe possibile perché il sensore di supporto non riuscirebbe a riprodurre la sequenza di numeri casuali genuina, proprio perché non avrebbe a disposizione i valori parziali della *hash*.

Inoltre, il protocollo PIV modificato può resistere anche agli attacchi con *hash* precalcolata, agli attacchi di *replay*, ed agli attacchi in cui l'avversario cerca di prevedere la prossima *challenge* che il server invierà. PIVm eredita queste proprietà da PIV, perché utilizza la stessa funzione di hash, cioè la RHF descritta nel par.5.1.1

Di conseguenza, è possibile affermare che il protocollo PIV modificato è in grado di resistere ad attacchi fisici di livello 1 e di livello 2 perpetrati da un avversario ai danni di uno o più sensori. In particolare, per ingannare il protocollo PIV modificato, l'avversario non potrebbe più fare affidamento su sensori che lavorino in parallelo ma dovrebbe ricorrere a modifiche hardware dei sensori. Questo perché solamente avendo una memoria RAM più capiente in cui conservare le porzioni di memoria che sono state modificate, il sensore potrebbe calcolare la *hash* corretta nei tempi previsti dal verificatore.

5.5 Ulteriori vantaggi

Vi sono altri tre grandi vantaggi che possono essere ottenuti, grazie a questa modifica:

5.5.1 Risparmio nelle comunicazioni

Il protocollo PIV originale prevede che il server, per iniziare il test di verifica di un nodo, gli invii un messaggi contenente il codice PIVC e le chiavi G $\in \mathcal{G}$ ed H $\in \mathcal{H}$ necessarie per calcolare la hash. Il protocollo prevede che il server selezioni casualmente i valori di queste due chiavi, prima di inviarle al sensore. Nel caso del PIV modificato, invece, dovendo comunque installare un generatore di numeri pseudo-casuali sul sensore, è possibile evitare di inviare le due chiavi G ed H all'interno del messaggio M2. Infatti, basterà inviare un unico *seme*, necessario ad inizializzare il generatore random, e poi sarà quest'ultimo a riempire i valori di G ed H.

Considerando che $\mathcal{G} = \mathcal{F}^B$ ed $\mathcal{H} = \mathcal{F}^{k \times m}$, se k=4 ed m=96 come proposto in [3] e considerando che su un sensore Tmote-Sky ci sarebbero un totale di B=464 blocchi $\mathbf{x}_i$ da analizzare, la dimensione complessiva di G ed H sarebbe dim(G + H) = dim(G) + dim(H) = 464 bytes + 384 bytes = 848 bytes. Quindi, installando un generatore di numeri pseudo-casuali sul nodo ci permette di risparmiare 848 bytes ogni volta che sia necessario verificare l'integrità di un nodo.

Inoltre, dopo aver stabilito che la procedura per il calcolo della *hash* è una procedura *nota* e dopo aver stabilito che, secondo le proprietà della RHF, è crittograficamente molto difficile che un sensore riesca a calcolare la *hash* corretta a partire da un'applicazione **x''** modificata, qualsiasi siano le chiavi G ed H e qualsiasi sia la procedura **Hash** che viene eseguita, risulta inutile inviare il codice PIVm insieme alla richiesta di iniziare un test di verifica dell'integrità. Quindi, la procedura **Hash**, verrà installata in maniera permanente su ogni nodo della rete, ed il server dovrà semplicemente attivarla con un messaggio indirizzato al sensore. Questo comporterebbe un ulteriore risparmio di 483 bytes nel pacchetto di richiesta di verifica.

Per calcolare il risparmio energetico dovuto a questa modifica, è necessario calcolare innanzitutto il consumo il consumo che si avrebbe se il server ed il client si scambiassero 1347 bytes come nel caso di PIV (1329 bytes per la *challenge* e 16

bytes per la *response*) e poi calcolare il consumo quando nel caso di PIVm, ovvero quando il server deve inviare soltanto 2 bytes nella *challenge* e quindi le comunicazioni ammontano ad un totale di 18 bytes (2 bytes per la *challenge* e 16 bytes per la *response*).

Il Tmote Sky utilizza un chip Chipcon CC2420 per le comunicazioni radio. Questo chip implementa le comunicazioni tramite il protocollo IEEE 802.15.4.
Analizziamo innanzitutto il caso di PIV. Attraverso i *datasheet* del Tmote Sky [19], il documento di definizione dello standard 802.15.4 [24] e i sorgenti di TinyOS è possibile scoprire che la massima dimensione del *payload* di un pacchetto è di 28 bytes, quindi per inviare 1347 bytes bisogna confezionare $\lceil 1329/28 \rceil = 49$ pacchetti. Ogni pacchetto ha, a livello di sistema operativo, un *header* di 10 bytes, mentre a livello MAC (*Media Access Control*) ha un *header* di 8 bytes. Quindi, per inviare i 1347 bytes di dati sarebbe necessario inviare ulteriori 18 bytes per pacchetto, di conseguenza sarebbe necessario inviare un totale di 49 · 18 bytes = 882 bytes di *overhead*. Inoltre, bisogna considerare un ulteriore vincolo, ovvero che dopo aver trasmesso un pacchetto il sensore deve attendere per un tempo pari almeno alla durata di trasmissione di 6 bytes, durante il quale rimane in pausa. Questi 6 bytes vengono chiamati space (o SIFS) e durante questo tempo il chip CC2420 rimane acceso, quindi il suo consumo è pari al consumo "di trasmissione". Per inviare 48 pacchetti, quindi, sarà necessario considerare ulteriori 6 bytes · 48 intervalli = 288 bytes. Aggiungendo questi ulteriori 6 bytes per pacchetto arriviamo ad un totale di 1347 bytes (dati) + 882 bytes (overhead) + 288 bytes (space) = 2517 bytes.

Per trasmettere un singolo byte il chip radio necessita di 32 μs, e quindi per inviare 2517 bytes dovrà rimanere acceso per 2517 · 32 μs = 0,08 s.

Considerando che l'assorbimento di corrente durante le trasmissioni è di 17,4 *mA* possiamo calcolare il consumo di carica dovuto alla trasmissione di 1347 bytes di dati, che sarà

$$Q_{TX\,[mAh]} = I_{TX\,[mA]} * \frac{t_{[s]}}{3600_{[s/h]}} = 17{,}4 * \frac{0{,}08}{3600} = 390 * 10^{-6}\,mAh\,. \tag{5.4}$$

Inoltre, considerando che durante la ricezione di dati l'assorbimento di corrente è di 19,7 mA, si ha un ulteriore consumo di

$$Q_{RX\,[mAh]} = I_{RX\,[mA]} * \frac{t_{[s]}}{3600_{[s/h]}} = 19,7 * \frac{0,08}{3600} = 438 * 10^{-6}\, mAh\,. \qquad (5.5)$$

In totale, quindi, il consumo energetico sarà di $Q_{TX[mAh]}$ + $Q_{RX[mAh]}$ = $828*10^{-6}$ *mAh* ogni volta che viene effettuata una verifica. Ovviamente questi consumi si riferiscono anche a tutti i nodi intermedi che in una rete *multi-hop* dovranno ricevere e rispedire i dati.

Nel caso di PIVm invece, i bytes di dati trasmessi sono soltanto 18, come è già stato visto. Questi dovranno comunque essere suddivisi in 2 pacchetti (trattandosi di *challenge* e *response*), ma non sarà necessario aggiungere i bytes di *space*. Quindi, per 2 pacchetti ci saranno 2 · 18 bytes = 36 bytes di overhead. In totale avremo che per ogni nodo intermedio bisognerà trasmettere 18 bytes + 36 bytes = 54 bytes. Il tempo necessario per queste trasmissioni sarà di 54 bytes · 32 µs/byte = 0,0017 s. Considerando l'assorbimento di corrente di 17,4 mAh durante la trasmissione, il consumo di carica elettrica per ogni verifica e per ogni nodo intermedio sarà di

$$Q_{TX\,[mAh]} = I_{TX\,[mA]} * \frac{t_{[s]}}{3600_{[s/h]}} = 17,4 * \frac{0,0017}{3600} = 8,4 * 10^{-6}\, mAh\,. \qquad (5.6)$$

In ricezione, invece, il consumo energetico sarà di

$$Q_{RX\,[mAh]} = I_{RX\,[mA]} * \frac{t_{[s]}}{3600_{[s/h]}} = 19,7 * \frac{0,0017}{3600} = 9,3 * 10^{-6}\, mAh\,. \qquad (5.7)$$

In totale, quindi, il consumo energetico sarà di $Q_{TX[mAh]}$ + $Q_{RX[mAh]}$ = $17,7*10^{-6}$ *mAh.*

E quindi è possibile vedere che la modifica introdotta da PIVm permette di risparmiare $810 \cdot 10^{-6}$ *mAh* su ogni nodo intermedio per ogni verifica, ovvero un risparmio del 97,8% sul consumo che il protocollo PIV comportava sull'intera rete.

5.5.2 Server di autenticazione AS

Ed infine, se la procedura **Hash** viene installata in modo permanente su ogni sensore, scompare il pericolo che un avversario, impersonificando il server PIVS, invii ad un sensore una procedura PIVc modificata in maniera tale da nuocere al sensore. Se questo pericolo non sussiste più, è plausibile ipotizzare anche l'eliminazione del server di autenticazione AS, che nel protocollo PIV originale era necessario per proteggere i sensori da server PIVS falsi.

5.5.3 Dati non comprimibili

PIV prevede che prima di iniziare il test un sensore deve riempire le parti di memoria inutilizzate con dei dati non comprmibili. La proposta che si può fare per migliorare l'efficienza (sia energetica che computazionale) del protocollo è che questi dati potrebbero essere scritti durante la fase di inizializzazione e configurazione dei nodi, almeno per quanto riguarda la memoria Flash. Infatti, scrivere sulla memoria flash è un'operazione molto costosa in termini energetici. E soprattutto, se la tensione di alimentazione del sensore scende sotto una certa soglia (2,7V nel caso dei sensori Tmote Sky), la scrittura su Flash NON è più possibile. Un protocollo di sicurezza che, per funzionare, necessiti di scrivere su Flash potrebbe essere facilmente ingannato se l'avversario utilizzasse delle batterie con bassa tensione in uscita o che riuscisse in qualche altro modo a far credere al sensore che le batterie sono quasi esauste. Questo si potrebbe ottenere anche, ad esempio, inserendo una resistenza in serie alle batterie, nel circuito di alimentazione.

5.6 Svantaggi

Uno svantaggio indotto dalla modifica che è stata proposta al protocollo PIV, è che il server PIVS non ha più la possibilità di calcolare la hash Y a partire dai *digest* $\mathbf{X}_1$, $\mathbf{X}_2$, … $\mathbf{X}_B$. Questi *digest* erano definiti come $\mathbf{X}_i = \mathbf{x}_i \cdot \mathbf{x}_i^T = (x_{i,h} \cdot x_{i,k})$. Dopo aver effettuato la modifica al protocollo PIV, siccome i blocchi $\mathbf{x}_i$ non corrispondono più a porzioni di memoria con indirizzi contigui, ma vengono bensì creati dinamicamente e sono diversi ad ogni test che viene effettuato, risulta impossibile definire dei *digest* come era previsto nel protocollo PIV. Ne consegue che il server PIVS dovrà possedere un'immagine **x** della memoria di ogni sensore della rete, e la procedura **Vrfy** dovrà effettuare le stesse operazioni che effettua la procedura **Hash**.

Questo non dovrebbe rappresentare un grosso problema però. Si ricorda che per come erano definiti i *digest*, questi richiedevano molto più spazio per essere immagazzinati, rispetto alle immagini **x**. Il motivo per cui venivano utilizzati i digest era che il calcolo della hash secondo l'equazione (5.2) risultava più semplice dal punto di vista della complessità computazionale, e quindi il PIVS poteva verificare più sensori contemporaneamente senza risultare sovraccaricato. Ciononostante, di fronte al guadagno ottenuto innalzando il livello di sicurezza ottenibile con PIV, questa perdita di efficienza sul "lato server" non è un prezzo eccessivo da pagare. Infatti, il server PIVS in ogni caso non ha nessun vincolo né riguardo al consumo energetico e può quindi disporre di una CPU all'avanguardia che permetta comunque di verificare le *hash* in tempi minimi.

6. Implementazione del client PIVCm

La procedure per il calcolo della *hash*, ovvero la RHF vista nel par 5.1.1, le procedure per effettuare l'*interleaving* dinamico con un generatore pseudo-casuale e la libreria matematica per effettuare le operazioni aritmetiche in GF(2^8) sono state realizzate utilizzando il sistema operativo TinyOS (versione 1.1.14), il linguaggio di programmazione nesC ed il linguaggio *assembly* per TI MSP430.

6.1 TinyOS

TinyOS [20] è un sistema operativo open source progettato appositamente per *sistemi embedded*, in particolare nell'ambito delle reti di sensori per una categoria di sensori chiamati *Berkeley Motes*. TinyOS consiste in un'architettura software composta da una serie di librerie che realizzano componenti di alto livello e da moduli (*wrappers*) che consentono di accedere all'hardware senza che il programmatore ne conosca i dettagli. Ad un livello più basso si trovano invece le parti che si occupano di gestire il *bootstrap* del dispositivo, la *schedulazione* delle operazioni, il *power management*, le interfacce di comunicazione, i sensori montati sulla scheda e le eventuali periferiche esterne.

6.1.1 Caratteristiche

Una caratteristica fondamentale di TinyOS è l'architettura *component-based*: il sistema fornisce un insieme di componenti, i quali vengono collegati fra loro dal programmatore attraverso l'operazione di *wiring*, indipendente dall'implementazione interna. La modularità del sistema consente il riuso del codice e la possibilità di escludere dall'applicazione i moduli inutilizzati risparmiando memoria, che nei sensori è presente in quantità particolarmente limitata.

Poiché le applicazioni per reti di sensori generalmente non prevedono lunghe elaborazioni di dati, ma piuttosto la risposta ad eventi quali la ricezione di un messaggio o il completamento di un'operazione di monitoraggio ambientale, TinyOS sfrutta il modello di concorrenza *event-driven*. L'esecuzione del codice è suddivisa in *eventi* e *task*.

Un task è una parte di codice che viene eseguita in maniera differita, cioè quando non sono in esecuzione altri task o eventi. Un modulo software può richiedere l'esecuzione di un task attraverso l'operazione di *posting*, che inserisce il task in una coda con politica *FIFO* (first-in-first-out): lo *scheduler* avvia l'esecuzione dei task nello stesso ordine con cui sono stati inseriti in coda. I task hanno una priorità di esecuzione bassa, cioè durante la loro esecuzione possono essere interrotti dal verificarsi di un evento, mentre invece non si possono interrompere tra loro.

Gli eventi invece sono porzioni di codice che possono interrompere (*preempt*) l'esecuzione di task o di altri eventi. Essi sono generalmente la conseguenza di un'interruzione hardware, generata in seguito alla ricezione di un messaggio, all'acquisizione della misurazione di un sensore o al passare del tempo scandito da un timer. Un evento può essere sollevato anche per indicare all'applicazione che un'operazione richiesta è stata portata a termine.

L'interazione fra i componenti software di TinyOS è specificata attraverso le *interfacce*, le quali descrivono quali sono i servizi che un certo modulo mette a disposizione e quali sono invece quelli richiesti. Un'interfaccia è composta da un certo insieme di comandi ed eventi: un *comando* è generalmente una richiesta di iniziare una determinata operazione (generalmente verso un componente a livello più basso), mentre un *evento* serve per indicare che la richiesta è stata completata oppure può essere scatenato direttamente dall'hardware, come visto in precedenza.

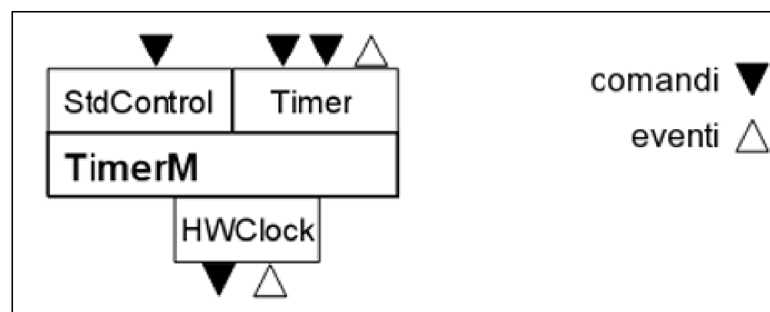


Fig6.1 L'interfacciamento del modulo **TimerM**.

Come esempio, in figura 6.1 è mostrato l'interfacciamento del modulo **TimerM**.

Il fatto che il modello di concorrenza preveda che i task non possano essere interrotti implica l'assenza in TinyOS di operazioni bloccanti: tutte le azioni che richiedono un certo periodo di tempo per essere eseguite sono di tipo *split-phase*. Questo significa che il comando di richiesta del servizio restituisce immediatamente

il controllo al modulo chiamante, che prosegue la propria esecuzione. Al termine dell'operazione viene generato un evento di completamento. Ad esempio quando si vuole inviare un messaggio, con il comando **send** si informa il componente di comunicazione della volontà di trasmettere un pacchetto, mentre quando l'operazione è terminata viene scatenato l'evento **sendDone**.

6.1.2 Comunicazione in TinyOS

La comunicazione in TinyOS si basa fortemente sugli *Active Messages* [21] o messaggi attivi. Un messaggio attivo è un messaggio che include al suo interno un *identificatore* che indica l'operazione che deve essere eseguita in seguito alla ricezione. Quindi TinyOS inserisce nel pacchetto da inviare, oltre all'indirizzo del nodo destinatario e ai dati, un campo che ne determina il tipo. Questo valore non fa altro che indicare al sistema operativo del ricevitore quale deve essere la funzione (*handler*) destinata ad elaborare il *payload* del messaggio stesso.

Tutte le funzionalità di comunicazione *single-hop*, quali l'invio di pacchetti e la generazione degli eventi di ricezione, sono fornite dal sistema operativo attraverso il componente **GenericComm**. L'interazione fra i componenti principali coinvolti è rappresentata in figura 6.2

Per limitare al massimo l'utilizzo di memoria, in **GenericComm** non è implementato nessun tipo di *buffering* per i pacchetti, ma si utilizza lo spazio allocato dall'applicazione. Tale spazio diventa di proprietà di **GenericComm** durante le operazioni di invio e viene restituito al momento dell'invocazione dell'evento di completamento. Una situazione simile avviene anche durante la ricezione: una volta ricevuto un pacchetto l'applicazione deve restituire allo *stack di comunicazione* il buffer utilizzato, oppure può mantenerne il possesso purché gliene fornisca un altro (operazione di *buffer swapping*). Se si rende necessaria la memorizzazione di più pacchetti, l'applicazione deve provvedervi autonomamente.

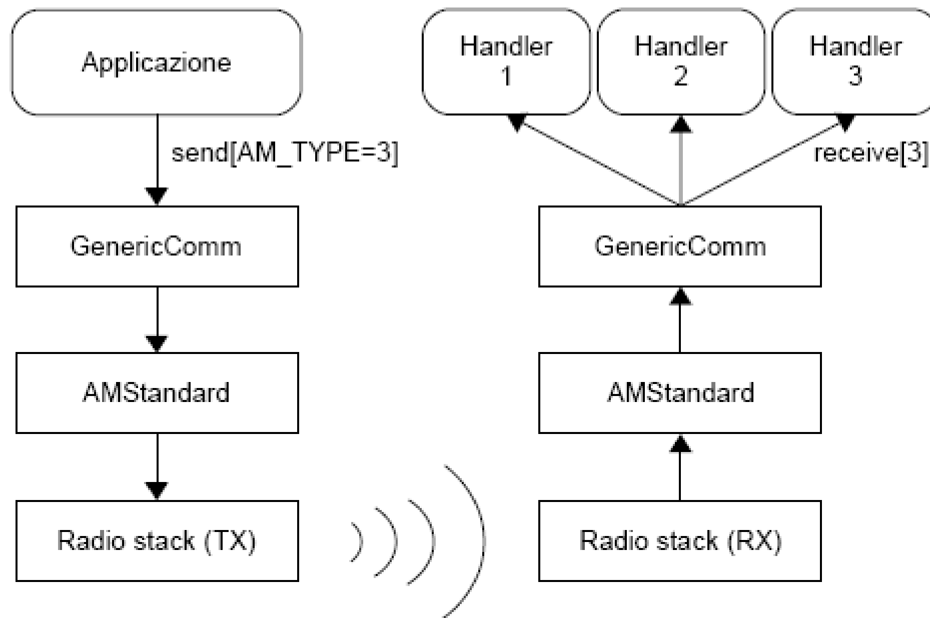


Figura 6.2: Comunicazione in TinyOS

Il paradigma di comunicazione utilizzato in TinyOS è quello *best effort*: la consegna dei messaggi non è garantita, non essendo implementato nessun protocollo di trasporto né alcun meccanismo di ritrasmissione dei pacchetti (anche se esiste la possibilità di abilitare l'invio di messaggi di *acknowledgment*[5]). Per quanto riguarda il protocollo di accesso al mezzo condiviso, si utilizza il CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance), sebbene esistano altri protocolli implementati da terze parti.

[5] I messaggi di acknowledgment notificano il mittente della ricezione di un pacchetto

6.2 Programmazione in nesC

Le applicazioni basate su TinyOS, così come il sistema operativo stesso, sono scritte utilizzando il linguaggio nesC [22] e [23]. Il nesC è un estensione del linguaggio C, ed è molto utilizzato per programmare i sistemi embedded dato che fornisce tutti i servizi necessari all'accesso all'hardware ed è molto conosciuto dagli sviluppatori. In particolare il nesC permette di ottimizzare il codice tramite l'*analisi statica* e di rilevare eventuali problemi di accesso concorrente alle variabili condivise. Questi risultati sono ottenibili grazie alla staticità imposta al sistema: in nesC non esiste l'allocazione dinamica della memoria (le variabili vengono memorizzate nello stack oppure allocate staticamente nello spazio di memoria globale) e non si usano i puntatori a funzioni, per cui il flusso di esecuzione è interamente noto a tempo di compilazione.

6.2.1 Creazione dei componenti

Le applicazioni in nesC sono costruite scrivendo e collegando fra loro componenti, ognuno dei quali fornisce e usa servizi come specificato dalle interfacce; questo approccio rispecchia l'architettura component-based di TinyOS.

Esistono due tipi di componenti in nesC: i moduli, che implementano i comandi e gli eventi definiti dalle interfacce, e le configurazioni, che servono per collegare fra loro altri componenti, connettendo le interfacce fornite con le omologhe interfacce usate. Un esempio di come si crea una configurazione è rappresentato in figura 6.3

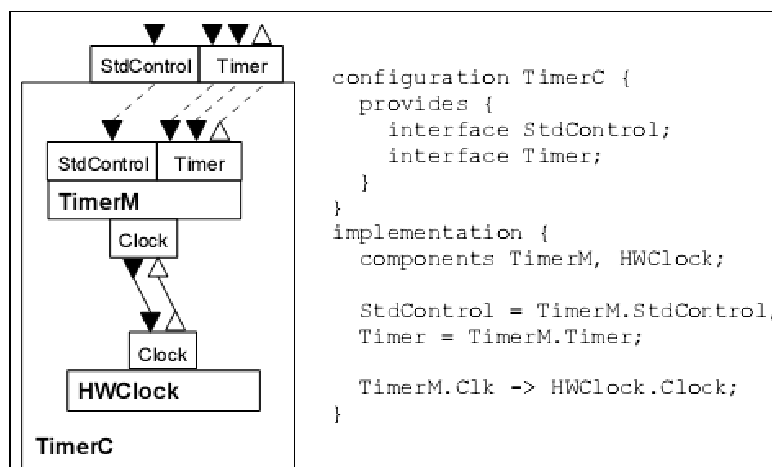


Figura 6.3: La configurazione **TimerC**

La struttura di un'applicazione viene descritta complessivamente da una *top-level configuration*, che definisce il *wiring* di tutti i componenti utilizzati e deve necessariamente contenere il componente **Main**, che provvede al *boot* del dispositivo. Il corpo dei moduli è scritto utilizzando la sintassi del linguaggio C, con le opportune estensioni utili per rappresentare i comandi e gli eventi. Tali funzioni speciali vengono definite semplicemente facendo precedere la loro definizione dalla parola chiave **command** o **event**. Per quanto riguarda l'invocazione di un comando, questa viene effettuata anteponendo la parola chiave **call** alla chiamata di funzione, mentre gli eventi vengono sollevati attraverso la parola chiave **signal**.

6.2.2 Concorrenza in nesC

Il modello di concorrenza di TinyOS, con i suoi task ed eventi, comporta la possibilità di incorrere in *race conditions*[6] nel momento in cui flussi di esecuzione a priorità diversa accedono a una variabile condivisa. Per facilitare il rilevamento di queste condizioni critiche, il codice viene logicamente suddiviso in:

- codice sincrono: codice raggiungibile solo da task;
- codice asincrono: codice raggiungibile da almeno un interrupt handler.

Il codice sincrono risulta essere atomico[7] rispetto ad altro codice sincrono, in quanto i task si susseguono senza interrompersi tra loro, per cui in questo caso non ci sono problemi ad accedere a risorse condivise. Il problema si può presentare invece quando le variabili vengono utilizzate da codice asincrono, visto che gli eventi possono interrompere altri eventi o task. In questo caso il programmatore ha due alternative: fare in modo che tutto il codice in questione sia sincrono, oppure rendere atomico l'accesso alle risorse. In caso contrario il compilatore risponde con un errore. L'accesso atomico si realizza in nesC attraverso un blocco di codice dichiarato con la parola chiave **atomic**, che comporta la disabilitazione degli interrupt durante l'esecuzione del blocco stesso.

[6] Corse critiche, ovvero situazioni in cui parti di codice diverse operano su una risorsa comune ed in cui il risultato dipende dall'ordine in cui le operazioni sono eseguite

[7] Non suddivisibile in azioni più semplici, non interrompibile

6.3 Applicazione

Durante lo sviluppo si è cercato di mantenere una struttura quanto più modulare possibile per garantire il riuso del codice. L'applicazione ha la struttura rappresentata nella figura 6.4

.

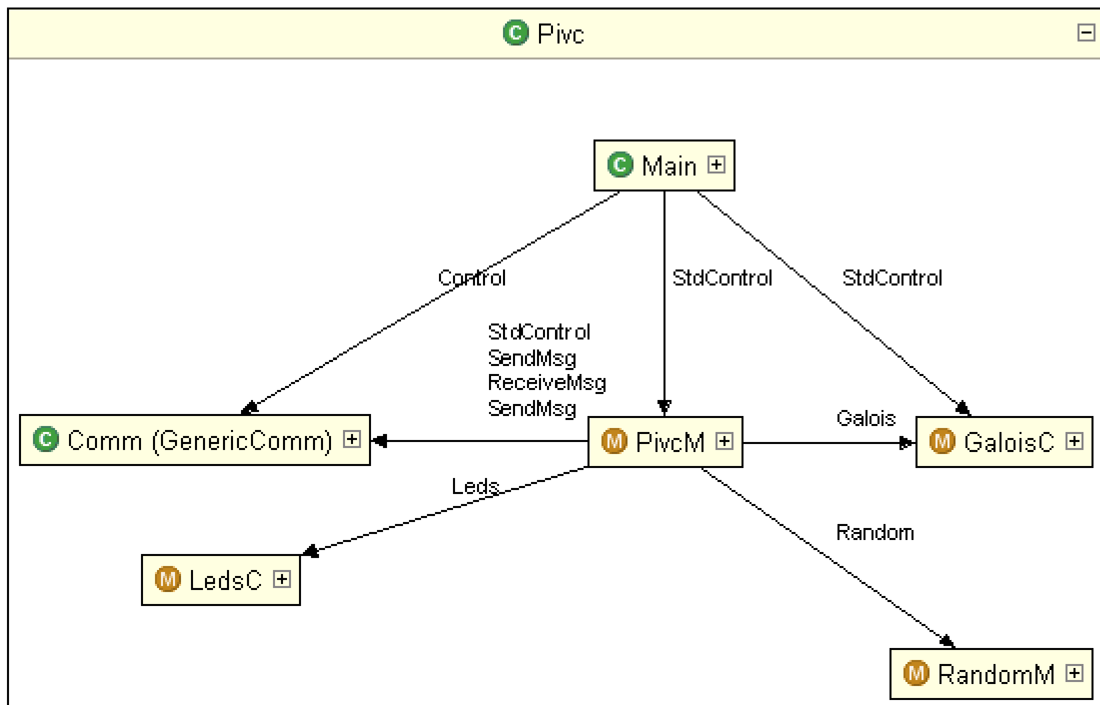


Figura 6.4: Interconnessioni tra i moduli dell'applicazione **Pivc**

In figura è mostrato il *wiring*, ovvero le interconnessioni tra i moduli dell'applicazione **Pivc**. E' possibile vedere che l'applicazione è composta di 6 moduli. I moduli **Main**, **GenericComm** e **LedsC** fanno parte del sistema operativo, il modulo **RandomM** è stato prelevato dalle librerie di TinyOS 2.0 , mentre i moduli **GaloisC** e **PivcM** sono stati realizzati.

Nella figura sono mostrate anche le interfacce attraverso cui questi moduli comunicano. Le frecce indicano i collegamenti tramite interfaccia: alle estremità con la punta c'è il modulo che fornisce l'interfaccia mentre all'altra estremità c'è il modulo che ne fa uso.

L'interfaccia **StdControl** è un interfaccia standard che deve essere offerta da ogni modulo che ha bisogno di essere inizializzato. All'avvio dell'applicazione,

infatti, il modulo **Main** effettua una chiamata a **StdControl.init()** e successivamente a **StdControl.start()**. Queste chiamate si trasmettono automaticamente a tutti i moduli che sono collegati al modulo **Main** attraverso l'interfaccia **StdControl**.

In particolare, il modulo **GaloisC** fornisce l'interfaccia **StdControl** perché in questo modo, all'avvio, esso potrà inizializzare le due tabelle di *lookup* per le moltiplicazioni in $GF(2^8)$.

Il modulo **PivcM** è il modulo principale dell'applicazione ed è il modulo che gestisce le comunicazioni con il server PIVSm e che effettua il calcolo della *hash* secondo l'algoritmo previsto dal protocollo PIVm (PIV modificato) visto nella sezione 5.3 del capitolo 5.

6.3.1 Il modulo GaloisC

Il modulo **GaloisC** implementa una libreria matematica di operazioni aritmetiche in un campo chiuso di Galois-Rijndael di 256 elementi. Questo modulo fornisce due interfacce: l'interfaccia standard **StdControl**, necessaria per inizializzare le tabelle di lookup e l'interfaccia **Galois**, attraverso la quale è possibile effettuare le suddette operazioni. Le due tabelle di *lookup* permettono di velocizzare le operazioni di moltiplicazione e divisione.

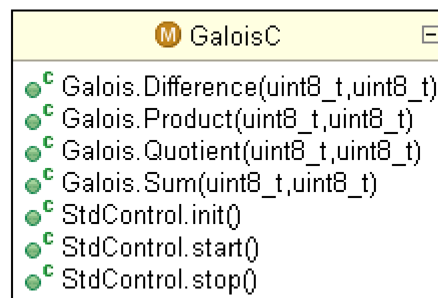


Figura 6.5: l'interfaccia offerta dal modulo **GaloisC**

Nella figura 6.5 è mostrata l'interfaccia offerta dal modulo **GaloisC**. Questa mette a disposizione quattro comandi, che effettuano rispettivamente la sottrazione, la moltiplicazione, la divisione e la somma, effettuate in $GF(2^8)$. I quattro comandi

dell'interfaccia prendono ognuno due parametri di tipo **uint8_t** e restituiscono un parametro **uint8_t**.

Questa interfaccia non prevede architettura *split-phase*, ovvero, essendo le operazioni da effettuare molto semplici, non è necessario che il modulo GaloisC effettui una richiamata (con una funzione di tipo *event*) dopo aver effettuato le operazioni. Bensì, i calcoli vengono effettuati direttamente all'interno delle funzioni *command*, e queste restituiscono direttamente i risultati.

Le implementazioni di queste operazioni sono particolarmente semplici, e sono state scelte proprio perché sono molto efficienti dal punto di vista della complessità computazionale. In particolare, in GF(2^8) la somma e la sottrazione sono uguali e consistono in uno **XOR** bit a bit tra i due operandi, mentre il prodotto e la divisione vengono effettuate con una trasformazione nel campo dei logaritmi ed una successiva antitrasformazione, secondo le relazioni

$$a \cdot b = ln^{-1}(ln(a)+ln(b)) \tag{6.1}$$

$$a/b = ln^{-1}(ln(a)+ln(b)) \tag{6.2}$$

Il modulo, oltre allo spazio che occuperà in memoria programma, avrà bisogno di 512 bytes di memoria RAM nei quali conservare le due tabelle di *lookup* necessarie ad effettuare le operazioni di logaritmo naturale (*ln*) ed antilogaritmo naturale (ln^{-1}).

E' in questo modulo che è stato utilizzato il linguaggio *assembly* per il microprocessore MSP430. Durante il calcolo della *hash*, il comando **Product(uint8_t,uint8_t)** viene chiamato più di 200'000 volte, rendendolo quindi il punto più critico dal punto di vista della complessità computazionale dell'intera applicazione PIVCm.

Quindi vi è stata fatta una piccola modifica, sostituendo un'operazione di "resto di divisione intera" con un operazione migliore, implementata in *assembly*. Questa modifica ha permesso di risparmiare più di 8 secondi sulla durata totale del calcolo della *hash*..

6.3.2 Il modulo RandomM

Il modulo **RandomM**, fornito con il sistema operativo TinyOS (ver. 2.0) implementa un generatore di numeri pseudo-casuali di tipo lineare congruente. La sua complessità computazionale è simile ad algoritmi come ad es. RC4, ed è di facile implementazione.

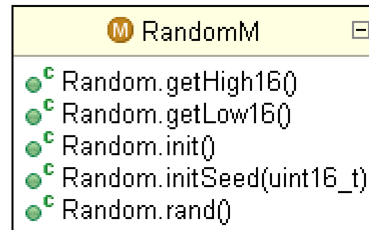


Figura 6.6: l'interfaccia offerta dal modulo **RandomM**

Nella figura 6.6 è mostrata l'interfaccia offerta dal modulo **RandomM**. Essa mette a disposizione cinque comandi. Questi sono:

- **Random.init()**, inizializza l'algoritmo usando come seme l'identificativo di rete che è stato assegnato al sensore al momento dell'installazione. Questo identificativo è accessibile tramite la costante **TOS_LOCAL_ADDRESS**.
- **Random.initSeed(uint16_t)**, che prende come parametro un valore **uint16_t** che usa come seme per inizializzare l'algoritmo.
- **Random.rand()**, che restituisce un parametro di tipo **uint32_t**. Questa funzione genera un numero intero a 32 bit e lo restituisce alla funzione chiamante.
- **Random.getHigh16()**, restituisce un parametro di tipo **uint16_t**, ottenuto prendendo i 16 bit più alti dell'ultimo numero generato con **Random.rand()**.
- **Random.getLow16()**, restituisce un parametro di tipo **uint16_t**, ottenuto prendendo i 16 bit più bassi dell'ultimo numero generato con **Random.rand()**.

NOTA: Random.getHigh16() e **Random.getLow16()** non generano un nuovo numero, ma restituiscono semplicemente la parte alta o la parte bassa del numero che è stato generato con l'ultima chiamata a **Random.rand()**.

6.3.3 Il modulo PivcM

Il modulo **PivcM** è il modulo che implementa la funzione per il calcolo della *hash* secondo il protocollo PIVC modificato, così come è stata descritta nel capitolo 5. Questo modulo, a livello più alto può essere vista come un automa a stati finiti.

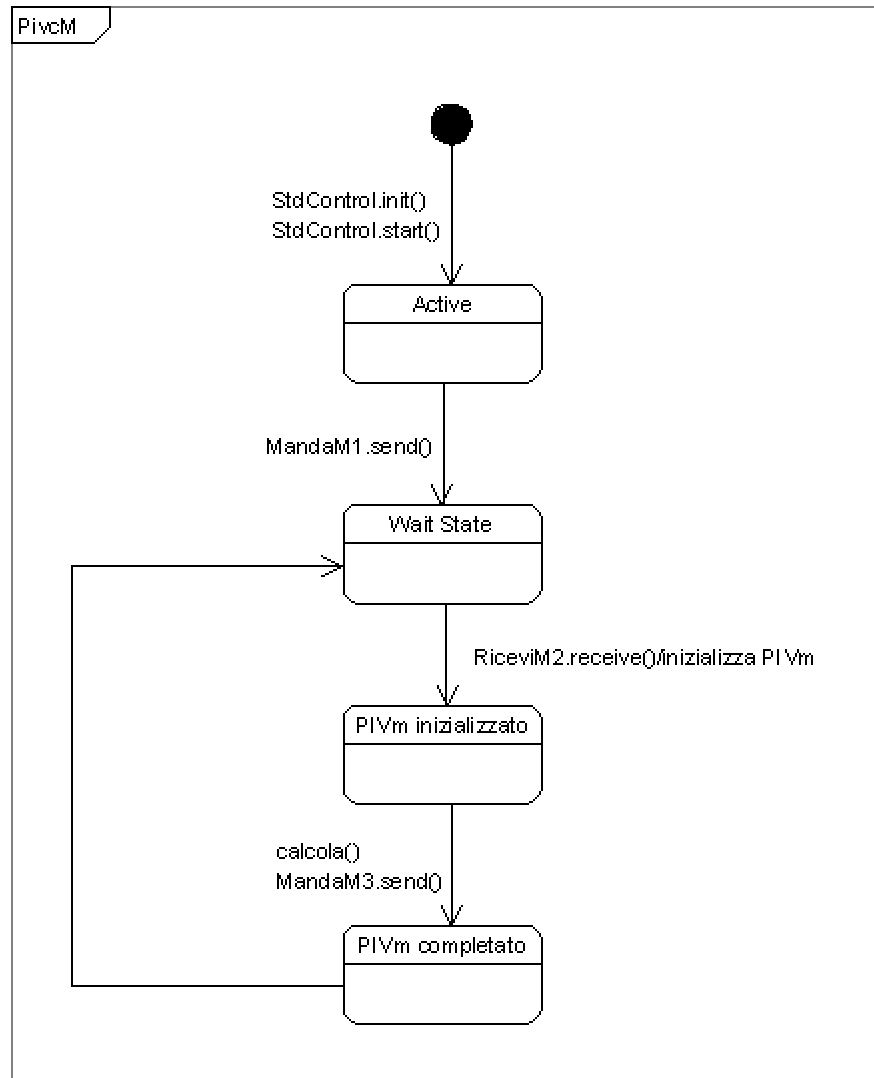


Figura 6.7: stati del modulo **PivcM**

In figura 6.7 sono mostrati gli stati interni del modulo **PivcM**. Il sensore, al momento dell'accensione invia al server una richiesta di ingresso nella rete. A quel punto il server risponde con una *challenge*, ovvero con un messaggio contenente i dati necessari al sensore per effettuare il test di verifica dell'integrità. Dopo aver ricevuto i dati, il sensore può effettuare il calcolo e spedire la *hash* al server all'interno di un messaggio di tipo M3. Infine il sensore si porta nuovamente nello stato *Wait State*. Al

modulo **PivcM** non arriverà alcuna risposta dal server, perché il server, una volta presa una decisione riguardo a quel sensore, provvederà eventualmente a trasferire la sua decisione ad altri protocolli di sicurezza che consentiranno l'accesso alla rete oppure negheranno l'accesso alla rete al sensore. Rimanendo nello stato *Wait State* il sensore è pronto a ricevere eventuali altre richieste di verifica dell'integrità che possano arrivare dal server. Questo potrà accadere periodicamente oppure se un servizio di IDS segnala il sensore come 'sensore potenzialmente manomesso' (come è stato visto nel paragrafo 4.1.20).

E' possibile vedere che, all'accensione del nodo, il modulo **PivcM** viene inizializzato attraverso il comando **StdControl.init()**, secondo le regole standard di TinyOS.

All'interno del comando **StdControl.init()** vengono chiamati i comandi **StdControl.init()** dei moduli **LedsC** e **GenericComm**. Nel comando **StdControl.start()** del modulo, invece, viene *postato* il task **mandaM1()**, in modo da iniziare lo scambio di messaggi con il server.

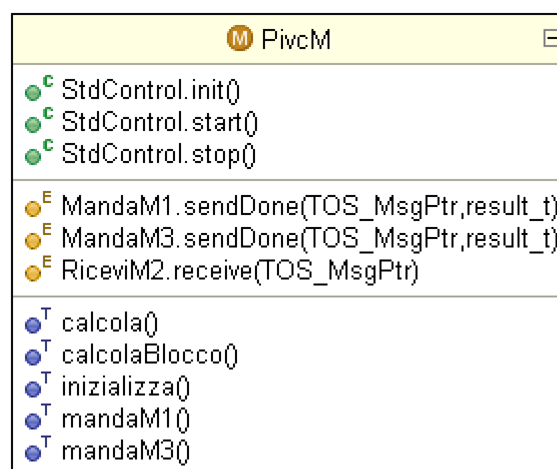


Figura 6.8: interfacce offerte dal modulo **PivcM**

In figura 6.8 sono mostrate le interfacce offerte dal modulo **PivcM**. In questa figura è possibile vedere anche i task di cui è composto il modulo.

Trattandosi di un'applicazione per un sistema con pochissima memoria RAM e bassa potenza di calcolo molte variabili sono state dichiarate *globali*, per evitare che nelle chiamate a funzione si dovesse ricorrere ad operazioni di copia. Nel linguaggio nesC

le variabili globali hanno visibilità e scopo ristretti al modulo in cui sono state dichiarate, e quindi se due moduli dichiarano due variabili con lo stesso nome, non vi sarà alcun conflitto.

Avendo precedentemente definito le seguenti costanti, necessarie per adattare il protocollo PIVm al sensore Tmote Sky:

- **k** = 4
- **m** = 128
- **B**=384
- **dim_vettore** = 48

dove **k** è stato scelto pari a 4 perché secondo il dimensionamento fatto in [3] risulta essere la soluzione più ragionevole tra sicurezza e complessità computazionale, **m** è stato scelto pari a 128 perché, non comportando più problemi di *overhead* nelle comunicazioni tra PIVSm e sensore (ricordiamo che nel protocollo PIV originale il server PIVSm doveva inviare un pacchetto di richiesta contenente la matrice H di dimensione $k \cdot m$, mentre nel protocollo PIVm ciò non è più vero) risulta una soluzione molto più comoda. Infatti, mentre la dimensione della memoria è perfettamente divisibile in partizioni da 128 bytes, ciò non è vero quando le partizioni sono di 96 bytes.

B è stato scelto pari a 386 secondo la relazione ($B=\Lambda/m$) ed infine **dim_vettore**, che rappresenta la dimensione del vettore di bit è pari a 48, ovvero **B**/8, perché per ogni partizione viene riservato un bit nel suddetto vettore di bit.

Il modulo PivcM fa uso delle seguenti variabili globali:

- **uint8_t x[m]**
- **uint8_t z[k]**
- **uint8_t Yi[k][k]**
- **uint8_t Y[k][k]**
- **uint8_t H[k][m]**
- **uint8_t g[B]**
- **uint16_t seme**

- **uint16_t partizioni_attuali[m];**
- **uint8_t vettore[dim_vettore];**

- **int macrobloc**
- **int bloc**

Il primo gruppo di variabili corrisponde esattamente ai vettori ed alle matrici viste nella descrizione di PIVC nella sezione 5.1 del capitolo 5. Il vettore **partizioni_attuali** è un vettore di interi in cui l'algoritmo conserva gli indici delle ultime **m** partizioni selezionate, il vettore **vettore** è il vettore di bit (inizializzato a zero) in cui vengono marcati i bit relativi alle partizioni che sono già state selezionate, ed infine **macrobloc** e **bloc** sono due contatori che vengono utilizzati dall'algoritmo. Il contatore **macrobloc** viene incrementato ogni volta che vengono selezionate **m** partizioni da analizzare, e quando raggiunge il valore di **B**/**m** indica che sono state selezionate tutte le partizioni (questo controllo sulla variabile **macrobloc** verrà effettuato all'interno del task calcola(), come è possibile vedere nella figura 6.10). Il contatore **bloc** viene azzerato ogni volta che vengono selezionate **m** partizioni nuove e viene incrementato ogni volta che viene creato un blocco a partire dalle partizioni attuali. Quindi, quando **bloc** raggiunge il valore **m**, sono stati creati **m** blocchi dalle partizioni attuali e quindi è giunto il momento di selezionare altre **m** partizioni, se ve ne sono ancora disponibili (il controllo sulla variabile **bloc** verrà effettuato nel task **calcolaBlocco()**, come è possibile vedere nella figura 6.11).

Analizziamo brevemente cosa accade nel sensore dopo aver mandato il messaggio M1, in cui esso chiede al server di essere sottoposto al test di verifica dell'integrità. In figura 6.9 è possibile vedere tutte le funzioni, i comandi, gli eventi e task di cui è composto il modulo PivcM. Dopo aver chiesto l'ingresso nella rete, il nodo rimane in attesa del messaggio M2 dal server, nel quale il server gli comunicherà il seme con cui inizializzare il generatore (questo seme rappresenta l'informazione *fresca* di cui si è parlato nei capitoli 3,4 e 5). Una volta ricevute queste informazioni, il nodo può (grazie al *task* **inizializza()**) inizializzare il generatore di numeri pseudo-casuali, la matrice **H[ ][ ]** ed il vettore **g[ ]**. Questi ultimi vengono riempiti con valori prodotti dal generatore. Invece, la matrice **Y[ ][ ]** e il vettore di bit **vettore[ ]** vengono semplicemente inizializzati a 0.

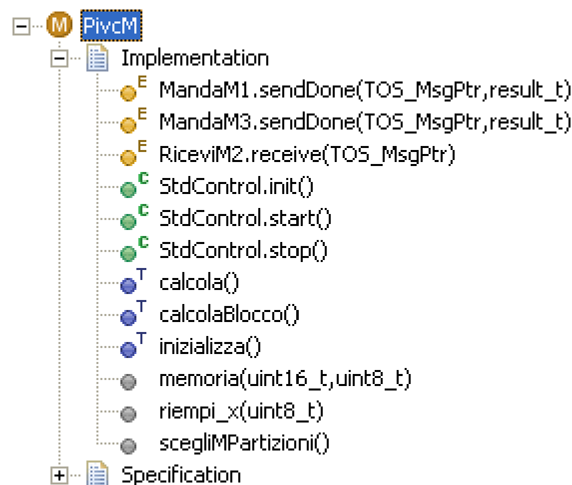


Figura 6.9: Implementazione del modulo **PivcM**

6.3.4 I task del modulo PivcM

Dopo la fase di inizializzazione, dall'interno del task **inizializza()**, viene postato il task **calcola()** che gestisce la creazione dei blocchi di dati e le chiamate al task **calcolaBlocco()**, che gestisce la creazione dei blocchi ed il calcolo delle sotto-*hash* Y'_l sui blocchi così ottenuti.

Durante la fase di implementazione si è resa necessaria una scelta abbastanza singolare. Il calcolo della *hash* è stato suddiviso in due *task* che si richiamano a vicenda secondo uno schema che viene illustrato nelle figure 6.10 e 6.11.

Questa scelta è stata necessaria perché in TinyOS, per garantire il corretto funzionamento del sistema operativo è importante non impegnare mai il processore nell'esecuzione di un unico *task* per più di 2-3 secondi. Infatti, non essendoci *preemption* ed avendo un unico livello di priorità per l'esecuzione dei *task*, i *task* di sistema vengono inseriti nella stessa coda in cui vengono inseriti i *task* utente. Tra i *task* di sistema troviamo i gestori del chip radio, delle interfacce di conversione A/D, gli *handler* dei timer nonché molte altre operazioni. Quindi è stato necessario evitare che il calcolo della *hash* venisse eseguito all'interno di un unico task, perché questo avrebbe impegnato il processore per una decina di secondi. Il calcolo è stato suddiviso su 2 *task*: il *task* **calcola()** si occupa della selezione delle partizioni da utilizzare per effettuare l'*interleaving*, mentre il task **calcolaBlocco()** prepara un blocco di dati a partire dall'ultimo gruppo di partizioni selezionate, calcola la sotto-*hash* relativa a quel blocco e infine posta sé stesso nella coda dei *task*. Questa

operazione è necessaria perché, in questo modo, il task termina la propria esecuzione e permette ad eventuali task di sistema di essere eseguiti prima che il task **calcolaBlocco()** venga rischedulato e riprenda con il calcolo della sotto-hash del blocco successivo.

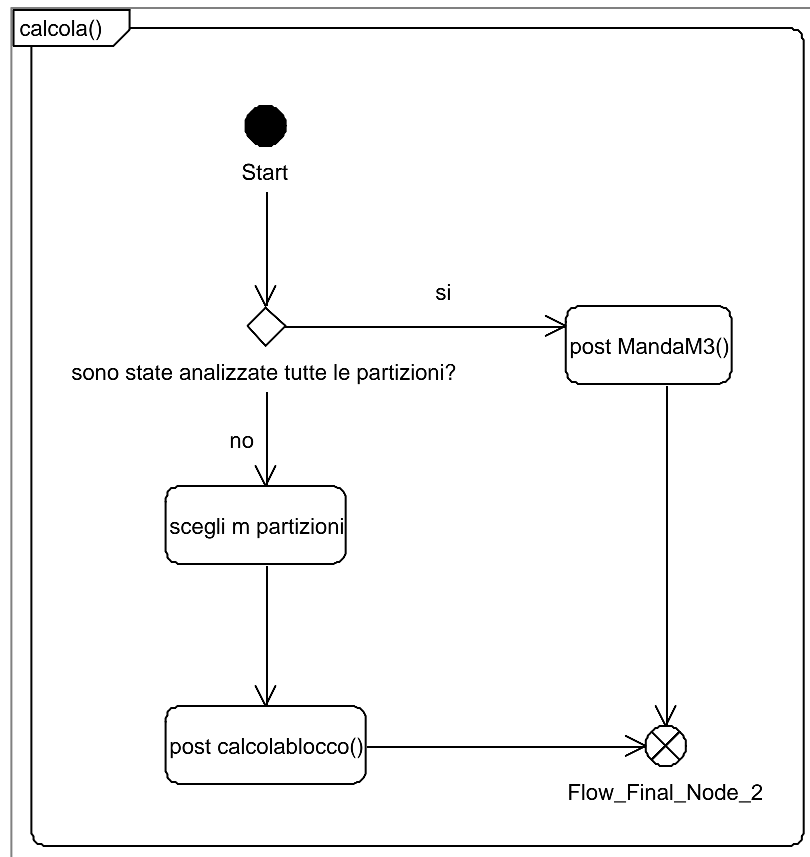


Figura 6.10: diagramma delle attività del task **calcola()**

6.3.5 Le funzioni di supporto del modulo PivcM

scegliMPartizioni():

La scelta delle **m** partizioni con cui comporre i prossimi **m** blocchi di dati viene effettuata dalla funzione **scegliMPartizioni()**. Ogni volta che viene invocata, essa azzera il contatore **bloc** e successivamente, utilizzando il generatore di numeri pseudo-casuali e il vettore di bit in cui sono state marcate le partizioni già utilizzate,

seleziona **m** partizioni che non sono ancora state usate, inserisce i loro indici nel vettore **partizioni_attuali[]** e le marca come 'gia usate' nel vettore di bit **vettore[ ]**.

Grazie a questa funzione, il task **calcolaBlocco()** può creare blocchi di memoria sempre diversi, a seconda della sequenza di numeri casuali prodotti dal generatore implementato nel modulo **RandomM**.

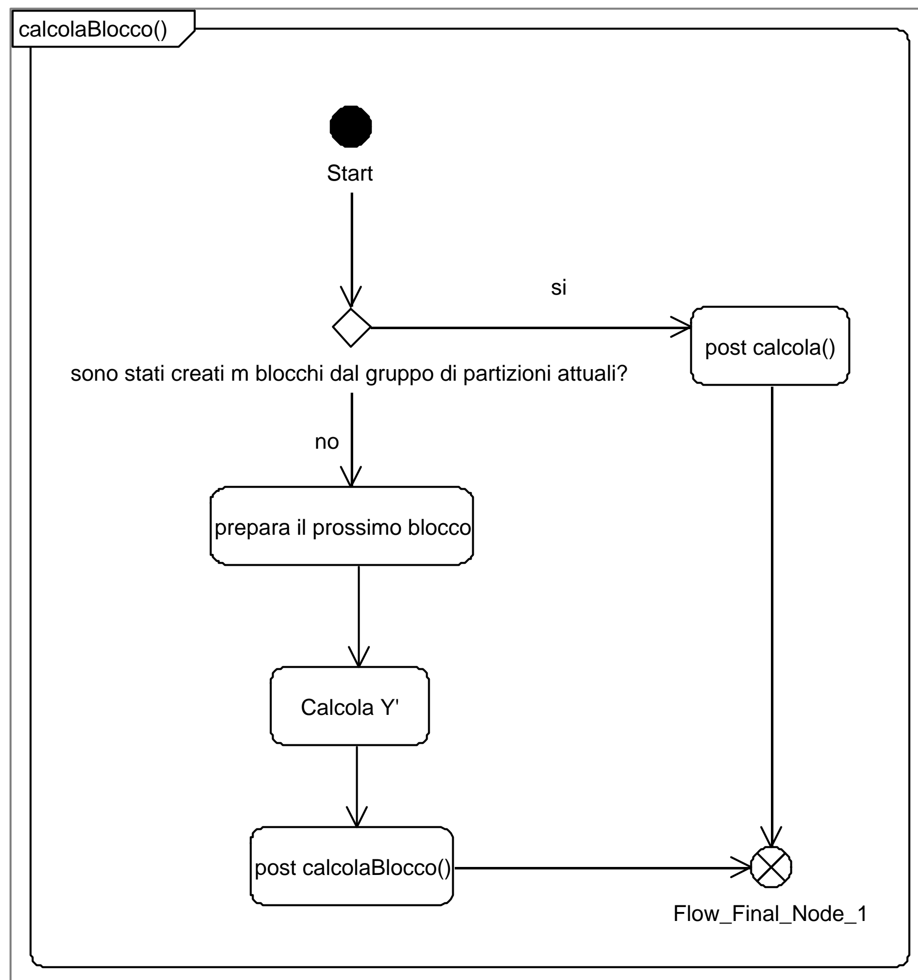


Figura 6.11: diagramma delle attività del task **calcolaBlocco()**

riempi_x (uint8_t):

La funzione **riempi_x (uint8_t)** viene invocata dal task **calcolaBlocco()**, ed è la funzione che si occupa di preparare il prossimo blocco di dati sul quale, poi, il task **calcolaBlocco()** calcolerà la prossima sotto-*hash*. Questa funzione prende un parametro di tipo **uint8_t**, ed effettua la seguente operazione, che corrisponde in pratica all'operazione di *interleaving* vista nella sezione 5.3 del capitolo 5

Funzione riempi_x (uint8_t)

```
int   partizioni_attuali[m]
int   shift (0 < shift < m)
```

```
result_t riempi_x(uint8_t shift) {
      int i;
      for(i=0;i<m;i++) {
            x[i]= memoria(partizioni_attuali[(i+shift)%m],i);
      }
      return SUCCESS;
}
```

Siccome ogni gruppo di **m** partizioni viene utilizzato per creare **m** blocchi di dati, il parametro **shift** indica quale degli **m** blocchi bisogna creare, dando modo di selezionare tutti i bytes da tutte le partizioni, al variare di questo parametro. La funzione inizia prelevando il primo byte della partizione il cui indice è contenuto in **partizioni_attuali[shift]** e copiandolo in **x[0]**. I bytes successivi di **x** vengono prelevati dalle partizioni seguenti, nello stesso ordine in cui esse compaiono nel vettore **partizioni_attuali[ ]**.

memoria (uint16_t, uint8_t):

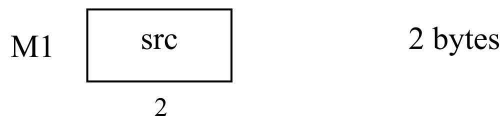
La funzione **memoria (uint16_t, uint8_t)**, quando chiamata con 2 parametri **i** e **j**, permette di prelevare il byte **j**-esimo della partizione **i**-esima. Ovviamente, in questo caso "partizione **i**-esima" si riferisce alla partizione fisica di indice **i**, e non, come nel caso della funzione **riempi_x (uint8_t)** alla partizione il cui indice è contenuto in **partizioni_attuali[i]**.

NOTA: Le partizioni fisiche descritte nella sezione 5.3 non sono necessariamente contigue. Ovvero, su alcune architetture, soprattutto quando lo spazio di indirizzamento è composto da più memorie e quando sono presenti delle memorie esterne non direttamente accessibili dal processore, le partizioni fisiche potrebbero non essere contigue e la funzione **memoria(int, int)** avrebbe anche il compito di pilotare i *controller* di queste memorie esterne. Se il modulo PivcM dovesse essere portato su un'altra piattaforma, oltre a cambiare le costanti globali **B** e **dim_vettore**, sarebbe assolutamente necessario modificare anche questa funzione.

6.4 Formato dei pacchetti

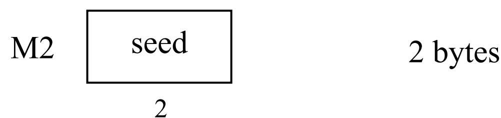
Messaggio M1 (richiesta d’ingresso nella rete):

Al momento dell’accensione, dopo aver inizializzato tutti i moduli e i componenti dell’applicazione, il sensore invia un pacchetto al server PIVSm contente il proprio ID di rete.



Messaggio M2 (*challenge*, ovvero richiesta di esecuzione della procedura di hash)

Dopo aver ricevuto una richiesta da parte di un sensore, il server PIVSm gli invia la challenge contenente soltanto il seme da utilizzare per inizializzare il generatore di numeri pseudo-casuali.



Messaggio M3 (*response*, ovvero la hash *Y’* ottenuta dal sensore)

Dopo aver calcolato la *hash*, il sensore invia al server PIVSm un messaggio di tipo M3 contenente la hash che ha ottenuto, ovvero la matrice Y[][]. I bytes della matrice vengono inseriti nel messaggio nel seguente ordine: Y[0][0], Y[0][1], …, Y[0][3], Y[1][0], Y[1][1], …, Y[3][3].

Alternativamente, è possibile descrivere questa operazione col codice:

creazione del messaggio M3

```
for (i=0;i<k;i++)
for (j=0;j<k;j++)
      payloadM3->hash[(i*k)+j] = Y[i][j];
```

7. Implementazione del server PIVSm

Il prototipo del server PIVSm è stato implementato in linguaggio Java. Analogamente al caso del client PIVCm, per il server PIVSm sono state implementate procedure per il calcolo della *hash*, ovvero la RHF vista nel par 5.1.1, le procedure per effettuare l'*interleaving* dinamico con un generatore pseudo-casuale e la libreria matematica per effettuare le operazioni aritmetiche in GF(2^8). Questa scelta implementativa è stata determinata dal fatto che erano già disponibili, per Java, delle librerie di comunicazione tra un Personal Computer e una rete di sensori.

7.1 Applicazione

La struttura della RHF implementata nel server PIVSm è molto simile a quella della RHF implementata nel client PIVCm, per cui non verrà ripetuta l'analisi dei dettagli implementativi che sono già stati analizzati nel capitolo 6, bensì verranno soltanto mostrate le differenze sostanziali che distinguono le due procedure successivamente verrà analizzato il diagramma degli stati del server nonché il suo diagramma delle attività. Come è stato visto nel capitolo 5, la modifica effettuata al protocollo PIV impedisce di implementare una funzione di *hash* che sfrutti l'equazione 5.2 per effettuare i calcoli, quindi è stato necessario creare un server che eseguisse le stesse operazioni che esegue il client, ovvero delle operazioni che implementano l'equazione 5.1.

7.1.1 Tipi di dati

In Java, tra i tipi primitivi concessi dal linguaggio non vi sono numeri interi senza segno, per cui l'operazione di trasporto del codice in linguaggio Java non è stata banale. La scelta implementativa effettuata è stata di estendere tutti i tipi di dati presenti nella RHF a tipi più grandi, ai quali poi sono stati aggiunti i dovuti controlli affinché questi non uscissero mai dagli intervalli previsti dai tipi originali.

- I dati di tipo **uint8_t** (nesC) sono stati promossi a **int** (Java), includendo controlli che impedissero loro di uscire dall'intervallo [0,255].
- I dati di tipo **uint16_t** (nesC) sono stati promossi a **int** (Java), includendo controlli che impedissero loro di uscire dall'intervallo $[0,2^{16}-1]$.
- I dati di tipo **uint32_t** (nesC) sono stati promossi a **long** (Java), includendo controlli che impedissero loro di uscire dall'intervallo $[0,2^{32}-1]$
- I dati di tipo **uint64_t** (nesC) sono stati trasformati in **long** (Java), includendo controlli che impedissero lori di uscire dall'intervallo $[0,2^{63-1}]$.

La trasformazione dei dati di tipo **uint64_t** (nesC) in dati di tipo **long** (Java) è stata possibile soltanto perché il tipo **uint64_t** viene usato soltanto per una variabile all'interno del generatore di numeri pseudo-casuali, e questa variabile non può mai uscire dall'intervallo $[0,2^{46}-1]$. In caso contrario sarebbe stato necessario realizzare un tipo di dati appositamente progettato per contenere interi positivi di valore superiore a 2^{63}.

7.1.2 Diagramma a stati di PIVSm

In figura 7.1 è mostrato il diagramma a stati del server PIVSm. Il server trascorre la maggior parte del tempo nello stato '*Wait State*' in attesa che i sensori chiedano l'accesso alla rete. Quando riceve una richiesta da un sensore il server seleziona una chiave di inizializzazione per il PIVCm, manda il messaggio M2 (contenente la chiave selezionata) al sensore e fa partire un timer.

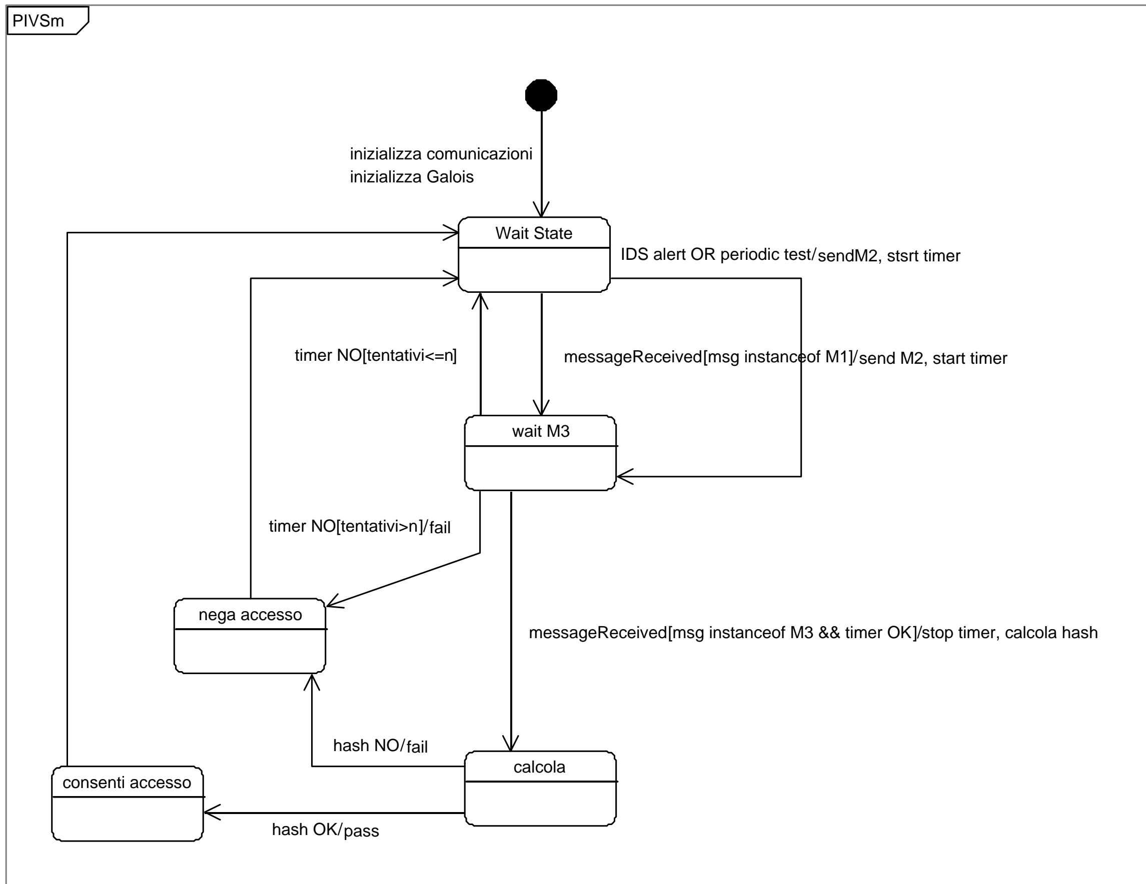


Figura 7.1: Diagramma a stati del server PIVSm

Se il timer supera una deadline stabilita al momento dell'invio della richiesta, il server interrompe il test e torna nello stato Wait State, ma rimane comunque traccia del tentativo fallito. Se il sensore ha già fallito più di un certo numero di test, gli viene definitivamente negato l'accesso alla rete. Se il server riceve un messaggio M3 dal sensore entro la deadline stabilita, calcola la hash relativa a quel sensore in base all'immagine della memoria in suo possesso. Dopo aver effettuato il calcolo il server confronta le due hash, quella ricevuta dal sensore e quella calcolata, e decide se consentire l'accesso alla rete a quel sensore oppure se negargli l'accesso alla rete.

Inoltre, è possibile vedere che il server, periodicamente oppure dopo aver ricevuto un *alert* da un servizio di IDS (*Intrusion Detection System*), come è stato illustrato nel paragrafo 4.1.20, può richiedere la verifica di un sensore, inviando un messaggio M2 ad un sensore, senza prima aver ricevuto alcun messaggio M1.

7.1.3 Le immagini della memoria dei sensori

Il server, per funzionare correttamente, deve possedere delle immagini della memoria dei sensori, ottenute dopo l'installazione delle applicazioni sui sensori. Per ottenere queste immagini è necessario collegare i sensori ad un Personal Computer su cui è stato installato l'ambiente di sviluppo di TinyOS e utilizzare il programma msp430-bsl (il programma utilizzato per installare le applicazioni sui sensori). All'interno di una *shell* di comando, dopo aver scoperto a quale porta è collegato il sensore con il comando `motelist`, è necessario digitare:

```
> msp430-bsl –c COMn –P path_dell_applicazione/main.exe --tmote --
upload=0x4000 --size=0xC000 –b > ID.dump
```

Dove *n* è il numero della porta seriale alla quale è connesso il sensore, *path_dell_applicazione* è il percorso del filesystem in cui si trova l'applicazione che è stata installata sul sensore e *ID* è l'identificativo di rete che è stato assegnato al sensore al momento dell'installazione.

7.1.4 Il package pivs e la classe pivserver

Per implementare il server PIVSm è stata creata un *package* **pivs**, del quale fanno parte le classi **pivserver**, **M1**, **M2** ed **M3**. Queste ultime tre classi sono state derivate per ereditarietà dalla classe **Message**, del package **net.tinyos.message** fornito insieme all'ambiente di sviluppo di TinyOS.

La classe **pivserver** contiene le funzioni **inizializza()**, **calcola()**, **scegliMBlocchi()** e **memoria(int, int)**, con la differenza che la funzione **memoria(int, int)** legge i bytes da un vettore **dump[]** in cui, durante la fase di inizializzazione, viene caricato il contenuto dell'immagine della memoria del sensore. In questo modo è stato possibile riutilizzare gran parte del codice scritto per il modulo **PivcM**, salvo ovviamente averlo tradotto in linguaggio Java.

7.1.5 Il generatore di numeri pseudo-casuali

Per effettuare una verifica dell'integrità di un sensore, il server deve poter generare la stessa *hash* che calcolerebbe il sensore (ovviamente considerando che la sua memoria sia integra) se interrogato. Questo implica che, siccome la *hash* calcolata dipende anche dalla sequenza di numeri pseudo-casuali generati, è necessario che il generatore di numeri presente sul server sia identico al generatore installato sul

sensore. Ovvero, è necessario che, se inizializzato con lo stesso seme, esso produca la stessa sequenza di numeri.

Per questo motivo è stata implementata una versione del generatore MLCG in linguaggio Java. Questo generatore è accessibile tramite due funzioni: la funzione **initSeed(int)** che inizializza il generatore con il seme che viene fornito come parametro e la funzione **rand()** che genera il prossimo numero della sequenza e lo restituisce alla funzione chiamante tramite un parametro di tipo **int**.

7.1.6 la funzione calcola()

La funzione **calcola()** presente nel server è molto simile all'implementazione presente nel client. Ovviamente, in questo caso non è stato necessario suddividere il calcolo in due procedure distinte, ma è stato sufficiente scrivere una procedura che esegue le seguenti operazioni:

Funzione calcola()

```
int   partizioni_attuali[m]
int   vettore[sup(B/m)]
int   x[m]
int   z[k]
int   Y'[k][k]
int   Y[k][k]
int   H[k][m]
int   g[B]
int   l     (0 < l < B)
```

```
public void calcola () {
      int macrobloc,bloc;

      for (macrobloc=0; macrobloc<B/m; macrobloc++) {
            scegliMPartizioni();
            for (bloc=0; bloc<m; bloc++) {
                  riempi_x(bloc);
                  <z[]= H[][]*x[]>;
                  <Y'[]= z[]*zT[]> ;
                  <Y = Y + g[l]*Y'[]> ;
                  l++;
            }
      }
}
```

In questa variante semplificata della funzione **calcola()** è possibile vedere che ad ogni esecuzione del ciclo più esterno vengono selezionate **m** partizioni con le quali,

nel ciclo più interno, vengono creati **m** blocchi di dati sui quali poi vengono effettuate le operazioni necessarie per implementare l'equazione 5.2. Ovviamente qui non sono stati rappresentati i dettagli delle operazioni aritmetiche su matrici e vettori, ma va comunque fatto notare che tutte le operazioni di somma e moltiplicazione devono essere eseguite in $GF(2^8)$, considerando che questi elementi appartengono a $GF(2^8)$. Quindi è necessario utilizzare anche in questo caso una libreria matematica per le operazioni aritmetiche nel campo di Galois-Rijndael $GF(2^8)$.

7.1.7 La funzione scegliMPartizioni()

La funzione **scegliMPartizioni** effettua le stesse operazioni che sono state illustrate nel capitolo 6, riguardo all'omonima funzione del modulo **PivcM**.

7.1.8 La funzione riempi_x(int)

La funzione **riempi_x(int)** effettua le stesse operazioni che sono state illustrate nel capitolo 6, riguardo all'omonima funzione del modulo **PivcM**.

7.1.9 La funzione memoria(int, int)

La funzione **memoria(int, int)**, come è già stato accennato nel paragrafo 6.3.5 e nel paragrafo 7.2.4, deve essere modificata ogni volta che la funzione RHF modificata, ovvero la funzione che calcola la *hash* secondo l'algoritmo PIV modificato visto nel capitolo 5, viene trasportata su una piattaforma diversa. Questa funzione si occupa di tradurre gli indirizzi espressi secondo due indici:

- indice di una partizione fisica
- indice di un byte all'interno di una partizione

in un indirizzo fisico che abbia senso all'interno di una data architettura. Ovviamente, siccome il server PIVSm effettua il calcolo della *hash* su un immagine della memoria di un sensore, la quale è conservata in un file, il server dovrà semplicemente caricare questa immagine in memoria RAM, ovvero all'interno di un vettore di interi (**int**) e leggere i bytes contenuti in questo vettore. Il file contiene una

copia dei bytes della memoria flash del sensore, a partire dall'indirizzo 0x4000h fino all'indirizzo 0xFFFFh del sensore.

NOTA: Le partizioni fisiche descritte nella sezione 5.3 non sono necessariamente contigue. Ovvero, su alcune architetture, soprattutto quando lo spazio di indirizzamento è composto da più memorie e quando sono presenti delle memorie esterne non direttamente accessibili dal processore, le partizioni fisiche potrebbero non essere contigue e la funzione **memoria(int, int)** ha anche il compito di pilotare i *controller* di queste memorie esterne.

In questo caso, il server PIVSm accede al file contente l'immagine della memoria del sensore. Questa viene caricata nel vettore **dump[]** durante la fase di inizializzazione dell'algoritmo, e la funzione **memoria**, quando viene chiamata con dei parametri *a* e *b*, ovvero **memoria (int a, int b)** deve soltanto accedere al byte di indice **(a·m)+b** del vettore **dump[ ]**, ovvero **dump[(a·m)+b]**.

7.2 Formato dei pacchetti

I pacchetti scambiati tra il server ed il client sono quelli già visti nel paragrafo 6.3.6.

8. Test del sistema

Tutti i moduli e le funzioni componenti il sistema, sia dal lato *client* che dal lato *server* sono stati testati in maniera estensiva. Trattandosi nella maggior parte dei casi di funzioni che implementano operazioni matematiche, esse sono state analizzate staticamente per individuare eventuali errori, sono state analizzate dal punto di vista dei *tipi di dati* utilizzati e i loro possibili intervalli, ed infine ne è stato fatto fatto un *testing* estensivo comprendente tutti i possibili valori in ingresso, per verificare se i valori in uscita corrispondevano alle specifiche.

Inoltre è stata creata una versione delle due applicazioni in cui, prima di ogni operazione che svolge calcoli su matrici (o vettori), veniva effettuato un controllo sui valori degli indici per stabilire se l'algoritmo stesse commettendo un errore accedendo a valori inesistenti, oltre i limiti delle suddette matrici (o vettori). Questi controlli sono stati poi tolti, ovviamente, per rendere l'implementazione più snella ed efficiente.

Per fare un esempio, considerando che il vettore **x[ ]** contiene **m** elementi, prima di effettuare un accesso ad esempio ad **x[i]**, è utile verificare se **i** si trova nell'intervallo $0 < \mathbf{i} < \mathbf{m}$. Solitamente, questa operazione di verifica permette di individuare moltissimi errori che si annidano nel codice e che non vengono segnalati dai compilatori.

Inoltre, dovendo sviluppare due applicazioni distinte, in due linguaggi di programmazione diversi, è stato introdotto un ulteriore livello di *testing*, per verificare che le due implementazioni delle funzioni fornissero gli stessi risultati, a partire dagli stessi ingressi.

8.1 La libreria delle operazioni aritmetiche in GF(2^8)

Essendo questa libreria il punto cardine di tutte le operazioni aritmetiche, è stato necessario, ma anche ovvio, verificare che il suo funzionamento fosse privo di errori. In particolare, per ognuna delle due implementazioni è stato verificato che, per ogni valore dei parametri di input, i valori in uscita fossero uguali.

Inoltre, per le due operazioni **Product(a,b)** e **Quotient(c,d)** è stata effettuata una verifica per stabilire se, per ogni valore di **a** e **b**, sono vere le relazioni:

Quotient(Product(a,b),a) == b (8.1)

Quotient(Product(a,b),b) == a (8.2)

Inoltre su queste sue funzioni è stata effettuata la verifica degli indici con cui si indicizzano i vettori **Log[ ]** e **ALog[ ]**, per evitare accessi fuori dai limiti di questi vettori.

8.2 Il generatore di numeri pseudo-casuali

Per verificare il corretto funzionamento dei due generatori di numeri casuali, essendo che il prossimo numero generato dipende soltanto dallo stato interno ed essendo che lo stato interno è espresso con un numero intero positivo a 16 bit (**uint16_t**), per verificare la coerenza dei due generatori è stato sufficiente inizializzare i due generatori con tutti i possibili semi (2^{16} possibili valori) e verificare che il primo numero prodotto dai due algoritmi fosse uguale, per ogni valore del seme.

8.3 La RHF, ovvero i task calcola(), calcolaBlocco()(lato client) e la funzione calcola() (lato server)

Per verificare il corretto funzionamento della RHF non è stato fatto un test esaustivo, considerando che i tempi di esecuzione e di comunicazione tra sensore e server non lo permettono. In ogni caso, grazie all'analisi statica e avendo effettuato un buon

numero (>500) di test, in cui il server ed il client hanno prodotto la stessa hash, è ragionevole affermare che i task **calcola()**, **calcolaBlocco()** (lato *client*) e la funzione **calcola()** (lato *server*) sono prive di errori.

Inoltre su questi task e sue funzioni è stata effettuata la verifica degli indici con cui si indicizzano i vettori e le matrici, per evitare accessi fuori dai loro limiti.

8.4 Le funzioni memoria(int,int) e riempi_x(int)

Queste funzioni, essendo parte integrante della RHF, sono state testate insieme a quest'ultima. Siccome non è stato possibile effettuare un *testing* esaustivo, a causa dei grandi numeri in gioco, ci si è affidati nuovamente al confronto tra i valori prodotti da *client* e *server*. In particolare è stato verificato se le due implementazioni fanno gli stessi accessi in memoria per comporre un determinato blocco, selezionato a caso. Inoltre su queste funzioni è stata effettuata la verifica degli indici con cui si indicizzano i vettori e le matrici, per evitare accessi fuori dai loro limiti.

8.5 La funzione scegliMPartizioni

Questa funzione è stata testata verificando che, dato un seme di inizializzazione per l'algoritmo di generazione di numeri pseudo-casuali, il client ed il server selezionassero le partizioni nella stessa sequenza.

9. Conclusioni

Dopo aver realizzato le due applicazioni, la prima per i sensori Tmote Sky in linguaggio nesC e la seconda per Personal Computer in linguaggio Java e dopo averne effettuato il testing, queste due applicazioni sono state messe in funzione per effettuare delle verifiche di integrità e contestualmente fare delle misurazioni.

9.1 Comportamento del protocollo

Le due applicazioni si comportano effettivamente come previsto dal protocollo ed in effetti è stato possibile appurare che anche piccolissime modifiche nella memoria del sensore oppure nel file immagine producevano l'effetto desiderato, ovvero un cambiamento *totale* nel valore della *hash* calcolata.

Dalle verifiche effettuate è risultato che questo protocollo riesce a rilevare anche modifiche di un singolo bit in una qualsiasi posizione della memoria. Ovviamente, lo stesso vale se le modifiche vengono effettuate nel file di immagine contenuto nel server.

9.2 Tempi di esecuzione

Effettuando delle misurazioni si è determinato che l'implementazione di PIVCm, una volta installata sui sensori, è in grado di calcolare una hash su 48 KBytes in soli 3,4 secondi.

Purtroppo, per effettuare un confronto con l'algoritmo PIVC originale proposto in [3], non è stato possibile ottenere il codice dell'applicazione sviluppata dagli autori di PIV. Di conseguenza i confronti sono stati effettuati utilizzando l'applicazione sviluppata nell'ambito di questa tesi, avendo disattivato il supporto per l'interleaving dinamico e facendo quindi in modo che il suo comportamento fosse uguale a quello previsto da PIVC originale. Effettuando delle misurazioni su quest'ultima si è determinato che essa è in grado di calcolare una hash in 3,05 secondi risultando quindi di 0,35 secondi più veloce.

Di conseguenza ora è possibile affermare che la modifica introdotta a PIV impedisce la parallelizzabilità di PIVC comportando un *overhead* computazionale di

soli 0,35 secondi, ovvero +11,5% rispetto al tempo di esecuzione di PIVC nella sua forma originale.

9.3 Overhead di comunicazione

Come è già stato visto nel paragrafo 5.4.1, il protocollo PIVm non prevede che il server invii al sensore da verificare né la matrice H, né il vettore g, né il codice eseguibile della funzione PIVCm. In questo modo è possibile evitare di trasmettere 1329 bytes ad ogni verifica.

E quindi è possibile vedere che la modifica introdotta da PIVm permette di risparmiare $381 \cdot 10^{-6}$ *mAh* per ogni verifica e per ogni nodo intermedio, ovvero un risparmio del 97,9% sul consumo che il protocollo PIV comportava sull'intera rete.

9.4 Sicurezza ed efficienza

Le modifiche proposte al protocollo PIV si rivelano quindi doppiamente vantaggiose. Il protocollo PIVm, con un piccolo overhead computazionale pari a +11,5%, permette alla rete di resistere ad attacchi fisici di livello 1 e livello 2 effettuati sui sensori. Inoltre, il protocollo PIVm permette di risparmiare il 97,9% dell'energia che nel protocollo PIV veniva usata per trasferire dati tra server e client.

Bibliografia

[1] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. *SWATT: Software-based attestation for embedded devices*. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 2004.

[2] M. Shaneck, K. Mahadevan, V. Kher and Y. Kim. *Remote Software-based Attestation for Wireless Sensors*. Security and Privacy in Ad-hoc and Sensor Networks, Second European Workshop, ESAS 2005, Visegrad, Hungary, July 13-14, 2005.

[3] T. Park, KG Shin. *Soft tamper-proofing via program integrity verification in wireless sensor networks*. IEEE transactions on mobile computing, 2005, Volume: 4, Issue: 3, May-June 2005.

[4] R. Kennell and L. H. Jamieson. *Establishing the genuinity of remote computer systems*. In Proceedings of the 12th USENIX Security Symposium, pages 295-308, Aug 2003.

[5] Texas Instruments. *Mixed Signal Microcontroller MSP430F1611 product sheet*. Reperibile da http://focus.ti.com/docs/prod/folders/print/msp430f1611.html.

[6] Sariel Har-Peled. *The Stable Marriage problem and the Coupon Collector problem*. Reperibile da http://valis.cs.uiuc.edu/~sariel/teach/2002/a/notes/06_collector.pdf.

[7] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, Ke Yang, *On the (Im)possibility of Obfuscating Programs*, Advances in Cryptology -- CRYPTO'01, *Springer Lecture Notes in Computer Science* vol. 2139, pp. 1-18, Santa Barbara, CA, November 2001.

[8] W. Kaiser, G. Pottie. *The Balance Between Local Computation and Communications in Widely Distributed Wireless Embedded Systems*. Sensoria Corporation, 15 January 2000.

[9] D.W. Carman, P.S. Kruus, and B.J. Matt. *Constraints and Approaches for Distributed Sensor Network Security*. NAI Labs Technical report #00-010, Sept. 2000.

[10] RJ Anderson, MG Kuhn. *Tamper Resistance - a Cautionary Note*. The Second USENIX Workshop on Electronic Commerce Proceedings, November 1996.

[11] D. L. Lough. *A Taxonomy of Computer Attacks with Applications to Wireless Networks*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, April 2001.

[12] DG Abraham, GM Dolan, GP Double, JV Stevens. *Transaction Security System*. IBM Systems Journal v 30 no 2, pp 206-229, 1991.

[13] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. *Dynamic self-checking techniques for improved tamper resistance*. In *Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001)*, volume 2320 of *Lecture Notes in Computer Science*, pages 141–159. Springer-Verlag, 2002.

[14] H. Chang and M. Atallah. *Protecting software code by guards*. In *Proc. 1st ACM Workship on Digital Rights Management (DRM 2001)*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer-Verlag, 2002.

[15] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinba, and M. Jakubowski. *Oblivious hashing: A stealthy software integrity verification primitive*. In *Proc. 5th Information Hiding Workship (IHW)*, volume 2578 of *Lecture Notes in Computer Science*, pages 400–414, Netherlands, Oct.2002.

[16] P. C. Van Oorschot. *Revisiting Software Protection*. In *6th International Information Security Conference* (ISC 2003), pages 1–13, Springer LNCS 2851, 2003.

[17] Ross Anderson and Markus Kuhn. *Low cost attacks on tamper resistant devices*. In *International Workshop on Security Protocols*, pp. 125—136, 1997.

[18] Robert Sedgewick. *Permutation Generation Methods*, ACM Computing Surveys (CSUR), v.9 n.2, p.137-164, June 1977.

[19] Moteiv. *Tmote Sky Datasheet*. Reperibile da http://www.moteiv.com.

[20] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, D. Culler. *The emergence of networking abstractions and techniques in TinyOS*. In *First Symposium on networked system design and implementation (NSDI04)*, pages 1–14, San Francisco, California, USA, 2004.

[21] P. Buonadonna, J. Hill, D. Culler. *Active message communication for tiny networked sensors.* In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies* (INFOCOM'01), April 2001.

[22] D. Gay, P. Levis, J. Robert von Behren, M. Welsh, EA. Brewer, DE. Culler. *The nesC language: A holistic approach to networked embedded systems*. In PLDI, pages 1–11, 2003.

[23] P. Levis. *TinyOS programming*. Reperibile da http://csl.stanford.edu/%7Epal/pubs/tinyos-programming-1-0.pdf, June 2006.

[24] IEEE Computer Society. *IEEE Std 802.15.4™ - Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*, October 2003.