Politecnico di Milano
V Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di di Elettronica e Informazione

Applicazione automatizzata di contromisure ad attacchi side-channel basati sull'analisi del consumo di potenza per primitive crittografiche a chiave simmetrica

Relatore: Prof. Giovanni AGOSTA
Correlatore: Ing. Alessandro BARENGHI
Correlatore: Ing. Gerardo PELOSI

Tesi di Laurea di:
Francesco FIDUCCIA Matr. 735152

Anno Accademico 2013–2014

To Valentina

For love and help

To my sister and mother
You will never walk alone

To the silence
Full with voices gone

Acknowledgements

I wish to thank, first and foremost, Valentina, who has always stood by me and dealt with all my absence.

I dedicate this thesis to my family who has always supported me and has given me the opportunity of an education from the best institutions and support throughout my life.

I would like to acknowledge the professor and the correlators for the help and tutoring received.

I would like to thank my friends who have always helped me and believed that I could do it.

I have never forgot my very first computer programming lesson and the teacher that for the first time has shown me the “computer language”. It has been a long road since then.

"Begin at the beginning," *the King said gravely*,
"and go on till you come to the end: then stop."

Lewis Carroll, Alice in Wonderland

Sommario

L'obiettivo di questo lavoro di tesi è stato l'implementazione di un framework per l'applicazione di contromisure software, per la protezione degli algoritmi crittografici da attacchi di tipo *side channel*.

L'introduzione presenta un riassunto della storia della crittografia e della crittoanalisi a partire dalla prima guerra mondiale, ripercorrendo l'introduzione delle macchine elettromeccaniche capaci di eseguire i passi degli algoritmi crittografici e gli attacchi. Il capitolo termina con l'introduzione degli attacchi di tipo Side Channel, che, come ogni attacco ad un algoritmo crittografico, hanno come obiettivo finale la ricostruzione della chiave dello stesso, ma che si basano, per la ricostruzione delle informazioni sulla chiave, sulla rilevazione dei segnali elettrici generati dal dispositivo che esegue l'algoritmo crittografico.

Nel capitolo Stato dell'arte sono presentati nel dettaglio diversi tipi di attacchi *side channel*, nello specifico: le tipologie di attacchi Simple Power Analysis e Differential Power Analysis del primo e del secondo ordine. Il generico attacco di Power Analysis, sfruttando un Side Channel, è in grado di ricostruire la chiave crittografica a partire dai segnali raccolti, facendo uso di modelli statistici del consumo di potenza del dispositivo che esegue l'algoritmo crittografico.

Sia in questo lavoro che in letteratura, si identifica con il termine *power trace* il segnale elettrico raccolto durante un'esecuzione dell'algoritmo. L'approccio di analisi dei campioni è possibile perché i grafi delle power trace sono correlabili nel tempo; per esempio sono di facile individuazione i round di operazioni ripetute come i 16 round dell'algoritmo DES. Il lavoro di tesi

comprende un'introduzione alla DPA del DES attraverso la presentazione, dal punto di vista teorico, di un attacco presente in letteratura in [Kocher et al., 1999]. Una volta dimostrata la fattibilità di un attacco di questo tipo, la ricerca crittoanalitica, in letteratura, si è concentrata sull'individuazione delle power trace utili per l'analisi, al fine di ridurre le risorse computazionali necessarie e al fine di stimare la dimensione del campione da raccogliere.

Tra gli attacchi che fanno uso di tecniche di filtraggio presenti in letteratura, in [Barenghi et al., 2010] viene descritto un attacco realizzato contro un micro-controllore ARM Cortex M3 in grado di ridurre di un ordine di grandezza il numero di power trace necessarie per eseguire un attacco.

Una sezione del capitolo dello stato dell'arte presenta una descrizione dei tipi di contromisure applicabili contro un attacco di tipo DPA, illustrando sia l'eventuale implementazione in hardware che l'implementazione in software; le tecniche descritte sono quelle di hiding, masking e random precharging. Questi approcci alterano il flusso di operazioni da eseguire al fine di nascondere sia l'operazione eseguita che il valore degli operandi delle istruzioni. Se dal valore degli operandi è facile ricondursi alla chiave dell'algoritmo crittografico, attraverso l'alterazione della sequenza di operazioni è possibile rendere più complessa la sovrapposizione delle power trace, disallineando gli eventi temporali e di conseguenza richiedendo di incrementare le power trace da raccogliere.

La sezione si conclude presentando l'implementazione di un framework presente in letteratura [Agosta et al., 2012], per l'applicazione automatica di contromisure software, che utilizza tecniche di masking e hiding per proteggere l'algoritmo crittografico da DPA; essendo tale implementazione alla base di questo lavoro, si è scelto di soffermarsi maggiormente nel presentare la sua struttura.

Nel capitolo Implementazione sono formulate le specifiche funzionali alla base del framework realizzato: un framework per sviluppatori, integrabile facilmente con il codice sorgente da proteggere e che permetta di configurare un trade-off tra aumento di richieste computazionali e protezione. I dispositivi e quindi l'instruction set di riferimento sono ARM, nello specifico ARMv5 e ARMv7. Le contromisure scelte sono implementate in due moduli che inter-

vengono a runtime per modificare il corpo delle funzioni da proteggere. Un modulo realizza lo shuffling che altera la sequenza di esecuzione di operazioni su un blocco di dati, come per esempio un array, alterando la variabile indice di iterazione; l'altro modulo realizza il morphing, alterando le istruzioni a runtime e realizzando *instruction substitution*, *random instruction insertion* e *masking* dei valori.

Sono successivamente presentati nel dettaglio i componenti del framework e l'integrazione degli stessi nel codice dell'utilizzatore. Il framework si basa sul compilatore Clang, a sua volta parte di LLVM; questa scelta è stata basata sulla disponibilità e maturità del codice sorgente e sulla sua architettura interna, che permette facilmente di introdurre delle estensioni. Per la realizzazione del framework sono stati infatti creati degli attributi per marcare le definizioni di variabili e funzioni, al fine di realizzare le funzionalità dei moduli delle contromisure; volendo fare un esempio, gli attributi specificano delle relazioni tra le funzioni da proteggere e il gestore del morphing e introducono la definizione delle variabili di induzione da modificare con lo shuffling.

Gli attributi sono riconosciuti dal parser di Clang e interpretati dal generatore del codice intermedio per modificare le chiamate a funzioni e gli accessi a variabili, ed anche per generare dei meta-dati aggiuntivi che, nel nostro caso, sono stati usati dall'emettitore di codice ARM per valorizzare delle variabili calcolate al tempo di emissione del codice ARM. Per fornire un quadro più completo del sistema sono presentate anche le strutture dati utilizzate all'interno del framework.

La sezione procede presentando le informazioni necessarie all'utilizzatore per realizzare l'integrazione del framework nel proprio progetto, attraverso variabili e macro preparate allo scopo. Una sezione a parte è dedicata all'analisi dei problemi riscontrati in fase di sviluppo del modulo di morphing e collegati alla gestione della cache applicativa a bordo della CPU: poiché il codice da modificare si trova prima in una sezione di memoria marcata dati e successivamente passa ad una sezione marcata eseguibile, questo cambio di stato può richiedere o meno, in base all'architettura implementata dal dispositivo, un flush della cache per poter mantenere consistente la memoria di

programma.

Per i moduli delle contromisure, morphing del codice e shuffling delle operazioni su array, sono descritti nel dettaglio i passi svolti durante l'esecuzione a runtime, distinguendo tra azioni svolte in inizializzazione, durante l'esecuzione vera e proprio dell'algoritmo crittografico e in fase di finalizzazione. Vengono successivamente specificati gli attributi da utilizzare per l'integrazione da parte dei moduli e i parametri di configurazione del framework, nello specifico la tabella con l'equivalenza delle istruzioni, usata dal modulo di morphing, e la parametrizzazione del framework per implementare il trade-off configurabile. La parametrizzazione del trade-off è implementata con una costante di configurazione che esprime ogni quante esecuzioni della funzione crittografica standard uno dei moduli del framework debba essere eseguito, per esempio ogni 100 chiamate alla funzione crittografica.

Nel capitolo è anche disponibile una sezione sull'istruzione ARMv5 *LDR* che ha richiesto particolare cura da parte del sotto-modulo che realizza la sostituzione del codice.

Il capitolo Verifica e risultati sperimentali espone i dati raccolti durante la fase di verifica dell'implementazione. La prima sezione presenta i test implementati per la verifica delle funzionalità dei moduli, organizzati singolarmente per ciascuno degli attributi. Questa parte fornisce sia informazioni al possibile sviluppatore che vuole estendere il framework, sia informazioni aggiuntive sul funzionamento interno dei moduli.

La sezione sulla metodologia di verifica delle performance e dell'overhead dell'implementazione espone le differenti implementazioni algoritmiche di AES scelte per la verifica; a partire da queste, descrive quindi i modi con cui sono stati scelti i campioni per la verifica, iniziando dalla parametrizzazione del trade-off computazionale, passando alle opzioni di compilazione per finire con i diversi dispositivi fisici utilizzati per eseguire l'algoritmo. La distribuzione e la raccolta dei risultati sono state automatizzate con la realizzazione di un framework in python per la compilazione ed esecuzione. Il profiling è stato realizzato raccogliendo il tempo di esecuzione con l'utilizzo della funzione di sistema operativo *clock_gettime* e l'analisi numerica dei tempi raccolti attraverso la libreria di calcolo scientifico *numpy*.

Le aspettative prospettate prima di affrontare l'analisi dei risultati sono raccolte nella parte iniziale della sezione Analisi dei risultati e, per ognuna di esse, sono presentati i dati raccolti dai campioni.

È stata verificata come vera l'assunzione che i tempi di esecuzione potessero migliorare o rimanere costanti al crescere del livello di ottimizzazione, quindi l'implementazione del modulo di sostituzione del codice non degrada la performance.

Non è stato riscontrata invece l'assunzione che, per la presenza dell'operazione di flush della cache nel modulo di morphing, si dovesse verificare una degradazione delle performance al crescere della frequenza di esecuzione del morpher. I dati raccolti hanno invece mostrato che i tempi medi di esecuzione della funzione dell'algoritmo crittografico protetta dal morpher rimanevano inalterati al variare del parametro di attivazione del modulo morpher.

Sono stati invece rilevati, per il modulo di shuffling, dei tempi di esecuzione crescenti al diminuire della frequenza di attivazione del modulo. Dai dati rilevati per il tempo di esecuzione del modulo di shuffling, si nota un incremento del 10% del tempo di esecuzione passando da un'attivazione del modulo ogni 100 esecuzioni della funzione ad un'attivazione ogni 1000.

In generale, i dati raccolti mostrano una concentrazione elevata attorno al minimo, e presentano pochi valori fuori scala rispetto al 90-esimo quartile; la media si trova sempre tra la mediana e il 75-esimo quartile e prossima, per lo più, a quest'ultimo. È stato infine stimato per ognuno dei moduli il costo computazionale dello stesso rispetto alla funzione crittografica AES; i dati così raccolti sono stati tabulati per permettere di individuare la parametrizzazione più adatta da utilizzare per il framework. Per le contromisure di tipo code morphing è infatti disponibile in letteratura, in [Agosta et al., 2012], una stima dell'incremento computazionale richiesto per portare a termine un attacco DPA. A fronte di un'attivazione del morpher ogni 2000-3000 iterazioni dell'algoritmo crittografico, il dispositivo presentava un'ottima resistenza ad un attacco DPA (stimata al 79%). Con questo lavoro si è raggiunto l'obbiettivo di integrare in un framework, alla portata dello sviluppatore, delle metodologie per applicare delle contromisure ad un algoritmo crittografico. L'implementazione è stata inoltre in grado di mantenere le aspettative in

termini di performance e, grazie alle modifiche apportate al modulo di sostituzione del codice, è riuscita a gestire correttamente le istruzioni ARMv5 (LDR).

Nelle conclusioni ci si sofferma sui perfezionamenti che è possibile applicare all'implementazione corrente: i) abbattere il costo di esecuzione della sostituzione del codice attraverso la parallelizzazione multi-threaded oppure con il precalcolo a compile-time di blocchi di codice alternativi; ii) perfezionare il modulo di shuffling per eseguire dei controlli sul codice e sulle variabili da modificare.

Indice

Elenco delle figure

Elenco delle tabelle

Listings

Capitolo 1

Introduzione

> ... It must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience
>
> 1883, Auguste Kerckhoffs

> The enemy knows the system
>
> Unsourced, Claude Shannon

Come postulato dalla massima di Shannon, nonché secondo principio di Kerckhoffs, la sicurezza di un algoritmo crittografico dipende dalla segretezza della sua chiave e non dalla riservatezza dell'implementazione.

Il presente capitolo introduce la storia della crittografia e i principali concetti tecnici di crittoanalisi che sono alla base della presentazione della sezione sugli attacchi agli algoritmi crittografici. Vengono introdotti in maniera approfondita gli attacchi di tipo *side channel* e le relative contromisure presentate in letteratura. Infine vengono affrontate nello specifico le contromisure e le implementazioni significative presenti in letteratura e quelle utilizzate per questo lavoro.

1.1 Crittografia e attacchi side channel

Per questa introduzione alla crittografia e alla crittoanalisi si farà riferimento agli eventi dell'era moderna, che precedono e seguono l'avvento dell'elettronica e dell'informatica.

1.1.1 Crittografia del 20esimo secolo

Il balzo evolutivo e applicativo compiuto dalla crittografia e dalla crittoanalisi nell'era moderna è ben evidente nella storia dei confitti del ventesimo secolo. La sua evoluzione in termini tecnologici e applicativi segue quella dell'evoluzione delle macchine automatiche per il calcolo, come le macchine elettromeccaniche, attraverso l'elettronica fino all'informatica. L'evoluzione in termini teorici della crittografia ha invece seguito quella della matematica e dell'informatica e specificamente il campo della teoria dei numeri e quello della complessità computazionale.

L'esplosione applicativa della crittografia ebbe inizio dopo la fine del primo conflitto mondiale, con l'introduzione di macchine di supporto alla cifratura di messaggi, come la macchina tedesca Enigma, un apparecchio elettromeccanico che operava utilizzando un cifrario a sostituzione, e successivamente le più avanzate macchine telescriventi Lorenz SZ42.

La prima, essendo portabile e di facile utilizzo, ebbe una vasta diffusione applicativa soprattutto diventando lo standard del servizio delle forze armate tedesche durante il periodo di ascesa nazista e la seconda guerra mondiale. La seconda iniziò a diffondersi dopo l'inizio della seconda guerra mondiale e il suo impiego era riservato alla comunicazione automatizzata di tipo punto-punto dei messaggi tra le alte cariche della catena di comando [1].

Mentre con la crittoanalisi della prima diventarono accessibili un ampio volume di messaggi scambiati, con la seconda diventarono accessibili informazioni chiave, perché al seppur basso volume di messaggi scambiati si contrapponeva una maggiore importanza tattica.

[1] http://www.cryptomuseum.com/crypto/lorenz/sz40/

La crittoanalisi della macchina Enigma era stata completata in Polonia nel 1932 da un gruppo di ingegneri polacchi: Marian Rejewski, Jerzy Ròzycki e Henryk Zygalski e prima dell'inizio del conflitto trasferita agli Alleati. Prima dell'invasione della Francia, la Germania cambiò però parte dei protocolli di comunicazione, rimuovendo dall'intestazione una ripetizione della chiave cifrata, perché si era rivelata essere un elemento che rendeva il sistema più debole. Dall'aumento della complessità di calcolo per la ricostruzione della chiave e dalla necessità bellica di ricostruire i messaggi nel più breve tempo possibile, nacque la necessità di costruire una elettro-macchina per svolgere il compito velocemente; compito che, prima della modifica, si poteva svolgere anche manualmente. Il progetto, guidato da Alan Turing, porto alla costruzione delle macchine Bombe

Gli sforzi svolti in campo crittoanalitico, nel campo della ricerca e anche in termini di numero di operazioni svolte dalle elettro-macchine nell'eseguire la ricostruzione delle chiavi, aumentarono ulteriormente quando si cercò di rendere pratica la decifrazione dell'algoritmo della macchina Lorenz SZ40 e la decifrazione della variante della marina tedesca di Enigma, modificata per funzionare a 6 Rotori per l'utilizzo sugli UBoot. Successivamente, proprio dalla necessità computazionale richiesta per l'attacco della macchina SZ42 di Lorenz, nacque il progetto che diede vita al primo computer programmabile della storia, Colossus Mark 1, progettato da Tommy Flowers secondo le specifiche di von Neuman e ispirato alla universal computing machine di Turin ma ancora non Turing-completo e progettato per svolgere solo operazioni booleane utili per la decifrazione.

La seconda guerra mondiale aveva fornito alla crittografia e alla crittoanalisi un terreno fertile per svilupparsi e diventare elemento chiave della *intelligence* di un conflitto e, come conseguenza, la riservatezza divenne di importanza capitale. Con la fine della seconda guerra mondiale, ad esempio, gli Stati Uniti fecero degli algoritmi crittografici interesse nazionale, impedendone la divulgazione con l'introduzione di leggi speciali contro l'esportazione degli stessi, tuttora vigenti.

La crittografia moderna si fonda profondamente sulla teoria matematica e informatica della complessità computazionale; infatti gli algoritmi

crittografici standardizzati si basano su problemi computazionali complessi.

1.1.2 DES e l'evoluzione della crittoanalisi

Il DES (Data Encryption Standard), algoritmo a chiave simmetrica di 56 bit, progettato da IBM, è stato standardizzato nel 1977 dalla NBS (ora NIST) come standard federale USA (FIPS).

L'introduzione del DES ha rivitalizzato il campo di ricerca della crittoanalisi al di fuori dell'ambiente militare, fornendo, nell'ambito universitario, un soggetto d'analisi comune e di importanza commerciale per la ricerca.

Alla ricerca crittoanalitica del DES, si deve la scoperta sia della crittoanalisi differenziale che della crittoanalisi lineare; entrambe fornivano, in via teorica, una metodologia d'attacco migliore dell'attacco a ricerca esaustiva della chiave o *brute force*.

Per quanto riguarda la crittoanalisi differenziale nello specifico, essa è stata riscoperta pubblicamente nel 1991 e presentata nel lavoro [Biham and Shamir, 1991], ma era già nota al team di sviluppo del DES di IBM e alla NSA. Infatti l'algoritmo DES, unico tra gli altri algoritmi di cifratura a blocchi suoi contemporanei, è risultato essere resistente a questo tipo di attacco e nel 1994 è stato reso pubblico in [Coppersmith, 1994] che, tra le finalità progettuali del DES, vi fosse proprio la resistenza all'attacco differenziale.

L'attacco a crittoanalisi differenziale, nonostante rappresentasse un miglioramento rispetto ad un attacco di tipo *brute force*, è risultato tuttavia impraticabile, vista la necessità di operare con 2^{47} testi in chiaro scelti appositamente, una quantità eccessiva, a quei tempi, per le risorse di un attaccante.

La crittoanalisi lineare è stata scoperta successivamente e presentata pubblicamente nel [Matsui, 1994]; essa non richiede l'uso di testi in chiaro scelti ma solo di testo in chiaro noto, cosa che dal punto di vista pratico rappresenta un notevole miglioramento. Per portare l'attacco sono necessari 2^{47} messaggi in chiaro e come risultato dell'attacco di crittoanalisi saranno noti 14 dei 56 bit della chiave; per gli altri 42 bit sarà necessario procedere con un attacco di tipo brute force.

Questo approccio con equazioni lineari rappresenta un miglioramento in termini di costo computazionale, sceso per la parte brute force a 2^{42}. In lette-

ratura è presente un approccio migliorativo, che utilizza molteplici equazioni lineari e che è stato stimato richiedere 2^{41} plaintext [Kaliski and Robshaw, 1994].

Nel 1997 è stato realizzato un attacco di tipo brute force contro DES che, con l'utilizzo di uno dei primi approcci al calcolo distribuito su internet e con hardware comune dal punto di vista commerciale, ha impiegato 96 giorni a individuare la chiave. L'attacco, come i successivi, era stato sponsorizzato dalla compagnia RSA per portare all'attenzione pubblica la debolezza del DES [Curtin and Dolske, 1998].

L'anno successivo è stato realizzato un nuovo attacco, anch'esso sponsorizzato dalla RSA, questa volta con hardware custom, basato su una CPU speciale DES Cracker, realizzata a partire da design ad hoc per lo scopo di eseguire un attacco *brute force* contro DES. Il sistema è stato realizzato con 1856 CPU ed un costo complessivo di 220.000 dollari; l'attacco è stato completato in 56 ore [EFF, 2014]. Nel 1999 un approccio misto, con lo stesso DES Cracker e una rete di calcolo distribuita, ha completato l'attacco in sole 22 ore.

Sull'onda di questi risultati, dal 1999 era in realtà già consigliato l'utilizzo del Triple DES, una variante del DES che operava sul singolo blocco con 3 passate, mentre dal 2000 è iniziata la ricerca di un sostituto a lungo termine per l'algoritmo DES. Quest'ultimo è rimasto lo standard di riferimento fino al 2002, quando è stato sostituito definitivamente dall'algoritmo Rijndael, dal nome dei due autori [Daemen and Rijmen, 2002], divenuto lo standard AES.

Per quanto riguarda gli attacchi possibili contro il nuovo standard AES, risultano impraticabili sia un attacco con approccio puramente *brute force* sia gli attacchi noti di crittoanalisi differenziale e lineare, considerando le risorse computazionali attualmente disponibili. Qualora nascesse l'esigenza di una maggiore sicurezza, l'algoritmo è stato progettato con lunghezza della chiave variabile ed è quindi adattabile alle nuove esigenze con il solo trade-off di una maggiore complessità computazionale.

1.1.3 Side Channel

Lo scopo finale di un attacco rimane, comunque, quello di recuperare la chiave crittografica; per questo è possibile realizzare, in determinati contesti, un tipo di attacco differente dagli attacchi all'algoritmo crittografico come sono gli attacchi crittoanalitici o la ricerca esaustiva del *brute force*.

In questo attacco tipologicamente diverso che definiamo basato sui *side channel*, si attacca, per ottenere il segreto o parte di esso, l'implementazione dell'algoritmo crittografico: ad esempio una CPU generica, una smart card, un'implementazione FPGA, una CPU Embedded, al fine di rilevare dai segnali da loro emessi durante l'esecuzione dell'algoritmo delle informazioni sullo stato dell'algoritmo.

I contesti in cui si possono realizzare attacchi *side channel* sono caratterizzati dall'esistenza di un canale di comunicazione, da cui deriva il nome *side channel*, e dalla possibilità di effettuare delle rilevazioni di natura fisica sullo stato dell'esecutore hardware.

Nel caso dei dispositivi elettronici embedded, le specifiche di progetto fanno tendere ad una scarsa protezione dei dispositivi, e quindi degli esecutori degli algoritmi, dalla rilevazione per mezzo di una campionatura di un segnale elettro-magnetico.

È possibile sfruttare tali segnali per raccogliere informazioni al fine di eseguire un attacco. I campioni dei segnali raccolti sono analizzati con metodi statistici e correlati attraverso il tempo di campionatura al clock e all'esecuzione delle istruzioni dell'implementazione dell'algoritmo, permettendo quindi di individuare i valori intermedi dell'algoritmo, calcolati a partire dalla chiave, e quindi parte della rappresentazione binaria della chiave, e in ultima istanza di recuperare interamente la chiave stessa.

Il livello di informazioni fornite da un *side channel* è, naturalmente, specifico sia del design che dell'implementazione dell'esecutore; per questo si possono distinguere *side channel* più o meno deboli, quindi con maggiore o minore perdita di informazione.

Riferendosi al principio di Kerckhoffs, le applicazioni in ambito embedded, o smart card della crittografia, si prestano perfettamente a rappresen-

tare quei sistemi per la comunicazione sicura che possono cadere in mano al *nemico*, senza che questo debba arrecare danno e quindi rappresentare un'inconvenienza.

1.2 Attacchi e Contromisure

La crittografia moderna permette attraverso le definizioni rispettivamente pubblica, di un algoritmo, e segreta, di una chiave, di ottenere riservatezza, integrità, autenticità.

Nella crittografia moderna un attacco ad un algoritmo crittografico ha lo scopo di individuarne la chiave, o informazione segreta, mentre storicamente gli attacchi più noti sono stati eseguiti sulla base di analisi dell'algoritmo stesso atte ad individuare debolezze di natura matematica. Nel caso dell'algoritmo DES si possono citare gli attacchi matematici [Biham and Shamir, 1991] e [Matsui, 1994].

Un attacco di tipo *side channel* ha lo scopo di recuperare il segreto ed è eseguito contro l'esecutore hardware dell'algoritmo crittografico, attraverso l'analisi dei segnali elettromagnetici e termici e della relativa correlazione tra i tempi degli stessi.

Questi segnali formano un canale secondario, *side channel*, da cui recuperare delle informazioni sullo stato di processazione dell'algoritmo crittografico, quindi anche, più o meno direttamente, la chiave segreta. Le informazioni di natura fisica raccolte vengono poi processate con un'analisi di natura statistico-probabilistica. Come vedremo nella sezione 2.1.2, uno degli attacchi basati su *side channel* più conosciuto è noto come *differential power analysis* [Kocher et al., 1998].

Una contromisura plausibile contro questi tipi di attacchi prevede di rendere altrettanto impraticabile quanto l'approccio brute force l'analisi dei campioni di dati raccolti.

Questo lavoro utilizza a tale scopo l'applicazione di contromisure di protezione di natura software per incrementare l'impegno e le risorse (*effort*) richieste per eseguire un attacco di tipo *side channel*, fornendo in questo mo-

do uno strumento per incrementare la sicurezza degli algoritmi crittografici, sia per le piattaforme progettate senza meccanismi di sicurezza hardware, sia per quelle già presenti sul mercato o derivanti da scelte di design prive di tali accorgimenti.

Le tecniche di protezione software si basano su approcci di tipo *masking* e *hiding* delle istruzioni eseguite o dei dati; una presentazione sintetica di questi concetti si trova in sezione 2.3. Le contromisure implementate sono state individuate per la loro generalità e indipendenza rispetto all'algoritmo da proteggere e, naturalmente, per l'efficacia.

1.3 Il framework implementato

Il framework implementato si compone: di un modulo in C per realizzare l'integrazione con il codice sorgente ed eseguire e applicare le contromisure, di un'estensione al compilatore LLVM per la modifica del codice oggetto da generare per realizzare l'integrazione e dell'introduzione in CLANG, il frontend di LLVM per il linguaggio C, di attributi per il linguaggio C implementati per l'integrazione.

È possibile suddividere le funzionalità del modulo C in: integrazione tra l'algoritmo da proteggere e i differenti moduli, applicazione delle contromisure morphing e shuffling, inizializzazione del sistema e costruzione delle strutture dati sull'algoritmo da proteggere.

All'interno del modulo di morphing si trova il codice corrispondente al polymorphic engine implementato in [Agosta et al., 2012], che si occupa della sostituzione casuale delle istruzioni presenti in un blocco con un possibile set di istruzioni semanticamente equivalenti.

Il modulo C comprende una serie di macro in C che generano variabili e funzioni utilizzate per l'integrazione con i sorgenti dell'algoritmo da proteggere. Questo compito è in parte svolto anche dagli attributi specifici, aggiunti al compilatore, che sono utilizzati per identificare le variabili e le funzioni interessate dal modulo nell'insieme.

CLANG e LLVM realizzano parte dei meccanismi di integrazione attraverso gli attributi, a disposizione dello sviluppatore per marcare il codice sorgente dell'algoritmo crittografico da proteggere; inoltre si occupano della raccolta di informazioni sul binario eseguibile, che sono usate sia per generare il codice specifico all'integrazione, sia per fornire al modulo di morphing e shuffling informazioni sul codice binario dell'algoritmo, informazioni che vengono raccolte durante la fase di compilazione.

Nello specifico i moduli di morphing e di shuffling intervengono durante l'esecuzione dell'algoritmo crittografico da proteggere, sostituendo al corpo eseguibile della funzione da proteggere una versione equivalente (morphing) e riorganizzando gli indici di iterazione delle operazioni su array per modificare la sequenza delle istruzioni e degli accessi in memoria, in lettura e scrittura; questi array nello specifico corrispondono alle variabili che contengono valori intermedi/stati dell'algoritmo crittografico.

L'implementazione è stata realizzata con supporto alla trasformazione dell'instruction set ARM, anch'essa mutuata da [Agosta et al., 2012] ed è stata estesa per gestire le istruzioni che fanno uso delle *constant pool*, presenti ad esempio nel set di istruzioni ARMv5 3.3.1.

Il framework è stato infine sottoposto a benchmark prestazionali su piattaforme ARMv5 e ARMv7, al fine di stabilire l'overhead del costo computazionale dell'implementazione.

1.4 Struttura del documento

Il prosieguo di questo documento presenta in sequenza stato dell'arte, implementazione e risultati.

Il capitolo 2 presenta lo stato dell'arte delle tecniche di contromisura e i lavori correlati già presenti in letteratura. Per completezza è stata inserita in questo capitolo una presentazione delle metodologie di attacco.

Il capitolo 3 descrive l'implementazione del framework realizzato, dalla struttura del modulo sorgente in C che realizza le contromisure, agli attributi che realizzano l'integrazione e alle estensioni di CLANG e LLVM. Vista la

peculiarità dell'implementazione, si è scelto di dare spazio ai dettagli tecnici realizzati specificatamente per ARMv5 per la gestione delle *constant pools*.

Il capitolo 4 sui risultati sperimentali comprende le informazioni specifiche sugli ambienti di esecuzione utilizzati per i test, le piattaforme ARMv5 e ARMv7, e quindi i dettagli sull'implementazione dell'eseguibile oggetto di analisi prestazionale.

All'interno del capitolo 5 è possibile trovare le conclusioni sui risultati ottenuti da questo lavoro ed un sintetico riferimento a possibili lavori incrementali e approcci futuri, individuati a partire dalla realizzazione di questo framework.

Capitolo 2

Stato dell'Arte

2.1 Attacchi Side Channel

La conoscenza del modus operandi della crittoanalisi differenziale e di quella lineare possono fornire dei metodi applicabili per la verifica e la validazione di un algoritmo crittografico; per loro natura, si basano sulle correlazioni statistiche tra gli ingressi e le uscite, affrontando solo la natura matematica dell'algoritmo. Gli attacchi *side channel* possono invece colpire, rendendole insicure, implementazioni di algoritmi che risultano resistenti alle tecniche di crittoanalisi.

Questa sezione copre gli attacchi di tipo power analysis, realizzati attraverso *side channel*, interessanti per i risultati che ne sono stati ottenuti, come vedremo successivamente.

Alla base di queste tecniche c'è l'assunzione che il consumo di un circuito dipenda linearmente dalla *Hamming Distance*, il numero di bit con cui differiscono due rappresentazioni binarie degli operandi, oppure dalla *Hamming Weight*, che corrisponde alla Hamming Distance dalla rappresentazione binaria dello 0.

2.1.1 Simple Power Analysis

La *simple power analysis* è una tecnica di analisi realizzabile con degli strumenti tecnologici alla portata di tutti. Nel lavoro [Kocher et al., 1998] viene presentato un attacco ad una smart card, campionata a 20Mhz mentre esegue DES a 16 round, attraverso l'utilizzo di hardware reperibile in un laboratorio di elettronica e di un multimetro in grado di campionare alle frequenze dell'esecutore.

Nel complesso del campione di segnali rilevati, si parla per la singola campionatura di *power trace*; queste sono rilevate e memorizzate durante l'esecuzione e analizzate con bassissime risorse computazionali, anche un semplice PC; per l'analisi può essere sufficiente un'ispezione visiva del segnale.

Durante una prima fase di elaborazione, le *power trace* sono associate al clock dell'esecutore hardware e ai singoli round dell'algoritmo. L'esempio citato [Kocher et al., 1998] contiene una *trace* che mostra 16 oscillazioni in potenza, ognuna corrispondente ai singoli round del DES.

Una ispezione più dettagliata di alcuni round permette di individuare i picchi di potenza per le singole operazioni eseguite, specificamente le operazioni che sono collegate a blocchi condizionali presenti nell'algoritmo. La *simple power analysis*, potendo fornire informazioni sul flusso di esecuzione eseguito dall'algoritmo, permette ad un attaccante di ricostruire il valore dei dati intermedi calcolati a partire dalla chiave.

Non solo il *datapath*, generato dai blocchi condizionali, ma anche le operazioni di calcolo, forniscono informazioni sui valori intermedi dell'algoritmo crittografico. Per quanto concerne l'implementazione del DES, oggetto di analisi dell'articolo [Kocher et al., 1999], sono state individuate come fonti di debolezza le istruzioni di salti condizionati, le moltiplicazioni e l'elevamento a potenza con modulo.

Dal punto di vista dell'esecutore fisico dell'algoritmo, sono state individuate come fonti di debolezza le CPU con implementazione a micro codice, perché, anche in caso di assenza di salti condizionati nell'algoritmo, presentano spesso sequenze di istruzioni e quindi consumi direttamente collegabili ai valori assunti dagli operandi.

2.1.2 Differential Power Analysis

La *differential power analysis* (DPA) è stata introdotta nel lavoro [Kocher et al., 1998] e, come la *simple power analysis*, ha come oggetto l'analisi del consumo energetico di un circuito al fine di recuperare informazioni sul segreto di un algoritmo crittografico.

Mentre la SPA svolge un'analisi delle *power trace* alla ricerca dei segnali corrispondenti a macro operazioni, come il cambio di datapath, la DPA punta ad un'analisi delle *power trace* alla ricerca di differenze nel consumo osservato al variare dei valori di input dell'algoritmo crittografico.

L'obiettivo della ricerca è infatti l'individuazione di informazioni, più o meno parziali, riguardanti i valori specificatamente assunti, dalle variabili o dagli intermedi, durante l'esecuzione dell'algoritmo crittografico.

2.1.3 DES e DPA

Il DES è un cifrario a blocchi che opera su blocchi di testo in chiaro di 64 bit con una chiave di 56 bit (8 byte con 1 bit di parità per byte), eseguendo una permutazione iniziale, 16 *round* e una permutazione finale.

La permutazione iniziale del testo in chiaro lascia il testo nella forma $m_0 = L_0\dot{R}_0$, dove m è il testo ed L e R sono rispettivamente le parti sinistra e destra di 32bit ciascuna. Alla permutazione seguono, quindi, i 16 round definiti nel modo seguente:

$$L_i = R_{i-1} \tag{2.1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \tag{2.2}$$

Il blocco $f(R_{i-1}, K_i)$ corrisponde alla funzione di *Feistel*, che opera una operazione non lineare, di seguito descritta, con gli operandi R_{i-1} e K_i, una parte di 48bit della chiave.

Nel blocco $f(R_{i-1}, K_i)$ si procede con un'espansione nota da 32bit a 48bit di $R \rightarrow E(R)$; il risultato della successiva operazione $E(R) \oplus K_i$ viene suddiviso in blocchi di 6 bit $B_{i|1\ldots8}$.

Il generico B_i è uno degli 8 input della S-box che esegue la trasformazione non lineare e che produce come output $C_{i...8}$, costituiti ciascuno da 4 bit concatenati a formare la stringa di 32 bit che verrà sommata a L_{i-1} (2.1).

Nel lavoro [Kocher et al., 1999] è stato realizzato un attacco all'algoritmo DES; l'attaccante osserva m operazioni di cifratura e ne memorizza i segnali T rilevati (di k campioni) e il testo cifrato C:

$$T_{1...m}[1...k]$$
$$C_{1...m}$$

Per questo attacco non serve la conoscenza dei messaggi in chiaro.

Per la stima è stato usato un modello probabilistico $D(C, b, K_s)$, che stima il valore dei 32 bit $b_{1...32}$ dell'intermedio L all'inizio del sedicesimo round, con il testo cifrato C e i 6 bit della chiave $K_{s|1...8}$ che entrano nella S-box in corrispondenza del bit b.

Durante la verifica della stima, la funzione di selezione viene usata per dividere il campione P di m *power trace* in base alla stima fornita da $D(C, b, K_s)$ per ognuno dei K_s. A questo punto viene calcolata la media di ognuno dei due sottoinsiemi e, dalle due medie, la differenza $\Delta_D[j|1...k]$.

Qualora la stima fosse corretta, il grafico della differenza presenterebbe un picco per l'operazione attaccata. Se la stima fosse completamente non correlata, allora la funzione di selezione si comporterebbe in modo del tutto casuale durante la divisione dell'insieme dei campioni di *power trace*; le due medie risulterebbero di conseguenza identiche e la loro differenza nulla, comportamento raggiunto al limite con $m \to \infty$.

In figura 2.1 è riportato un grafico presente in [Kocher et al., 1999]. Sono visibili il grafico di una *power trace*, il grafico della deviazione standard delle misurazioni e per ultimo il grafico differenziale, ottenuto con $m = 10^4$ e corrispondente alla stima effettuata all'istante di clock 6.

Il grafico differenziale mostra per gli intervalli non correlati al bit da stimare una caratteristica inferiore agli intervalli interessati dal bit, per più di un ordine di grandezza.

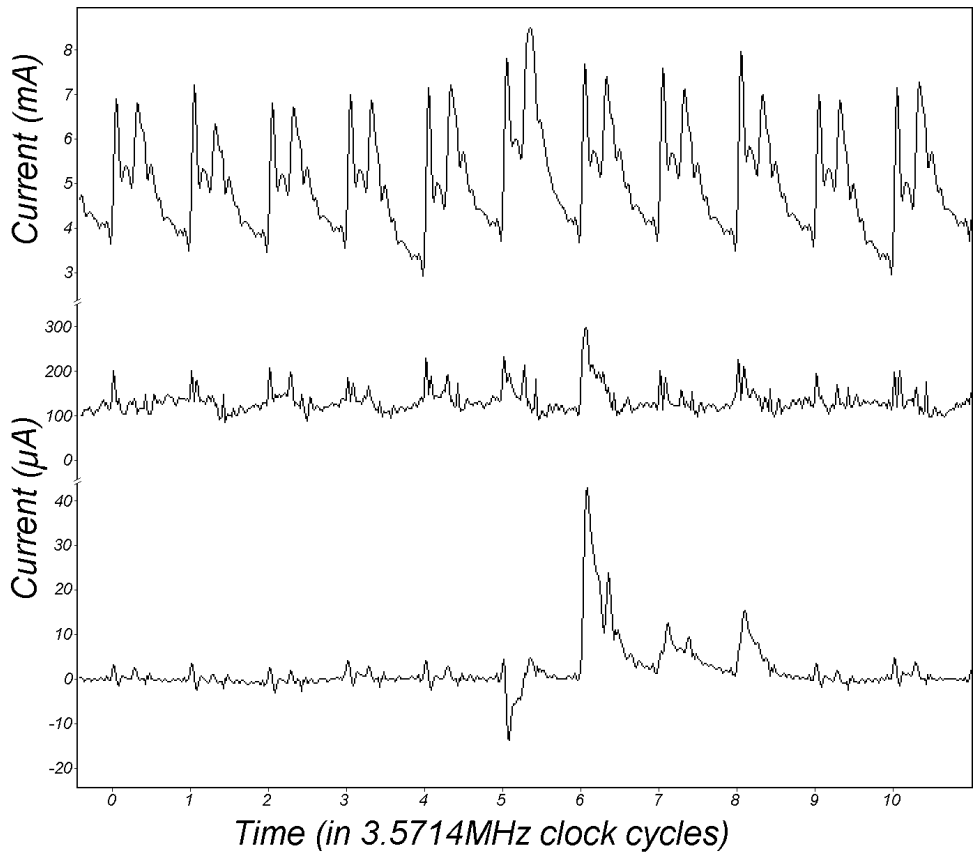

Figura 2.1: in alto *power trace*, al centro deviazione standard, in basso differenziale delle *power trace*

2.1.4 DPA del secondo ordine

Introduciamo ora una versione avanzata di DPA; a tale scopo anticiperemo la presentazione di una tecnica di contromisura specifica per DPA presentata in 2.3.

Il random masking è una tecnica di protezione da un attacco DPA che consiste nella trasformazione degli operandi e delle operazioni, eseguita per proteggere i valori intermedi calcolati a partire dalla chiave; ad esempio viene eseguita su un valore intermedio r un'operazione di XOR con un valore

random, $r \oplus w$. L'equivalenza semantica delle operazioni sui valori intermedi viene garantita dall'aggiunta/modifica delle operazioni che hanno il compito, ad esempio, di estrarre i bit significativi. Attraverso l'uso di questa tecnica è possibile alterare il consumo fatto registrare da un'operazione in funzione dei valori dei suoi operandi; quindi, tornando al grafico delle differenze delle DPA, all'instante specifico dell'operazione non sarà più presente il picco di potenza atteso.

In letteratura è possibile individuare dei metodi di analisi differenziale più complessi, in grado di far fronte al tipo di contromisura appena presentata, grazie all'utilizzo di un'analisi differenziale che viene definita del secondo ordine o semplicemente di ordine maggiore.

In un attacco DPA del primo ordine, l'attaccante correla i valori di consumo tra due soli istanti del medesimo campione di *power trace*; in un attacco del DPA del secondo ordine, l'attaccante analizza un intervallo τ_1, τ_2 durante il quale sa che il flusso di operazioni manipolerà il valore intermedio r e la sua trasformazione $r \oplus w$.

- $I(x, s)$ è il valore intermedio calcolato dall'input x e da una parte della chiave s
- $\mathfrak{S}_{b=[0,1]}(\hat{s})$ sono i due insiemi di *power trace* per i quali si suppone che il bit b valga 0 o 1
- $\mathscr{C}(t)$ è la potenza consumata dal device nell'istante t
- $\langle\rangle_x$ è l'operazione di media sugli elementi x di un generico insieme

La definizione della differenza tra i due insiemi in cui si ripartono le *power trace* per stimare il valore $\hat{s}$ diventa:

$$\overline{\Delta_2}(\hat{s}) = \langle|\mathscr{C}(\tau_2) - \mathscr{C}(\tau_1)|\rangle_{x \in \mathfrak{S}_1(\hat{s})} - \langle|\mathscr{C}(\tau_2) - \mathscr{C}(\tau_1)|\rangle_{x \in \mathfrak{S}_0(\hat{s})}$$

da cui deriva che il valore di $\hat{s}$ per il quale $\overline{\Delta_2}$ è massima in valore assoluto è il valore che si avvicina più verosimilmente al valore s di parte della chiave.

Nel lavoro [Joye et al., 2005] è presentato un attacco contro l'algoritmo RC6 atto a stimare il numero di *power trace* da raccogliere necessarie per completare un attacco di DPA del secondo ordine, considerando come fattore dell'analisi anche la variazione con dimensioni maggiori della chiave.

L'algoritmo è stato scelto dall'autore per la capacità di essere configurabile in numero di round, oltre che naturalmente in lunghezza della chiave.

Dai risultati sperimentali si evince la necessità di raccogliere un ampio campione di *power trace* per poter completare un attacco; si parla di raccogliere un campione di *power trace* rispettivamente di 3000 elementi per una lunghezza di parola di 8 bit e di 20000 elementi per l'esecuzione dell'algoritmo con una chiave di lunghezza 16 bit; maggiori dettagli sono disponibili nella Sezione 5 Experimental Results and Observations di [Joye et al., 2005].

Nel lavoro [Oswald et al., 2006] viene presentato un attacco DPA del secondo ordine, svolto contro un esecutore dell'algoritmo AES di tipo SmartCard a 8 bit. L'attacco è stato realizzato contro un'implementazione software facente uso della tecnica di contrattacco *additive masking*, sufficiente ad ostacolare un attacco DPA del primo ordine.

In questo lavoro sono stati affrontati tre quesiti fondamentali:

- identificare in una *power trace* i punti utili a ricostruire le informazioni
- identificare quante *power trace* sono necessarie per la riuscita di un attacco
- determinare se è possibile diminuire il numero di *power trace* necessarie per completare un attacco, cambiando la funzione di processazione e analisi delle textitpower trace

Dopo la raccolta di 3000 *power trace*, si fa una stima dell'intervallo della *power trace* di interesse, quindi si passa alla fase di pre-processazione dell'intervallo I individuato tra i punti P_1 e P_2, stimati costituire l'intervallo di inizio e fine della processazione dell'algoritmo AES.

La pre-processazione consiste nella costruzione, a partire dai punti dell'intervallo, di una sequenza di segmenti corrispondenti all'operazione $|P_a - P_b|$. Gli elementi della sequenza vengono poi concatenati a formare una nuova *power trace* derivata, che sarà oggetto di un attacco DPA standard. Per costruzione, ognuno dei punti dei segmenti della sequenza corrisponde al modulo della differenza di due punti della *power trace* originale; $|P_a - P_b|$ è equivalente all'inverso $|P_a - P_b|$, per questo la lunghezza della *power trace* derivata corrisponde a $\frac{l/(l-1)}{2}$ e quindi cresce con il quadrato della lunghezza dell'intervallo considerato.

L'attacco è risultato praticabile con approssimativamente 460 *power trace* di lunghezza pari a $\frac{l/(l-1)}{2}$, dove l è la lunghezza dell'intervallo stimato.

Nel lavoro citato vengono anche analizzate nel dettaglio le operazioni e le procedure per attaccare le singole fasi dell'algoritmo AES sottoposto a masking; nella sua appendice è presente uno schema a blocchi dell'AES con masking.

2.1.5 DPA e Debolezze in letteratura

Per un'introduzione e un approfondimento sugli attacchi DPA del primo ordine, sono disponibili in letteratura i lavori [Kocher et al., 1998] e [Kocher et al., 1999].

Per quanto riguarda gli attacchi DPA del secondo ordine sono disponibili in letteratura: [Joye et al., 2005], che presenta un'analisi matematica della stima, [Oswald et al., 2006], che offre un'analisi del costo di un attacco DPA del secondo ordine e [Prouff et al., 2009] che presenta una analisi statistica.

Esempi di attacchi del secondo ordine verso esecutori di tipo SmartCard sono presentati in [Messerges et al., 1999], [Quisquater and Samyde, 2001] e [Oswald et al., 2006].

La molteplicità di attacchi DPA presenti in letteratura, eseguiti nei confronti di numerose architetture, spinge a raccogliere informazioni su quali tipologie di istruzioni di una CPU siano fonte di maggiori debolezze in caso di un attacco di tipo DPA. Naturalmente non è compito di questo lavoro voler tracciare la linea di demarcazione o correlazione, se esiste, tra tipologia di istruzione e sua implementazione fisica per un determinato esecutore hardware. Quello che segue, quindi, è solo un elenco di fonti e delle relative debolezze riscontrate; per un'analisi approfondita si rimanda al relativo lavoro citato.

In [Bayrak et al., 2011] è descritto un attacco all'algoritmo AES in esecuzione su un micro-controllore *8-bit AVR*, realizzato con lo scopo di riscontrare le istruzioni più sensibili ad un attacco DPA e successivamente applicare solo per queste una metodologia di contromisura. Per questo micro-controllore sono risultate essere particolarmente suscettibili ad un attacco DPA le istru-

zioni che si occupano del trasferimento di dati da e verso la memoria, ovvero genericamente le istruzioni di load and store, con una maggiore suscettibilità per le istruzioni di load rispetto a quelle di store.

In [Agosta et al., 2012] viene illustrato un attacco DPA realizzato contro un esecutore ARM926 dell'algoritmo AES; in questo caso risultano essere ancora una volta le istruzioni di load and store della CPU ARM la fonte maggiore di informazioni per un attacco DPA. Come si può leggere in 2.3.4, l'operazione scelta per l'attacco è uno *xor* dell'AddRoundKey, che ha la peculiarità di risiedere solo nei registri, perchè, come viene illustrato nel lavoro, le solite operazioni di load presentavano un comportamento variabile come conseguenza delle interferenze dovute alla presenza della cache dati.

In [Barenghi et al., 2010] viene realizzato un attacco DPA del primo ordine nei confronti dell'algoritmo AES in esecuzione su una CPU Cortex-M3 a 32 bit, senza protezione hardware o software, e utilizzando come obiettivo dell'attacco l'output dell'operazione SubBytes.

2.2 Filtraggio di segnale e DPA

In [Barenghi et al., 2010] viene presentata una metodologia di filtraggio durante il campionamento delle *power trace* utilizzata dagli autori per migliorare l'effort necessario alla computazione, sia diminuendo la complessità dell'analisi riducendo la numerosità di campioni raccolti per la *power trace* corrispondente all'esecuzione dell'algoritmo crittografico, sia

L'applicazione di questa tecnica ha come effetto principale la riduzione dell'effort necessario a raccogliere il campione di *power trace* minimo per eseguire un attacco DPA del primo ordine.

La riduzione in numero e dimensione delle *power trace* da raccogliere è stimata essere di un ordine di grandezza per quanto riguarda il solo numero di *power trace* necessarie a compiere l'attacco DPA e recuperare la chiave e di 2.5 ordini di grandezza in termini di spazio occupato dalla *power trace*.

La riduzione dell'effort generale permette di applicare la tecnica di DPA a dispositivi che prima risultavano computazionalmente sicuri.

2.2.1 Implementazione Attacco

Per la progettazione del filtro passa banda si è reso necessario l'impiego di un filtro non rettangolare, perché la corrispondente rappresentazione nel dominio del tempo, *aliased sinc function*, avrebbe introdotto dei picchi fabbricati nel segnale risultante.

La forma del filtro più adatta è stata quindi riconosciuta essere quella del filtro di Chebycev. Utilizzando tale filtro, è infatti possibile controllare l'intensità delle *side lobes* generate dall'introduzione del filtro e quindi limitarne l'effetto. L'implementazione presentata nell'articolo citato fa uso di un filtro con de-amplificazione per i *side lobes* pari a $120db$, che li rende della stessa intensità dell'errore numerico introdotto dalle fasi di elaborazione dell'algoritmo dell'attacco DPA, e quindi trascurabili.

L'attacco è stato eseguito contro un'implementazione software 32bit dell'AES, in esecuzione su una CPU ARM Cortex M3 che implementa ISA ARM-7M, molto comune e impiegata soprattutto nei dispositivi low power. Con il solo filtraggio è stato possibile ridurre da 6000 a 480 le *power trace* da immagazzinare e analizzare per effettuare l'attacco.

La riduzione di risorse permette anche di rendere realizzabile un attacco DPA contro dispositivi che presentano dei *side channel* con bassa fuga di informazioni, perché questa tecnica permette di compensare memorizzando maggiori quantità di dati.

Questi risultati mostrano la necessità di applicare delle contromisure software per compensare le debolezze mostrate. Come vedremo nella sezione successiva, in [Oswald et al., 2006] e in [Agosta et al., 2012] sono presenti approcci all'applicazione automatica di contromisure software contro gli attacchi DPA.

2.3 Contromisure Software

Per quei dispositivi che fanno uso di algoritmi crittografici ma che permettono di raccogliere informazioni sul segreto essendo privi, quindi, di

contromisure realizzate in hardware, esiste la possibilità di applicare delle contromisure via software.

Queste contromisure sono utili al fine di rendere impraticabile o impossibile la rilevazione e/o analisi delle power trace per ricostruire il segreto.

Alla base delle motivazioni per l'impiego di contromisure software c'è l'idea di ottenere un'implementazione più sicura, che riduca la perdita di informazioni dai *side channel*, a partire dalla trasformazione nell'algoritmo da proteggere di uno o più tra: i) istruzioni da eseguire ii) valori degli operandi iii) flusso delle istruzioni

È possibile individuare in letteratura, nel libro [Mangard et al., 2007], un'analisi delle contromisure software efficaci contro attacchi che sfruttano *side channel* come DPA e SPA. Il libro citato identifica e distingue le tecniche in hiding e masking e per ognuna di esse propone una analisi della relativa implementazione sia in software che in hardware, quando possibile; l'analisi comprende i pro e i contro delle singole tecniche.

L'obiettivo di ogni contromisura, sia questa di tipo hiding o masking, è quello di rendere il consumo di un device indipendente dai valori intermedi della chiave o dalle operazioni eseguite.

2.3.1 Hiding

Per raggiungere l'obiettivo di nascondere il reale consumo del device, una tecnica di tipo hiding opera in modo da rendere il rapporto segnale rumore minimo.

Per realizzare questo, è possibile durante la progettazione costruire il device affinché consumi un determinato quantitativo di potenza per ognuna delle istruzioni, oppure affinché venga consumato un quantitativo completamente random, ad esempio eseguendo le istruzioni non strettamente in ordine di *issue*. Questi approcci modificano l'ampiezza del segnale e del rumore, oppure spostano i consumi nel tempo.

Gli approcci software per la modifica del tempo d'esecuzione delle istruzioni sono l'inserzione casuale di operazioni inutili e il riordino delle operazioni.

Random insertion

Per l'inserzione random delle operazioni, si distinguono tre possibili posizioni relative al codice dell'algoritmo crittografico: prima, durante e dopo. Sempre in [Mangard et al., 2007], si sottolinea l'importanza di eseguire l'inserzione di un numero totale, costante, di istruzioni, in modo da rendere impossibile la stima del numero di istruzioni random inserite misurando il tempo di esecuzione. Questa tecnica ha come principale svantaggio la diminuzione delle performance dovuta all'aggiunta di istruzioni inutili.

Shuffling

Partendo dall'idea di alterare la sequenza temporale con cui vengono eseguite le operazioni dell'algoritmo crittografico, è possibile applicare tale approccio a quelle parti dell'algoritmo che possono essere eseguite in un ordine qualsiasi, tipicamente codice che potrebbe essere eseguito anche in parallelo. Per quanto riguarda l'algoritmo AES, ad esempio, è facile individuare nel blocco delle 16 lookup della SBox la sequenza di operazioni che è possibile eseguire in un ordine qualsiasi.

La tecnica di shuffling, per quanto riguarda il throughput dell'algoritmo, rappresenta rispetto al *random insertion* un miglioramento, ma presenta uno svantaggio sostanziale; non è infatti applicabile in qualsiasi contesto, in quanto richiede delle sezioni peculiari dell'algoritmo che si prestano a tale approccio.

Noise Increase

Per aumentare invece il rumore prodotto dalla CPU, si possono eseguire più istruzioni in parallelo, nel caso di una CPU super scalare, oppure eseguire le istruzioni con un datapath maggiore. Nel libro cui abbiamo già fatto riferimento è proposto anche l'impiego di altri sistemi in parallelo, come coprocessori o interfacce di comunicazione. Ancora, tra le soluzioni hardware proposte che l'autore ha trovato più interessanti, vi è quella di cambiare il clock della CPU del sistema secondo un pattern random; ciò venne proposto

ai tempi di redazione del libro come soluzione hardware, ma, per la rapida diffusione della gestione del consumo nei dispositivi elettronici avvenuta in questi anni, è ora estendibile e applicabile via software.

2.3.2 Masking

Con l'obiettivo di nascondere i valori reali degli operandi dell'algoritmo crittografico (direttamente collegati alla chiave), le contromisure di tipo masking randomizzano il valore degli operandi, introducendo delle operazioni aggiuntive/alternative che li trasformano.

La randomizzazione si ottiene attraverso l'applicazione di un'operazione $*$, algebrica o booleana, al valore intermedio v con un valore random m:

$$v_m = v * m$$

Le operazioni più comunemente utilizzate sono $\oplus$, lo xor booleano, e la somma o il prodotto modulare, con il modulo definito specificatamente per l'algoritmo:

$$\begin{aligned} v_m &= v * m \\ v_m &= v + m \ (mod\ n) \\ v_m &= v \times m \ (mod\ n) \end{aligned}$$

Compatibilmente con l'operazione scelta, saranno modificati i passi successivi dell'algoritmo crittografico al fine di mantenere la sua semantica.

Questa modifica dell'algoritmo crittografico può risultare complessa, perché può richiedere di inserire una quantità considerevole di operazioni aggiuntive; oltretutto un algoritmo crittografico fa spesso uso di entrambi i tipi di operazioni, algebriche e booleane.

Gli algoritmi crittografici, utilizzando anche blocchi di operazioni non lineari, come la SBox dell'AES, impongono una cura maggiore; ad esempio, nel caso della SBox, che risulta essere compatibile con un masking basato

sulla moltiplicazione, ci si espone ad una debolezza.

$$f(a * b) = f(a) * f(b)$$
$$S(a * b) \neq S(a) * S(b)$$

Si può leggere in [Mangard et al., 2007] che la moltiplicazione lascia invariato l'elemento neutro, genericamente lo 0, lasciando quindi l'implementazione, nel caso si presenti il valore 0, esposta ad un attacco *side channel*.

Masking Table lookup

Genericamente, un'operazione non lineare come quella realizzata dalla SBox prevede l'utilizzo di una tabella, detta di lookup, che contiene per ogni input il corrispettivo output; l'algoritmo accede per ovvie ragioni di performance a questa tabella memorizzata in RAM.

Realizzare il masking di una SBox richiede la trasformazione della tabella di lookup (T) in modo che mantenga l'operazione lineare:

$$T_m(v \oplus m) = T(v) \oplus m$$

Generare questa tabella, passando attraverso tutti gli input v, è un processo semplice, ma va eseguito anche per ogni valore m usato per il mascheramento, facendo di conseguenza aumentare la richiesta, sia in termini computazionali che di spazio di memoria.

2.3.3 Contromisura: Random Pre Charging

Per le implementazioni che hanno un modello dei consumi che segue linearmente la distanza di hamming tra i due operandi trasmessi su un bus dati o memorizzati nel medesimo registro, è possibile mascherare il consumo mostrato dal sistema precaricando con un valore random il registro o il bus dati; in questo modo la potenza assorbita HD dal sistema sarà dipendente dal valore e dalla maschera.

$$HD(v, m) = HW(h \oplus m)$$

L'operazione di xor logico identifica i bit differenti tra le due rappresentazioni, HW calcola la potenza assorbita in base ai bit della rappresentazione. La potenza assorbita dal device all'istante di clock corrispondente all'operazione suscettibile ad un attacco *side channel* sarà dipendente dal valore di masking m e quindi mascherata. Secondo [Mangard et al., 2007], occorre prestare attenzione a non riutilizzare il valore usato per maschera per due random precharging eseguiti in successione, perché questo renderebbe suscettibile all'attacco la somma dei due valori intermedi.

$HD(v_m, w_m) = HW(v_m \oplus w_m) = HW(v \oplus m \oplus w \oplus m) = HW(v \oplus w)$

In Letteratura

Questa tecnica viene utilizzata in [Bayrak et al., 2011] come contromisura per mascherare le istruzioni. Il mascheramento dei consumi viene realizzato attraverso l'aggiunta di operazioni casuali e inutili prima delle istruzioni identificate.

Il principio fisico alla base di questa tecnica consiste nel fatto che il consumo dipenda dalla distanza di hamming tra i bit dei vari registri o bus dati della CPU coinvolti nelle operazioni. La realizzazione software di questa tecnica è discussa in [Tillich and Großschädl, 2007], ma l'implementazione in [Bayrak et al., 2011] si differenzia per la presentazione e l'utilizzo della tecnica di protezione *random precharging* prima delle istruzioni da proteggere, avendole identificate eseguendo inizialmente un test di *leakage* al fine di stabilire le singole istruzioni più suscettibili.

2.3.4 State of Art Polymorphic Engine

In [Agosta et al., 2012] è presentata l'implementazione di un framework generico per l'applicazione di contromisure software per la protezione da attacchi *side channel*.

Il framework proposto opera a *compile time* e *runtime* e comprende tre moduli che operano sul codice originario dell'algoritmo da proteggere: *code morphing*, *rescheduling* e *array access permutation*.

Questo framework, come vedremo, risulta essere trasparente dal punto di vista dell'algoritmo da proteggere, essendo questo un insieme di moduli del tutto generici.

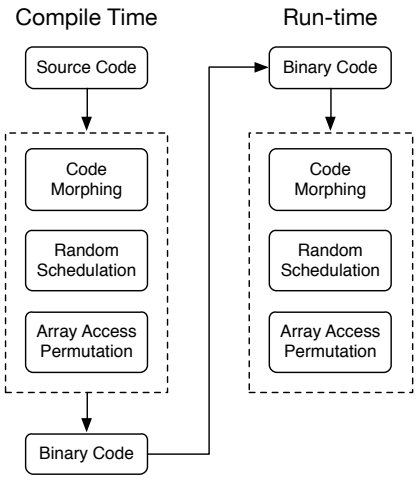

Figura 2.2: Grafico che illustra il polymorphic engine implementato in [Agosta et al., 2012]

Rescheduling

Il modulo di rescheduling opera una trasformazione della sequenza delle istruzioni, preservando la dipendenza delle stesse, in modo del tutto equivalente a quello che svolge un compilatore nello scheduling delle istruzioni. Il modulo di schedulazione non applica, come fa un compilatore, il vincolo della minimizzazione della latenza o della potenza consumata, ma costruisce una sequenza di operazioni del tutto casuale. Le differenti sequenze di istruzioni, generate a runtime dal modulo, realizzano una contromisura di tipo hiding equivalente ad una di tipo shuffling su una piccola sequenza di istruzioni all'interno di un blocco, impedendo l'allineamento temporale tra successive esecuzioni.

Array Access Permutation

Il modulo che realizza questa tecnica trasforma la sequenza di accesso ad un array; nel caso dell'AES, questa tecnica è applicabile alla sequenza di accessi alla LookupTable; variando quindi gli indici con sono eseguite le operazioni della SBox, si realizza lo shuffling descritto in Sezione 2.3.

Code Morphing

Il morphing è una contromisura di tipo hiding; essa applica infatti una modifica a runtime del codice binario, alterando le istruzioni dell'algoritmo del flusso di esecuzione e rimpiazzandole con istruzioni equivalenti.

Esso consiste nell'alterazione della sequenza di istruzioni in un blocco di codice, con l'esclusione delle istruzioni di tipo super-visor call, IO e MMU. Opera una sostituzione di un'istruzione, o di un insieme di queste, con una sequenza di istruzioni differenti ma scelte con il fine di preservare la semantica dell'operazione originale.

Il morphing è una tecnica che si utilizza come contromisura nel campo della sicurezza contro attacchi di tipo power, perché permette di introdurre del rumore di fondo durante gli attacchi di tipo *side channel*.

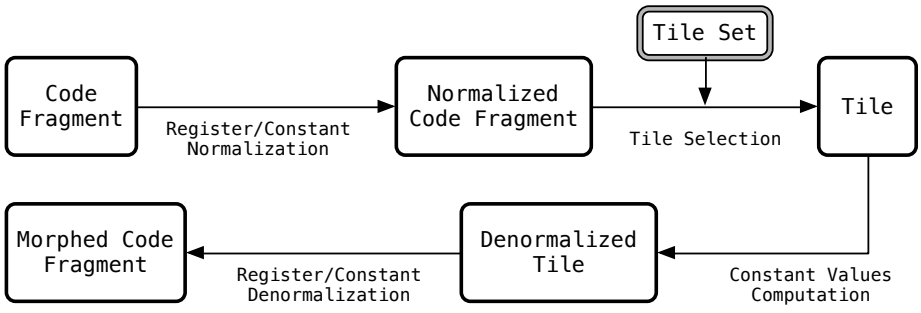

Figura 2.3: Grafico che illustra il i passi del code morphing come implementato in [Agosta et al., 2012]

Algoritmo di sostituzione

L'algoritmo di sostituzione implementato utilizza un approccio generico identificando dei *tile set* o insiemi di istruzioni equivalenti. I *tile* equivalenti sono rappresentati da *pattern/template* di istruzioni alternative, generalizzando quindi dai registri utilizzati da queste ultime; differiscono ad esempio, oltre che per le istruzioni, anche per i valori immediati utilizzati.

Gli autori forniscono anche una dimostrazione teorica dell'equivalenza dell'algoritmo di sostituzione, sotto forma di equivalenza semantica di due frammenti di codice contenenti istruzioni, a valle e a monte della trasformazione.

Con tale approccio, l'implementazione risulta del tutto indipendente dall'algoritmo da modificare, ma, dovendo trattare le istruzioni, i *tile set* risultano essere dipendenti e realizzati specificatamente per l'*instruction set* dell'architettura da proteggere.

La verifica sperimentale, presente in [Agosta et al., 2012], ha portato alla realizzazione dell'engine per *instruction set* ARMv5; la CPU scelta per l'esecuzione del framework e sottoposta alla verifica è un ARM926.

Dall'analisi eseguita, le istruzioni più sensibili sono risultate essere le istruzioni di load and store dei valori intermedi calcolati a partire dalla chiave. La specifica implementazione hardware si è rivelata essere fonte di poche informazioni per l'istruzione di load, a causa della presenza e dell'utilizzo della cache dati, che introduceva un comportamento irregolare per le rispettive *power trace*.

Capitolo 3

Implementazione

Nel capitolo 2 sono state introdotte le tecniche di Morphing e Shuffling nel campo delle contromisure per gli attacchi crittografici.

Nel presente capitolo vengono affrontate le specifiche tecniche impiegate per la realizzazione del framework, che permette l'applicazione delle contromisure per gli attacchi crittografici.

Vengono inoltre affrontati i dettagli dell'implementazione del sistema, a partire dal design sino alla realizzazione dei vari moduli e della loro interazione.

3.1 Specifiche e Design

L'obiettivo iniziale di questo lavoro era la realizzazione di un unico modulo, integrato all'interno del compilatore, che fosse in grado di eseguire autonomamente tutte le tecniche relative alle contromisure per gli attacchi crittografici.

In fase di sviluppo, questa soluzione è stata sostituita da un approccio più esplorativo e meno integrato, costituito da moduli distinti, modificabili e parametrizzabili. Tale approccio è risultato più adatto alla fase esplorativa e di testing e per questo è stato infatti possibile verificare e affrontare alcune problematiche presentatesi durante la fase di sviluppo. Vantaggi e problematiche affrontate sono approfonditi nella sezione 3.3.3.

3.1.1 Specifiche

Il fine ultimo del lavoro è realizzare uno strumento alla portata dello sviluppatore per l'applicazione di contromisure agli attacchi side-channel; si è scelto quindi di realizzare un framework per l'applicazione contemporanea del morphing e dello shuffling.

Il framework deve essere applicabile al codice sorgente in linguaggio C ed essere il meno invasivo possibile nella sua inserzione nel codice di partenza.

Una volta che framework e codice sorgente vengono compilati, il framework risulta integrato all'interno del binario eseguibile del codice originale; per questo motivo, è conveniente avere a disposizione dei parametri che permettano di attivare il framework e controllare singolarmente l'esecuzione dei moduli che implementano le diverse funzionalità. È stato minimizzato l'impatto dell'esecuzione dei moduli con il fine di eseguire un'analisi tramite benchmark e di valutare il costo in termini di performance e delay del sistema.

3.1.2 Design e Use Case

L'integrazione del framework all'interno di una libreria o di un'applicazione che voglia fare uso delle funzionalità di sicurezza interessa tutto il processo di sviluppo e di rilascio. Con ciò si intende che lo sviluppatore deve interagire con il framework durante la fase di realizzazione del codice sorgente dell'applicazione, ma deve anche operare per l'integrazione durante la fase di compilazione e linking dello stesso progetto.

In un sistema software che utilizza il framework, è possibile distinguere tra la funzione da proteggere, il framework e il resto del codice sorgente, cui spetta il compito di utilizzare la funzione, inizializzare e terminare il programma.

Scendendo maggiormente nel dettaglio, tale struttura modulare non rappresenta un sistema "a camere stagne", in quanto è necessario preparare una sezione di codice per la configurazione del framework, che interessa sia il codice per l'inizializzazione che quello per la finalizzazione del framework.

Il codice di inizializzazione e finalizzazione condividono con il codice di configurazione un insieme di funzioni, strutture dati e variabili, che spetta allo sviluppatore personalizzare in base alla sua applicazione e alle contromisure da applicare alla sezione di codice da proteggere.

Per il design si è fatto riferimento al seguente esempio d'uso e agli obiettivi definiti nella sezione precedente.

L'esempio d'uso è stato articolato in quattro fasi:

i) **identificazione** identificare la funzione e i suoi usi
ii) **inizializzazione** identificare nel flusso del programma i punti in cui inserire il codice di inizializzazione e finalizzazione del framework
iii) **integrazione** aggiungere al codice originale gli attributi e le variabili di supporto
iv) **compilazione** aggiungere un'unità di compilazione che sostituisce quella originale

Segue una breve illustrazione delle quattro fasi dell'esempio d'uso.

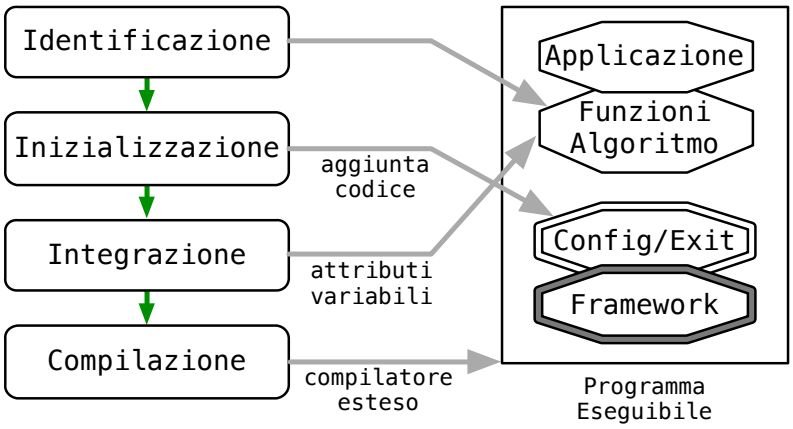

Figura 3.1: Schema che riassume le fasi dell'integrazione del tool con uno base nota di cd

Identificazione

Nella fase iniziale di inserzione all'interno di un progetto, lo sviluppatore deve selezionare la funzione da proteggere e la relativa unità di compilazione, per la quale vuole siano attive le contromisure.

A questo punto, lo sviluppatore deve identificare i punti dove inserire le chiamate all'inizializzazione e alla finalizzazione del framework. La finalizzazione libera le le risorse allocate dal framework per la gestione di strutture dati dinamiche, per questo è consigliato non omettere la chiamata alla funzione di finalizzazione del framework. Qualora fosse impossibile determinare il punto di ritorno è conveniente ricorrere alle funzioni di libreria *at_ exit* che registra una funzione da chiamare in finalizzazione. L'inizializzazione del framework viene realizzata in quella sezione del codice che fa parte del blocco di configurazione.

Inizializzazione

La funzione di inizializzazione del framework per come è costruita è una macro che serve a costruire un record per la funzione da proteggere contenente i parametri che regolano il framework nell'applicazione delle contromisure.

Internamente al framework e durante l'inizializzazione, vengono svolte l'allocazione delle strutture dati e vengono eseguiti gli algoritmi di analisi del codice binario della funzione, per costruirne una rappresentazione in termini di *basic blocks* e *literal pools*.

Per questo la chiamata alla funzione restituisce una struttura dati contenente le informazioni specificate e raccolte dal codice di inizializzazione; questo oggetto è unico per ogni singola funzione da proteggere e si comporta come un descrittore della funzione da proteggere all'interno del framework

Il puntatore all'oggetto restituito dalla funzione di inizializzazione, oltre ad essere usato internamente dal framework è da utilizzare esplicitamente nella chiamata al codice di finalizzazione del framework, contendo al suo interno strutture dati allocate dinamicamente, da finalizzare.

Integrazione

Per completare l'integrazione, lo sviluppatore aggiunge al proprio codice gli header del framework e, attraverso delle macro in linguaggio C in esso disponibili, maschera il simbolo della funzione. L'attivazione e l'utilizzo dei singoli moduli sono ottenuti attraverso l'impiego di attributi del compilatore, che lo sviluppatore aggiungerà contestualmente alla funzione da proteggere.

Compilazione

La compilazione del progetto prevede a questo punto l'aggiunta dei sorgenti del framework nello script di compilazione. L'unità di compilazione della funzione da proteggere sarà modificata dal framework per poter realizzare il mascheramento del simbolo. Di seguito nel listato 3.1 è mostrato un esempio, illustrativo, della definizione di una funzione di un meccanismo di protezione per la funzione **AES_encrypt**, che verrà protetta e controllata dal framework. Nel concreto il framework, come vedremo successivamente in 3.2.4, utilizza delle macro per la generazione di una funzione analoga al **main_morpher** presentato.

Listing 3.1: Template della dichiarazione di una funzione da proteggere

```
__attribute__((MorpherSubstitute(AES_encrypt)))
void main_morpher(const unsigned char* in, unsigned char *out, const
    AES_KEY *key) {


}
```

3.2 Elementi del framework

Il framework è stato implementato componendo insieme funzionalità e macro scritte in codice sorgente e realizzando delle estensioni del toolchain di compilazione C/C++ per Clang/LLVM.

L'obbiettivo originale di avere a disposizione il framework come sola estensione del compilatore si è trasformato in una soluzione mista, per andare incontro alle esigenze di facile modificabilità del sistema; tale soluzione risulta più adatta alla realizzazione esplorativa. In sostanza è stata mantenuta semplice una soluzione altrimenti strutturalmente complessa ed è stata resa modificabile una soluzione altrimenti legata alla ricompilazione del compilatore.

3.2.1 CLANG e LLVM

Clang è il frontend per il C/C++ per il compilatore LLVM ed è stato scelto essendo una soluzione industriale e dall'architettura modulare, caratterizzata da una definizione testuale del linguaggio intermedio comoda per lo sviluppo e la verifica. Clang e LLVM sono stati modificati per svolgere, sia delle modifiche al codice generato, sia la raccolta di informazioni per il codice emesso, con il fine di assistere il framework nella realizzazione dei meccanismi di protezione, rappresentati in sintesi dai moduli di Morphing e Shuffling.

La struttura standard di Clang prevede quindi un parser che costruisce un AST e un generatore di codice intermedio che, a partire dall'AST, emette un linguaggio intermedio specifico di LLVM, molto simile ad un generico instruction set della CPU, ma che astrae dall'hardware con risorse infinite, ad esempio usando un numero di registri infiniti.

Il codice generato in formato intermedio viene poi trasformato in codice eseguibile, attraverso una fase di generazione del codice specifica per architettura, oppure può essere interpretato attraverso la macchina virtuale di LLVM, che supporta *just in time compilation*.

3.2.2 Instruction set ARM

La CPU di riferimento per questo lavoro è una ARMv5 con instruction set a 32bit. L'instruction set ARM è di tipo RISC, con dimensione fissa delle istruzioni per facilitare la decodifica e con un set di 16 registri architetturali.

Come le architetture RISC, predilige la semplicità delle istruzioni, come ad esempio istruzioni di accesso diretto alla memoria solo di tipo Load/Store verso i registri, senza quindi il supporto ad istruzioni aritmetiche con operandi in memoria. Questo permette di avere un'unità di controllo semplice e con istruzioni che eseguono, per lo più, nello stesso numero di clock. Per garantire un buon numero di istruzioni per clock (IPC) eseguite a fronte della semplicità, l'ISA supporta l'esecuzione condizionata con un bit di guardia delle istruzioni.

In ARMv5 le operazioni che svolgono il caricamento di valori dalla memoria, da utilizzare nel codice come costanti, sono basate sull'istruzione LDR, che utilizza un offset rispetto al program counter per identificare il valore da caricare. La costante è quindi posta nella zona di memoria utilizzata dal programma eseguibile stream, in quelle che vengono chiamate *literal pool* oppure *constant pool*.

Per il framework implementato queste istruzioni sono di vitale importanze perchè, utilizzando il program counter e uno spiazzamento rispetto ad esso, necessitano di essere riposizionate, per esempio ricalcolando lo spiazzamento, qualora il framework aggiunga altre istruzioni.

Essendo la codifica dell'offset limitata in numero di bit (12bit), qualora la correzione dell'istruzione LDR richieda un offset maggiore, il framework dovrà introdurre una nuova *literal pool* all'interno dello stream del codice binario, aggiungendo i necessari salti nello stream, e ricalcolando per le altre istruzioni di salto gli spiazzamenti.

La figura 3.2 mostra un esempio di stream di istruzioni arm, prima e dopo l'aggiunta di due istruzioni, in verde, e mostra per l'istruzione LDR il ricalcolo dell'offset da 0x10 a 0x14, in base al program counter, utile per compensare l'aggiunta delle due nuove istruzioni.

L'istruzione finale di BR è un branch return usata per terminare il corpo di una funzione. Nella parola successiva di memoria è posta la costante che l'istruzione LDR caricherà in memoria; essendo questa variabile posta in memoria nella zona interpretata come codice eseguibile, si trova in una zona di memoria con permessi di sola lettura ed esecuzione; per questo si parla di accesso a valori costanti.

Address	**Value/Instruction**
0x00000000	...
0x00000004	LDR(0x0C,R1)
0x00000008	...
0x0000000C	RET(R14)
0x00000010	0xaabbccdd

Address	**Value/Instruction**
0x00000000	...
0x00000004	LDR(0x14,R1)
0x00000008	new instruction
0x0000000C	new instruction
0x00000010	...
0x00000014	RET(R14)
0x00000018	0xaabbccdd

Figura 3.2: Istruzione ARM LDR, ricalcolo spiazzamento

In figura 3.3, sempre sull'istruzione LDR, è invece mostrato il risultato di un riposizionamento della costante in una nuova pool all'indirizzo 0x10. Sono state ricalcolati gli indirizzi di salto a cavallo della nuova *constant pool* e aggiustati gli spiazzamenti delle istruzioni B.

L'esempio, per come è costruito, non mostra invece l'aggiunta di una istruzione di salto prima della nuova *constant pool*, utile a ristabilire il normale flusso di programma.

Address	**Value/Instruction**
0x00000000	...
0x00000004	B (+0x08)
0x00000008	LDR(0xFFC,R1)
0x00000008	B (+0x04)
0x0000000C	MV(0x01, R1)
0x0000000C	RET(R14)
0x00001004	0xaabbccdd

Address	**Value/Instruction**
0x00000000	...
0x00000004	B(+0x0C)
0x00000008	LDR(0x08, R1)
0x0000000C	B(+0x08)
0x00000010	0xaabbccdd
0x00000014	MV(0x01, R1)
...	...
0x0000000C	new instruction
...	...
0x00000FFC	RET(R14)
0x00001000	0xaabbccdd

Figura 3.3: Istruzione ARM LDR, aggiunta di una nuova pool

3.2.3 Implementazione di un attributo

Il framework fa uso di una versione estesa del compilatore Clang, realizzata attraverso l'introduzione di nuovi attributi. Il meccanismo dell'aggiunta di un nuovo attributo ad un compilatore è normalmente usato per fornire nuove funzionalità al di fuori dello standard del linguaggio, e per poter interagire con il compilatore e il compilato oltre la normale attività di compilazione del codice sorgente del programma. Con questo meccanismo, si può realizzare e modificare la generazione del codice.

Gli attributi del linguaggio C vengono aggiunti agli elementi sintattici del linguaggio, come le definizioni delle funzioni, delle variabili e dei tipi e sono, eventualmente, corredati con dei parametri.

Questo framework fa uso di attributi speciali introdotti in Clang, che svolgono le loro funzioni, sia in fase di generazione del codice, che in fase di emissione del codice prodotto, raccogliendo informazioni.

Definizione sintattica di un attributo

La registrazione di un nuovo attributo all'interno di Clang passa per la creazione di una nuova keyword e la definizione degli elementi sintattici connessi. Per il nuovo attributo devono quindi essere specificati gli altri elementi della sintassi a cui deve essere collegato, ad esempio una funzione o una variabile, ed eventualmente il numero e i tipi dei parametri necessari per la definizione dello stesso.

Per gli attributi implementati per il framework, i parametri passati sono principalmente nomi di altri simboli, che verranno verificati e referenziati dall'handler che gestisce l'attributo.

Questa definizione risiede in un file di definizioni [1], che verrà trasformato durante la compilazione di Clang in codice per il parser. Si tratta di un processo di generazione assistita del codice del parser del tutto equivalente a quello realizzato da Flex/Bison.

Durante la compilazione, il parser crea a partire dall'attributo un oggetto definizione che contiene tutte le informazioni sintattiche; a questo pun-

[1] `llvm/tools/clang/include/clang/Basic/Attr.td`

to un handler registrato dallo sviluppatore si occupa di svolgere un'analisi semantica sulla definizione.

Creazione di un attributo

L'aggiunta di un attributo a Clang prevede, dopo la creazione della keyword per il nome dell'attributo, la creazione di una funzione per la gestione dello stesso. Il parser, durante la compilazione, riconosce l'attributo e richiama l'*handler*, con informazioni quali: la definizione a cui è collegato e i parametri che accompagnano quella definizione.

Compito della funzione *handler* è riconoscere gli eventuali parametri, verificare la correttezza della definizione e, in ultima istanza, compiere delle modifiche sull'albero sintattico (AST), che è stato generato durante l'analisi da parte del *parser*.

Per il framework, normalmente, l'azione realizzata dal codice presente negli handler di llvm per gli attributi è l'aggiunta o di un attributo interno a LLVM o di metadati che guideranno la fase successiva di generazione del codice.

Sia gli attributi interni che i metadati sono, come è naturale aspettarsi, simili alla definizione di variabili o metodi, e presentano le informazioni contenute nell'attributo del linguaggio C.

Generazione del Codice Intermedio

Successivamente all'analisi sintattica e semantica, il compilatore Clang deve generare, a partire dall'AST costruito, il codice equivalente in linguaggio intermedio di LLVM .

Durante questa fase, gli handler registrati durante la fase precedente vengono eseguiti in una sequenza dipendente dall'emissione stessa del codice. In questa fase, possono operare sulla definizione di una funzione in linguaggio intermedio, cambiando e alterando istruzioni e variabili.

Il linguaggio intermedio di LLVM è caratterizzato, oltretutto, dal supporto all'aggiunta di metadata all'interno delle definizioni di simboli del linguaggio intermedio, con la possibilità quindi di costruire in modo più flessibile

delle funzionalità custom, andando a modificare e leggere le informazioni contenute direttamente nei metadata e senza quindi dover operare una estensione delle strutture dati tipiche del compilatore.

Questo approccio è di gran lunga il sistema maggiormente consigliato per l'estensione di LLVM, e per questo è anche lo strumento scelto per lo sviluppo e la realizzazione degli attributi per il framework.

3.2.4 Macro per l'integrazione

Questa sezione fornice una visione delle MACRO disponibili nel framework per la gestione e l'integrazione dello stesso all'interno di un progetto. Lo sviluppatore interessato all'utilizzo del framework le integra nel proprio codice sorgente per l'attivazione del sistema. Nella presentazione delle macro, si farà riferimento alla funzione da proteggere chiamandola "funzione":

- **ORIGINAL_FUNCTION**: mascheramento del simbolo
- **OF_RETURN_TYPE**: tipo del valore restituito della funzione
- **OF_SIGNATURE**: forma completa degli argomenti della funzione
- **OF_PARAMETERS**: lista dei nomi degli argomenti della funzione
- **MORPHER_CONTEXT**: nome della variabile del contesto

Queste macro sono utilizzate principalmente nel contesto di codice che si occupa attivamente dell'integrazione del framework, ad esempio il blocco di inizializzazione e finalizzazione del framework.

Queste macro coadiuvano lo sviluppatore, permettendogli di utilizzare un insieme di funzionalità preparate ad hoc, realizzate come template e definite nei sorgenti del framework.

3.2.5 Strutture dati per l'inizializzazione

L'inizializzazione del framework comporta, da parte dello sviluppatore utente, la preparazione e la gestione di due strutture dati:

- **morpher_context**: contesto del framework, uno per ognuna delle funzioni da proteggere con il morphing
- **shuffle_data**: elenco degli array di supporto allo shuffling

La struttura dati principale è chiamata *morpher_ context* e contiene tutte le informazioni necessarie per il framework per la gestione di una singola funzione da proteggere.

L'inizializzazione del framework costruisce un morpher_context per la funzione specificata; questo conterrà le analisi svolte sul codice binario della funzione, le dimensioni e i contatori per la gestione della frequenza di esecuzione. A carico della funzione di inizializzazione c'è anche l'allocazione della memoria necessaria.

Tra i parametri della funzione di inizializzazione, compare anche la seconda struttura dati, la cui gestione ricade in mano all'utente utilizzatore; si tratta della lista delle variabili per la realizzazione del modulo di shuffling.

Questa lista di array per il supporto allo shuffling è costituita da elementi che mappano un simbolo di un array con la relativa dimensione. Il modulo utilizza tale lista di tipo *shuffle_ data* ed esegue lo shuffling su ognuno degli array.

3.2.6 Strutture dati utilizzate internamente

Quella che segue è una lista delle altre strutture dati utilizzate internamente dal framework, di interesse per lo sviluppatore interessato ad estendere il framework:

- **function_layout**: struttura che contiene le dimensioni della funzione da proteggere
- **literalpool**: struttura che contiene le informazioni su una literal pool
- **loadstore**: struttura che contiene la lista dei riferimenti alle literal pool
- **bench_data**: struttura che contiene le informazioni per un elemento di benchmark
- **performance**: struttura che contiene tutti i bench_data del morpher

3.2.7 Parametrizzazione

Il modulo è stato parametrizzato per fornire all'utente finale una configurabilità post-compilazione, soprattutto per permettere di variare l'overhead

dovuto all'esecuzione dei moduli.

I parametri di controllo devono quindi essere passati al framework in fase di inizializzazione del modulo, utilizzando direttamente la funzione di inizializzazione del framework.

Sia per il modulo di Morphing che per quello di Shuffling, i parametri progettati per essere modificati sono la frequenza di esecuzione e il percorso al file contenente le regole di sostituzione delle istruzioni.

3.3 Modulo di Morphing

Questa sezione contiene una descrizione di che cosa è, come funziona e come è stato realizzato il modulo di morphing per la protezione di una funzione durante l'esecuzione.

Il modulo di Morphing del framework ha il compito di incrementare la protezione dell'implementazione di un algoritmo da un attacco di DPA. La protezione dell'algoritmo è realizzata a runtime dal modulo, attraverso l'interruzione del normale flusso di esecuzione e la trasformazione del codice binario dell'algoritmo. Nella sezione 2.3.1, sono disponibili maggiori dettagli sull'argomento ed è possibile trovare una classificazione delle tecniche.

I compiti svolti dal modulo si possono decomporre nelle funzionalità di sostituzione delle istruzioni, di ricostruzione dei *basic blocks* e di riposizionamento dei *literal pools*.

3.3.1 Fasi dell'esecuzione

Le attività svolte dal modulo durante l'esecuzione vengono qui presentate nel dettaglio e vengono distinte in tre fasi; ci riferiremo quindi all'inizializzazione, all'esecuzione e alla finalizzazione. La stessa struttura sarà ripresa successivamente per la presentazione del modulo di shuffling.

Per brevità, ci riferiremo all'algoritmo da proteggere come ad un singola funzione nel linguaggio C.

In inizializzazione

Il modulo è attivo durante la fase di inizializzazione del framework con il fine di svolgere un'analisi, una tantum, della funzione rispetto all'esecuzione del programma. In fase di inizializzazione, il modulo opera sul codice binario della funzione una scansione delle istruzioni di salto e di lettura da literal pools. Il fine di tale scansione è individuare e costruire due rappresentazioni della funzione, in forma di grafo dei *basic blocks* della funzione e in forma di mappa delle istruzioni che raggiungono una determinata *literal pool*.

Le rappresentazioni vengono memorizzate per un utilizzo successivo durante l'esecuzione del modulo; sono entrambe strutture dati utilizzate e direttamente collegate alla funzionalità di sostituzione. Hanno infatti il compito di svolgere rapidamente il calcolo degli offset, sia delle istruzioni di salto che delle istruzioni di load da literal pool.

In fase di inizializzazione, viene effettuata anche la lettura da file testuale della tabella dei *tile* di sostituzione, che contiene le regole per il riconoscimento delle istruzioni e l'applicazione delle sostituzioni.

L'implementazione, mutuata, realizza il caricamento di questa struttura attraverso l'uso di un parser, avvalendosi di un linguaggio definito per la scrittura delle regole.

Per maggiori dettagli su tali aspetti si rimanda alla sezione 3 di [Agosta et al., 2012].

In esecuzione

In fase di esecuzione, il modulo viene attivato dal framework e la frequenza di attivazione rispetto all'esecuzione della funzione è controllata con un parametro fornito in fase di inizializzazione. Una volta attivato, il modulo svolge il suo compito e procede con la costruzione di una variante del codice binario della funzione originale.

Come prima operazione, viene allocata dal modulo una zona di memoria sufficientemente grande rispetto alla funzione originale, che conterrà la variante della funzione. A questo punto, partendo dal fondo, sia di questa nuova zona di memoria che del blocco di codice binario della funzione, il

modulo inizia a ricostruire la funzione, utilizzando il motore di sostituzione delle istruzioni.

Per come è stata implementata in [Agosta et al., 2012], la funzionalità di sostituzione delle istruzioni opera localmente su una singola istruzione, oppure considerando solamente un sottoinsieme ben definito delle istruzioni ARM.

La trasformazione di una singola istruzione ha come risultato la creazione di un insieme di più istruzioni alternative; sia per le istruzioni successive che per quelle precedenti, è quindi importante modificare, ricalcolandoli, gli spiazzamenti (*offset*) delle istruzioni di salto e delle istruzioni di load da literal pool (LDR).

Il ricalcolo dello spiazzamento delle istruzioni LDR viene eseguito durante il normale flusso, perché lo spiazzamento è solo positivo e la processazione della funzione procede dal basso verso l'alto. Essendo quindi definitive le posizioni delle istruzioni successive alla corrente, gli offset delle LDR possono essere calcolati direttamente senza aver completato l'intera funzione.

Dove necessario, vengono introdotti delle nuove literal pool, prestando attenzione a non interrompere il flusso di esecuzione di un basic block. Nel caso degli offset di salto, si procede con il ricalcolo degli indirizzi solo alla fine della trasformazione completa del binario della funzione, ovvero quando il corpo e la dimensione della nuova funzione sono definitivi. In questo modo, le istruzioni di salto con offset negativi risulteranno posizionate e gli offset da calcolare definitivi.

Il ricalcolo di un offset prevede che, da una istruzione di salto, si debba dare l'offset per l'istruzione da raggiungere; per velocizzare il calcolo dell'offset, si è quindi optato per aggiungere un livello di tracciabilità per la coppia di istruzioni di partenza e arrivo.

Durante la fase di sostituzione delle istruzioni, si tiene quindi traccia, nella struttura dati dei basic blocks, delle posizioni delle istruzioni di partenza e arrivo nel nuovo codice binario.

A fine processazione, si scorre nuovamente la lista delle istruzioni di salto e, con l'ausilio dei nuovi indirizzi delle coppie, si sostituisce l'offset precedente con il nuovo valore.

Un'ulteriore ottimizzazione potrebbe distinguere tra offset negativi e positivi e scartare o tracciare solo quelli negativi.

Come ultimo passo, svolto nel framework per il modulo di sostituzione, un'operazione esposta dal sistema operativo forza la cache, nel momento del salto, a fornire la copia appena scritta nella memoria della funzione da eseguire, in modo da rendere consistente l'esecuzione. Le motivazioni per l'aggiunta di questa operazione si possono trovare nella sezione specifica 3.3.3.

In finalizzazione

Durante la finalizzazione del framework, la specifica funzione di *cleanup* del framework libera le zone di memorie riservate dal modulo per le strutture dati dei basic blocks e dei literal pools, e richiama la API di finalizzazione del parser usato per le regole di sostituzione, al fine di liberare la memoria delle strutture dati da esso create.

3.3.2 Integrazione

Per realizzare l'integrazione del modulo di morphing nel proprio codice, sono stati creati tre attributi del linguaggio C: un attributo per individuare la funzione da proteggere e due per calcolarne la dimensione; questi ultimi attributi fanno a loro volta uso della definizione di due variabili di supporto al modulo.

In inizializzazione il modulo richiede di specificare queste due variabili ed altre informazioni, come la frequenza con cui operare rispetto al normale flusso e la tabella contenente le definizioni delle sostituzioni.

Attributo: MorphingSubstitute

L'attributo principale utilizzato dal modulo è MorphingSubstitute; esso svolge il compito di sostituire/rinominare nell'eseguibile il simbolo della funzione da proteggere con il simbolo di una funzione del framework.

L'attributo è definito come direttamente collegato ad un elemento sintattico di tipo funzione; la sua definizione richiede un parametro che identifica il

nome di un'altra funzione. In questa implementazione, si è scelto di collegare l'attributo direttamente alla funzione da proteggere, e di usare il parametro per passare il nome della funzione wrapper usata dal framework.

Durante la compilazione, l'estensione integrata nel compilatore Clang svolge, nella fase di analisi dell'albero sintattico, la verifica e la risoluzione del parametro, nella funzione wrapper del framework. Il risultato finale di questa fase è l'aggiunta di un'istanza del gestore dell'attributo durante la fase di generazione del codice intermedio.

L'oggetto registrato di tipo MorpherSubstitute, quando viene richiamato dal gestore degli attributi, svolge il compito di sostituire, per ogni utilizzo della funzione da proteggere, una chiamata alla funzione wrapper del modulo di morphing.

Attributi e Parametri: MorpherSize, MorpherConstPool

Gli attributi MorpherSize e MorpherConstPool sono utilizzati per ottenere, sia la dimensione della funzione da proteggere, che la dimensione dell'ultima *literal pool* (se presente). Il compilatore si occupa di calcolare la dimensione e assegnarla alla specifica variabile. Tali attributi sono collegati alla variabile da assegnare e utilizzano entrambi un parametro per identificare la funzione di cui calcolare la dimensione.

Una volta identificati gli attributi, il parser aggiunge all'AST un handler, del tutto simile a quello usato dall'attributo MorpherSubstitute, che verrà eseguito durante la generazione del codice. Il suo compito è quello di aggiungere al linguaggio intermedio di LLVM i metadata necessari ad identificare il simbolo della variabile e il simbolo della funzione. Nel passo successivo della compilazione, durante la fase di emissione del codice binario specifico per la piattaforma, viene attivato un altro handler, che analizzerà il codice binario da emettere per la funzione da proteggere e, calcolando le dimensioni dei simboli marcati dall'attributo, inizializzerà le variabili corrispondenti con i relativi valori.

Parametro: substitution table

Il modulo di morphing utilizza un algoritmo di sostituzione delle istruzioni che fa uso di una tabella contenente le istruzioni equivalenti. In questa implementazione, è necessario, durante l'inizializzazione, fornire il percorso ad un file di testo contenente tale tabella di sostituzione.

Parametro: execution frequency

Il framework è parametrizzato per eseguire, con una frequenza a discrezione dell'utilizzatore, il modulo di morphing. È possibile specificare questa frequenza come parametro della funzione di inizializzazione del framework.

3.3.3 Flush della cache

In questa sezione viene presentata nel dettaglio l'aggiunta dell'operazione di flush della cache. Lo scopo di questa sezione è informare lo sviluppatore intenzionato a modificare il modulo, permettendogli di identificare immediatamente situazioni simili.

All'ambiente di sviluppo virtuale è stato sostituito un esecutore fisico ARMv5, lo stesso che verrà presentato nella sezione di testing. Qui, la prima implementazione del modulo di morphing ha presentato problemi durante l'esecuzione.

Si è giunti alla fase di testing con la consapevolezza che il modulo produceva una funzione binaria ragionevole, ma l'esecuzione della stessa generava problemi tra loro del tutto non correlati.

In seguito ad un'analisi più approfondita, si è notato che, disabilitata la sostituzione e realizzando quindi una semplice copia del codice del binario della funzione originale, l'esecuzione veniva ancora una volta inspiegabilmente alterata, analogamente al caso in cui il sotto-modulo di sostituzione fosse attivo; il flusso di esecuzione portava talvolta ad un errore di accesso alla memoria.

Ciò che ha reso l'analisi particolarmente complicata è stato che le funzionalità di debugging alteravano il flusso di esecuzione, mascherando il pro-

blema con esecuzioni regolari. Tale problematica può essere sintetizzata attraverso i due seguenti aspetti:

- gli approcci standard sono stati del tutto inefficaci ai fini dell'analisi del problema, poiché la semplice funzionalità di print su stdout era sufficiente a regolarizzare il flusso di esecuzione ed ottenere quindi una corretta esecuzione della funzione trasformata;
- la normale attività di debugging approfondita, quindi attraverso GDB, era anch'essa sufficiente a rendere l'esecuzione del codice prodotto corretta, poiché l'introduzione di un punto di interruzione prima o appena dopo il salto permetteva di regolarizzare l'esecuzione.

Inizialmente il problema è sembrato essere dovuto all'allineamento della literal pool, questo perché il codice eseguito durante l'inconsistenza portava spesso il program counter oltre la fine della funzione, cioè oltre le literal pools che si trovano alla fine del metodo. Si immaginava che una lettura da literal pools con offset non allineato, o con una literal pool non scritta come allineata, portasse l'unità di controllo in situazione d'errore.

Era già disponibile, come hardware, una board ARMv7, ma il passaggio a questa, che avrebbe permesso di scartare subito l'ipotesi precedente per via della mancanza del supporto alle literal pools, non era stato ancora completato.

La verifica si è a questo punto inutilmente concentrata sull'allineamento delle literal pools che venivano costruite; non è rimasto altro che supporre il problema come di origine hardware e passare quindi ad un'altra piattaforma. Anche in questo caso l'implementazione è risultata però afflitta dallo stesso problema.

La scoperta della causa del problema, e quindi la sua soluzione, sono giunte successivamente, quando, sostituendo anche la funzione dell'algoritmo AES con un'altra e incappando ancora una volta nello stesso problema, sono state riformulate tutte le ipotesi sullo stesso.

La causa è stata riscontrata essere uno stato, del tutto normale, di inconsistenza della cache istruzioni della CPU, documentata nelle specifiche di implementazione ARM. Come soluzione al problema, è stata aggiunta nel framework una chiamata alla funzione di flush della cache della CPU,

dopo il Morphing. La funzione di sistema specifica per ARM si chiama _-_*clear_cache* ed esegue il flush della cache corrispondente ad un range di indirizzi.

Per maggiori dettagli, il migliore riferimento è la sezione della documentazione ufficiale e della community ARM, argomento *caches and self modifyng code*[2].

Si suppone che questo problema non si sia presentato nella fase di testing del lavoro [Agosta et al., 2012], perché gli ambienti di esecuzione sono stati l'emulatore qemu - per lo sviluppo - e una board ARM9 - in testing - con esecuzione diretta da u-boot e cache disabilitata.

3.4 Modulo di Shuffling

Il modulo di Shuffling, in breve, ha come obbiettivo la protezione di un algoritmo da un attacco DPA, alterando la sequenza con cui vengono svolte le operazioni sui dati.

Il modulo realizzato in funzione dell'algoritmo aes_encrypt altera la sequenza dei valori assunti da una variabile di induzione di un ciclo.

I cicli alterabili dal modulo sono quindi quelli definiti dalla variabile di induzione esplicita, quando l'algoritmo opera in parallelo sui dati, come nel caso del vettore di stato dell'algoritmo di aes_encrypt.

Questa implementazione, con tali presupposti, non produce alterazioni o interferenze rispetto all'esecuzione naturale dell'algoritmo originale; è semplice eseguire questa verifica sul corpo di una funzione di un algoritmo.

Per maggiori dettagli, si rimanda alla sezione dello stato dell'arte 2.3.1.

3.4.1 Fasi dell'esecuzione

La presentazione delle attività svolte dal modulo di shuffling segue la struttura utilizzata per il modulo di morphing.

[2]http://community.arm.com/groups/processors/blog/2010/02/17/caches-and-self-modifying-code

Le attività svolte dal modulo durante l'esecuzione sono distinte in tre fasi: inizializzazione, esecuzione e finalizzazione.

Per brevità, ci riferiremo all'algoritmo da proteggere come ad un singola funzione nel linguaggio C, anche se nel nostro caso di studio sono state molteplici.

In inizializzazione

Durante l'inizializzazione del framework, il modulo costruisce nel contesto una lista di vettori di shuffling, e li inizializza con il range di valori corretti per la rispettiva variabile di induzione, per avere una corretta esecuzione dei rispettivi cicli.

In esecuzione

Durante l'esecuzione, il modulo viene attivato dal framework con una frequenza controllata da un parametro, e svolge il compito di procedere con la generazione di una nuova sequenza di indici di iterazione.

Il modulo di shuffling, una volta attivato, opera in sequenza sulla lista di vettori di shuffling con cui è stato inizializzato.

L'algoritmo di shuffling applicato è una versione in-place, che non alloca quindi un nuovo vettore, ed è noto in letteratura come l'Algoritmo P di Knuth, disponibile in [Knuth, 1998].

In finalizzazione

Durante la finalizzazione del framework, il modulo non svolge nessuna attività, perché, operando in-place direttamente sulla memoria da modificare, non ci sono zone di memoria o altre risorse di sistema da liberare.

3.4.2 Integrazione

Per l'integrazione del modulo di shuffling nel proprio codice, si è fatto uso di un attributo del linguaggio C, MurpherShuffle, realizzato per lo scopo, e dell'ausilio di variabili di supporto.

Il modulo è parametrizzato durante l'inizializzazione con la frequenza con cui eseguire, durante l'esecuzione, la sua funzionalità.

Attributo: MorpherShuffle

MorpherShuffle è l'unico attributo utilizzato dal modulo di shuffling; svolge la funzione di identificare la variabile oggetto dello shuffling e di sostituire le sue valutazioni (*right value*) con dei valori prelevati da un array.

L'attributo è definito come direttamente collegato ad un elemento sintattico di tipo funzione, e la sua definizione richiede due parametri che identificano due variabili.

La funzione a cui viene aggiunto l'attributo ha nel suo corpo un ciclo controllato da una variabile di induzione e da una variabile che opera su array, entrambe oggetti del modulo di shuffling. Uno dei parametri è proprio la variabile di induzione, mentre il secondo è una variabile array con elementi dello stesso tipo della variabile di induzione e che contiene la sequenza casuale di valori da utilizzare per la variabile di induzione.

La sequenza di valori assunti dalla variabile di induzione sarà quindi quella presente nell'array specificato e ricadrà sul componente a runtime del framework il compito di eseguire lo shuffling di questa sequenza.

Durante la fase di parsing, viene riconosciuto l'attributo e attivato il corrispondente handler. Quest'ultimo analizza i parametri e ne riconosce la corrispondenza tra i tipi, ovvero che la variabile è di un tipo base e che l'array è costituito da elementi dello stesso tipo base; l'handler registra quindi un oggetto gestore nell'AST, per l'esecuzione durante l'emissione del linguaggio intermedio.

Nella fase di emissione del linguaggio intermedio, l'oggetto gestore registrato nell'AST sostituisce gli accessi alla variabile di induzione con gli accessi ad un indice dell'array di shuffling, e identifica l'indice di accesso all'array di shuffling con la variabile di induzione.

Il codice che verrà emesso per queste funzioni conterrà, attraverso l'uso di questa variabile di induzione non più iterata in modo sequenziale, istruzioni di accesso in scrittura e lettura ai dati; tali istruzioni saranno consistenti

nell'insieme ma eseguite per un osservatore esterno non più in una sequenza fissa ma con un pattern random. Il modulo di shuffling modifica ad intervalli specifici la sequenza dei valori assunti dalla variabile di induzione.

Parametro: shuffle list

Per l'identificazione da parte del framework degli array di shuffling, è necessario, durante l'inizializzazione dello stesso, fornire la lista degli array di shuffling, costruendo per lo scopo un array di tipo shuffle_data per queste variabili di supporto. Il framework si preoccuperà di eseguire lo shuffling per questa struttura dati.

Parametro: execution frequency

Il framework è parametrizzato per eseguire il modulo di shuffling con una frequenza a discrezione dell'utilizzatore. È possibile specificare questa frequenza come parametro della funzione di inizializzazione del framework.

Capitolo 4

Verifica e risultati sperimentali

Questo capitolo affronta i test di verifica a cui è stato sottoposto il framework, soffermandosi a trattare la metodologia alla base dell'analisi prestazionale e analizzando, infine, i risultati dei test eseguiti. La sezione seguente presenta la verifica dell'implementazione, si è scelto di presentare anche questa metodologia avendo essa carattere introduttivo. A partire dalla sezione seguente, 4.2, si entrerà nel dettaglio per quanto riguarda i metodi e gli elementi oggetto dell'analisi prestazionale. Nella sezione 4.3 soni presentati sia i risultati attesi che l'analisi dei risultati concreti ottenuti.

4.1 Test di Verifica

In fase di sviluppo del framework, sono stati realizzati dei progetti prototipo con il compito di appurare la correttezza dei moduli implementati, attraverso la verifica dell'integrità dei risultati generati dalla versione della funzione da proteggere modificata dal framework.

Tale verifica è stata realizzata sul framework con un approccio empirico, realizzando e verificando le funzionalità che devono costituire il sistema. Questa sezione si occupa dei moduli completi di morphing e di shuffling, affrontandone la verifica e il test, prima individuali e poi come sistema integrato nel test finale.

Sono state quindi verificate singolarmente:

- l'estensione per il supporto al riposizionamento delle constant pool, introdotta in 3.2.2
- l'ottimizzazione del modulo di morphing, affrontata in 3.3
- l'implementazione dell'algoritmo di shuffling, alla base del modulo di shuffling, presentata in 3.4
- la funzione di sostituzione delle istruzioni

Come test finale, è stata verificata l'intera implementazione, eseguendo l'algoritmo di cifratura AES su un campione di coppie testo in chiaro e testo cifrato e verificando quindi la correttezza dell'implementazione attraverso il confronto del testo cifrato atteso con la cifratura.

4.1.1 Verifica singoli elementi

Nella verifica degli algoritmi seguenti, si è fatto riferimento genericamente a funzioni da analizzare o modificare; queste sono specifiche per lo scopo di verifica e del tutto scollegate dalla funzione da proteggere.

Verifica della gestione delle constant pool

Lo scenario per la verifica dell'algoritmo per il corretto ricalcolo e riposizionamento delle constant pool consiste nel simulare e poi verificare la riallocazione delle istruzioni, partendo dal codice binario di una funzione già nota. La funzione selezionata è stata l'implementazione dell'AES_128_encrypt di OpenSSL [aes_core.c], per la quale si conosceva già la posizione delle constant pool emesse dal compilatore per ARMv5.

Come già detto, la peculiarità delle istruzioni che utilizzano le constant pool di essere obsolete nel codice emesso per ARMv7 ha limitato l'esecuzione della verifica alla piattaforma ARMv5.

Per la funzione scelta per il test, il codice binario emesso dal compilatore risulta essere inferiore alla distanza massima di spiazzamento delle istruzioni di accesso alle constant pool [1]: precisamente, si tratta di meno di 2KB per il codice binario dell'intera funzione, contro i 4KB dell'offset massimo di un'istruzione da una constant pool.

[1] `http://infocenter.arm.com/help/topic/com.arm.doc.dui0041c/Babbfdih.html`

Si è optato quindi per modificare, all'interno del modulo implementato, una costante che specifica l'ampiezza massima di offset per ARMv5. In questo modo, si è ottenuto che l'algoritmo di riposizionamento delle constant pool operasse con un intervallo più piccolo; nel caso specifico si è scelto di usare un intervallo di 512 Byte per una distanza massima di spiazzamento di 128 istruzioni ARM equivalenti.

Verifica basic block

La funzionalità per la costruzione dei basic block è stata verificata analizzando una semplice funzione, costituita da una serie di blocchi condizionali e cicli. Si è provveduto per questa funzione a una verifica manuale dell'equivalenza della struttura di basic block generata dall'algoritmo di costruzione dei basic block rispetto alla struttura di partenza, intrinseca alle istruzioni emesse dal compilatore per la funzione.

Una ulteriore e più esaustiva verifica è stata completata utilizzando le istruzioni generate per la funzione AES_128 e usando come termine di paragone la struttura dati generata dalla funzionalità presente nella implementazione [Agosta et al., 2012], che era usata all'interno del modulo di sostituzione delle istruzioni.

Verifica algoritmo di shuffling

Per la verifica della corretta implementazione dell'algoritmo usato per lo shuffling è stato utilizzato un prototipo che eseguiva lo shuffling e infine verificava la presenza di ogni elemento dell'array originale nella versione trasformata.

4.1.2 Verifica degli attributi

La verifica delle funzionalità del framework comprende anche il corretto funzionamento degli attributi sviluppati sia a compile-time che a runtime.

Per la verifica che ha riguardato gli attributi MorpherShuffling e MorpherSubstitute, si è scelto di realizzare un prototipo che verifica l'attivazione a runtime delle specifiche funzionalità di questi ultimi.

Per il primo, MorpherShuffling, si è usata una funzione che comprendeva l'iterazione di un loop e verificava i risultati delle operazioni eseguite sui singoli elementi dell'array. Per il secondo attributo si è verificato che il codice generato dal compilatore contenesse al posto dei salti alla funzione specificata, dei salti alla funzione del framework.

Per la verifica degli attributi che calcolano la dimensione delle constant pool (MorpherSize, MorpherSizePool), si è proceduto sperimentalmente, modificando il corpo della funzione oggetto degli attributi e verificando il comportamento degli stessi.

MorpherShuffle e MorpherSubstitute

La verifica dell'attivazione e del funzionamento del modulo di shuffling, nello specifico dell'attributo MorpherShuffle, è stata realizzata con un prototipo che applicava il modulo di shuffling ad una funzione che eseguiva una somma degli elementi di un array, e verificava quindi che il risultato finale corrispondesse alla somma.

La verifica dell'attivazione e del funzionamento del modulo di morphing è stata realizzata accertando la corretta sostituzione delle chiamate di una funzione con un'altra, attraverso l'utilizzo di due funzioni scritte per lo scopo. Le due funzioni avevano la medesima definizione e diverso corpo; la funzione sostituta terminava il programma con una condizione d'uscita normale, mentre la funzione sostituita con una condizione d'errore.

MorpherSize e MorpherSizePool

La verifica dell'attributo che calcola la dimensione della funzione da proteggere è stata realizzata attraverso l'impiego degli strumenti della libreria C, nello specifico attraverso il programma *nm*, che permette di ispezionare la struttura di un eseguibile.

Si è riscontrato che le dimensioni rilevate da *nm* per i simboli e per gli oggetti nell'eseguibile prodotto corrispondevano ai valori che il gestore dell'attributo aveva rilevato e assegnato alla variabile aggiunta al binario.

Per l'attributo MorpherSizePool, si è provveduto anche a verificare il corretto calcolo della dimensione delle constant pool, accertando che, al variare del numero di simboli globali utilizzati dalla funzione marchiata, crescesse la dimensione della constant pool calcolata.

4.1.3 Verifica del framework

Il test dell'intero framework, comprendente quindi entrambe le funzionalità di Morphing e Shuffling, è stato realizzato attraverso la verifica dell'algoritmo di cifratura AES, con un campione di coppie di dati: dato in ingresso e dato in uscita, nel caso dell'AES corrispondenti al testo in chiaro e al testo cifrato.

Il prototipo per la verifica, per motivazioni legate a peculiarità hardware e software che affronteremo nel dettaglio nella sezione 4.2, è stato eseguito e compilato per hardware ARMv5 e con l'implementazione di AES encrypt adatta per l'esecuzione del modulo di shuffling.

4.2 Metodologia ed elementi dell'analisi delle prestazioni

Il framework è stato sottoposto ad un'analisi delle prestazioni, al fine di stabilire l'overhead computazionale generato dallo stesso rispetto alla semplice esecuzione dell'algoritmo da proteggere e ottenere così una valutazione del costo d'esecuzione per un utilizzo realtime attraverso un'esplorazione dei parametri di morphing, shuffling e compilazione. Oggetto dell'analisi prestazionale sono quindi stati i tempi d'esecuzione degli eseguibili generati dal framework, per ognuno dei quali sono stati raccolti i tempi di esecuzione dei moduli del framework e dell'algoritmo da proteggere, la funzione AES.

Data la variabilità dell'hardware disponibile in ambiente embedded, si è scelto di eseguire il benchmark su piattaforme di generazione differente con instruction set sia ARMv5 che su piattaforme con ARMv7. La scelta di utilizzare piattaforme con il più arretrato instruction set ARMv5 è stata dettata dalla valenza numerica della base installata, e quindi dalla necessità di verificare le prestazioni del sottomodulo che si occupa della gestione delle constant pool, le cui funzionalità dipendono dalla presenza delle istruzioni LDR nel codice generato per la funzione AES (si veda a tal proposito la sezione 4.1.1).

I test sono stati articolati per coinvolgere tutti i moduli del sistema e, per la peculiarità dell'implementazione dell'algoritmo di shuffling di essere applicato in presenza di una variabile di induzione esplicita, si è optato per utilizzare per il test differenti implementazioni dell'algoritmo da proteggere.

4.2.1 Implementazioni di AES_128_encrypt

Per l'implementazione dell'analisi prestazionale, è stata scelta la funzione di cifratura AES_128_encrypt secondo due implementazioni differenti: l'implementazione OpenSSL standard, che fa uso di tabelle di sostituzione o TTable, e l'implementazione standard dell'AES_128 in linguaggio C. Quest'ultima implementazione, scelta per la presenza all'interno dei suoi cicli di esplicite variabili di induzione, verrà di seguito denominata *implementazione scolastica* o più brevemente *scholarbook*. L'implementazione OpenSSL standard dell'AES, precisamente quella presente nel file sorgente del progetto `morpher/aescode/aes_core.c,` o nel pacchetto dei sorgenti openssl in `crypto/aes/aes_core.c`, utilizza un ciclo infinito e la SBOX in forma tabellare.

Le implementazioni OpenSSL di AES, oltre ad essere di riferimento, sono anche ottimizzate e ad esempio sono disponibili versioni specifiche per le piattaforme hardware più note, come x86_64; sono state scelte soprattutto in virtù della presenza, per alcune di esse, di supporto hardware per l'esecuzione di alcuni passi dell'algoritmo AES.

L' implementazione OpenSSL scelta è priva nel codice da proteggere della funzione *aes_128_encrypt* di un loop con la presenza esplicita di una variabile di induzione, rendendo impossibile quindi l'applicazione dello shuffling da parte del framework.

Per contro, lo shuffling è di immediata applicazione all'implementazione scolastica, facendo questa uso di variabili di induzione nel codice di molte funzioni che realizzano le operazioni dell'algoritmo AES. É possibile verificare l'applicabilità dello shuffling, per esempio attraverso una semplice ispezione visiva del codice sorgente delle funzioni MixColumns, AddRoundKey e ShiftRows, presenti precisamente nel file sorgente del progetto `morpher/aes_base/aes.c`; per queste funzioni citate è possibile verificare la presenza di una variabile di induzione nei cicli *for* del codice.

4.2.2 Algoritmi e Parametri

Per l'algoritmo OpenSSL standard, si è quindi proceduto ad eseguire il benchmark per l'esecuzione normale e per il solo modulo morphing. Per il benchmark dell'implementazione scolastica, sono stati invece applicati sia lo shuffling che il morphing.

Il seguente elenco riassume le implementazioni scelte per l'algoritmo *AES_128_encrypt* in funzione dei moduli del framework:

- OpenSSL
- OpenSSL con Morphing
- Scholar
- Scholar con Morphing e Shuffling

Le applicazioni dei moduli morphing e shuffling del framework sono controllate da due parametri, che regolano la frequenza con cui a runtime i moduli intervengono, interrompendo la normale esecuzione; tali parametri governano quindi la frequenza con cui vengono eseguiti il modulo che trasforma le istruzioni della funzione da proteggere e il modulo che riordina gli array che definiscono la sequenza di induzione.

All'aumentare del valore di un parametro cresce l'intervallo tra una esecuzione e la successiva e quindi diminuisce la frequenza con cui il framework

esegue il modulo. Questa possibilità di controllare ogni quanto, durante l'esecuzione dell'applicazione, il framework interrompa la normale esecuzione ed operi le sue attività, è stata oggetto di valutazione nel lavoro [Agosta et al., 2012] e si tratta di una funzionalità molto vantaggiosa, perchè permette di avere un meccanismo, basato su trade-off, che fornisca maggiore sicurezza al costo di maggiori tempi di esecuzione.

Si suppone che l'influenza di questi parametri sul benchmark per quanto riguarda il morphing sia duplice, perché non si limita al controllo diretto del tempo di esecuzione variando la frequenza di esecuzione, ma, come effetto secondario, all'aumento della frequenza dei moduli aumenta anche il numero di operazioni di cache flush, che resettano la cache della CPU degradando le prestazioni dell'esecuzione successiva. Maggiori dettagli sui risultati del benchmark e l'analisi degli stessi sono riportati nella sezione 4.3.2.

4.2.3 Opzioni di compilazione

Al fine di esplorare i risultati generati dalle possibili opzioni di compilazione, sono state scelte le seguenti opzioni disponibili :

- **OS**: optimize for size
- **O1**: optimize
- **O2**: optimize for performance
- **O3**: all optimization
- **loop**: loop unrolling

Maggiori dettagli sulle stesse sono disponibili in [Gcc, 2014].

È molto probabile che il compilatore Clang nella versione corrente 3.5 possa fornire migliori risultati di ottimizzazione nella produzione del codice.

I risultati presentati nella sezione 4.4 contengono i dati raccolti per ognuna delle opzioni.

Per il campione di analisi presentato come overhead dei moduli rispetto alle funzioni, sono presenti anche i tempi dell'esecuzione del sistema in modalità di compilazione debug, per fornire un upperbound rispetto ai tempi di esecuzione ottenuti includendo le ottimizzazioni.

4.2.4 Piattaforme Hardware

Le piattaforme hardware scelte per l'analisi prestazionale sono: una Pogoplug-V2 Marvell 88FR131 e processore single core Sheeva, una Pandaboard ES con CPU Omap 4 Dual Core 4460 e una Odroid XU con CPU Exynos 5 5410.

Pogoplug S02

La CPU Sheeva ha supporto a *instruction set* ARMv5te; nel benchmark esegue ad un clock 1.2Ghz ed ha una pipeline single issue da 5-8 stadi. La cache L1 è da 16KB+16KB, la L2 è da 256KB. Maggiori informazioni sulla CPU sono disponibili a questi riferimenti: la scheda tecnica in [7-cpu.com, 2014], il manuale completo [marvell.com, 2014], la documentazione di supporto del kernel [kernel.org, 2014] ai device disponibili e alle opzioni di compilazione del kernel. Una lista delle CPU ARM in commercio è disponibile in [wikipedia.org, 2014b] e nello specifico per la specifica dei core delle generazione ARM9e [wikipedia.org, 2014a]. Il sistema completo comprendeva 256MB di ram, kernel Linux 3.7.0, gcc 4.7.3 e clang 3.1.

PandaBoard ES

La CPU nell'OMAP 4460 è un Cortex-A9 con supporto a *instruction set* ARMv7; nel benchmark esegue ad un clock 700Mhz ed ha un core di tipo *out of order*, con *speculative issue* e *super-scalar*. È disponibile il manuale completo della board in [ti.com, 2014]

La cache L1 è da 32KB+32KB, la L2 è da 1MB condivisa. Il sistema completo comprendeva 1Gb di ram, kernel Linux 3.7.0, gcc 4.7.3 e clang 3.1.

Odroid-XU

La CPU dell'Odroid-XU è un big.LITTLE costituito da due *die* distinti, contenenti uno un quad core Cortex-A15 e l'altro un quad core Cortex-A7. Il processore usato per il testing è il Cortex-A15 con clock 1.6Ghz e *instruction set* ARMv7; l'architettura ha una pipeline da 15-25 stadi, *speculative issue*, *out of order* e *super scalar*. La cache L1 è da 32KB+32KB, la L2 è da 2MB

condivisa. Il sistema completo comprendeva 2Gb di ram, kernel Linux 3.7.0, gcc 4.7.3 e clang 3.1.

In configurazione standard l'architettura big.LITTLE si adatta al carico di sistema passando un processo in esecuzione una CPU *LITTLE*, che utilizza completamente le risorse, ad una CPU di tipo *big*.

Questo comportamento ha lo svantaggio di rendere poco consistenti i tempi rilevati; per questo motivo, i test eseguiti su questa piattaforma sono stati realizzati con un supporto all'architettura big.LITTLE parziale, in modo da garantire la consistenza dei risultati.

Nella configurazione usata per i test e mostrata in A.3 sono presenti due core ad alte performance (Cortex-A15), mantenuti sempre attivi.

4.2.5 Framework di distribuzione

Per facilitare e automatizzare la realizzazione e la raccolta dei dati del testing, è stato realizzato un framework che si occupa della distribuzione, sia delle operazioni di compilazione, che dell'esecuzione del binario di testing sulle singole piattaforme. Il framework di distribuzione si occupa anche di generare uno *stacktrace* in caso di crash dell'esecuzione, funzionalità che è risultata particolarmente utile sia in fase di sviluppo del framework, che in fase di realizzazione del testing di verifica.

Estensione

Qualora uno sviluppatore volesse estendere l'ambiente di analisi con una nuova piattaforma, sarebbe sufficiente l'aggiunta al framework di distribuzione dei dettagli sulla stessa, come la tipologia della CPU e alcuni percorsi dell'ambiente d'esecuzione.

La realizzazione e l'impiego di un'estensione richiedono un ambiente di esecuzione python3.3+, il supporto alla libreria Pyro (versione 4.18 e successive) e, per l'aggiunta di una piattaforma, la scrittura di un modulo in python, con informazioni di configurazione riguardanti la tipologia di CPU e i percorsi per il compilatore e le librerie di sistema installati.

Per quanto riguarda i dettagli riguardanti le CPU ARM, queste informazioni sono soprattutto la presenza nell'ambiente di esecuzione di sistema delle librerie hardfloat o softfloat e del supporto fisico alle stesse, perché ciò permette di distinguere tra le librerie da utilizzare per il linking e la compilazione di codice a virgola mobile e la modalità di esecuzione in cui si trova la CPU.

4.2.6 Timing dei moduli e calcolo statistico

Per la scelta del metodo di rilevazione dei tempi di esecuzione, si è optato per un approccio invasivo con l'aggiunta di codice, ma limitando la rilevazione dei tempi alle sole funzioni del framework.

All'interno del codice sorgente dei moduli del framework, sono stati inseriti dei blocchi di codice per la rilevazione del tempo di esecuzione; tale codice di rilevazione è controllato da delle macro, per permettere, utilizzando semplicemente un'opzione di compilazione, di ottenere dei binari privi dell'overhead generato dal codice usato per la rilevazione e l'analisi.

Per la rilevazione del tempo di esecuzione, si è optato per l'utilizzo della funzione di sistema *clock_gettime* di glibc. Maggiori dettagli sull'impiego per il profiling sono disponibili all'indirizzo [Rutenberg, 2014]

Tale funzione della libreria glibc è in grado di effettuare misurazioni con granularità dell'ordine dei nanosecondi, qualora la piattaforma hardware sottostante supporti una sorgente hardware di clock con tale granularità. Nel caso della Pandaboard, si è scoperto che la granularità minima del clock è di 32.000 nanosecondi; ciò ha reso invalide le misurazioni eseguite con tempi inferiori. Maggiori dettagli riguardo tale aspetto sono disponibili nella sezione 4.3.4.

Per l'analisi delle rilevazioni, si è optato per scindere le due fasi di rilevazione e di analisi, eseguendo quindi soltanto la rilevazione a runtime e spostando l'analisi dei tempi raccolti offline.

Durante l'esecuzione, il tempo rilevato viene bufferizzato in un blocco di 8KB; i campioni di tempo misurati vengono periodicamente scritti in un file binario. Le rilevazioni vengono quindi analizzate offline, per mezzo del modu-

lo statistico presente all'interno della libreria numpy, la cui documentazione è disponibile all'indirizzo [numpy.org, 2014].

Tale approccio misto ha permesso di rilevare grandi campioni di dati, minimizzando la propagazione degli errori di approssimazione dovuti alle operazioni di calcolo della media e della varianza effettuandole offline e con una precisione maggiore rispetto allo stesso calcolo eseguito durante l'esecuzione stessa.

Oltre alla rilevazione del tempo di esecuzione dei moduli di morphing e shuffling, sono stati raccolti i tempi di inizializzazione del framework e di esecuzione della funzione da proteggere.

4.3 Analisi dei risultati

Questa sezione tratta sia dei risultati attesi per quanto riguarda la rilevazione dei tempi di esecuzione, sia dell'analisi dei tempi stessi rilevati dai campioni.

Nello specifico, ci si sofferma sul comportamento previsto per il campione dei dati, sia al variare del livello di ottimizzazione del binario dell'eseguibile (Sezione 4.3.1), sia in seguito ad operazioni aggiuntive da svolgere come il flush della cache (Sezione 4.3.2).

Le utlime due sottosezioni analizzano in dettaglio il campione dei dati raccolti, per mostrare il problema della granularità del tempo su Pandaboard (Sezione 4.3.4) e per mostrare la distribuzione del campione di dati raccolto (Sezione 4.3.5).

4.3.1 Influenza ottimizzazione binario sull'esecuzione

Dall'analisi dei risultati, si prevede di verificare che, con l'aumento del livello di ottimizzazione della compilazione, migliorino o rimangano pressoché costanti i tempi di esecuzione sia dei moduli del framework che della funzione AES.

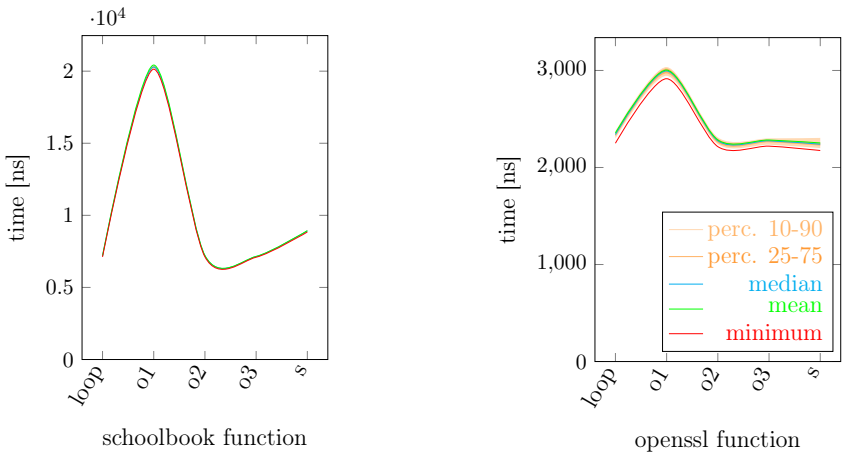

Figura 4.1: A sinistra il campione Pogoplug Scholbook function. A destra il campione Pogoplug Openssl function

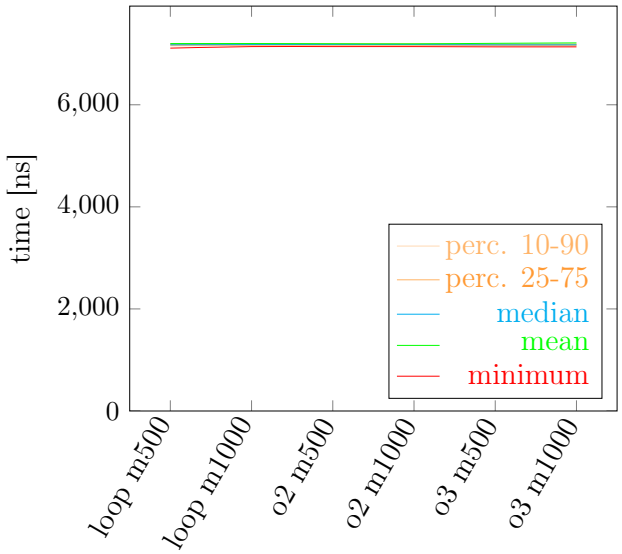

Figura 4.2: Campione Pogoplug Scholar-Morpher Function

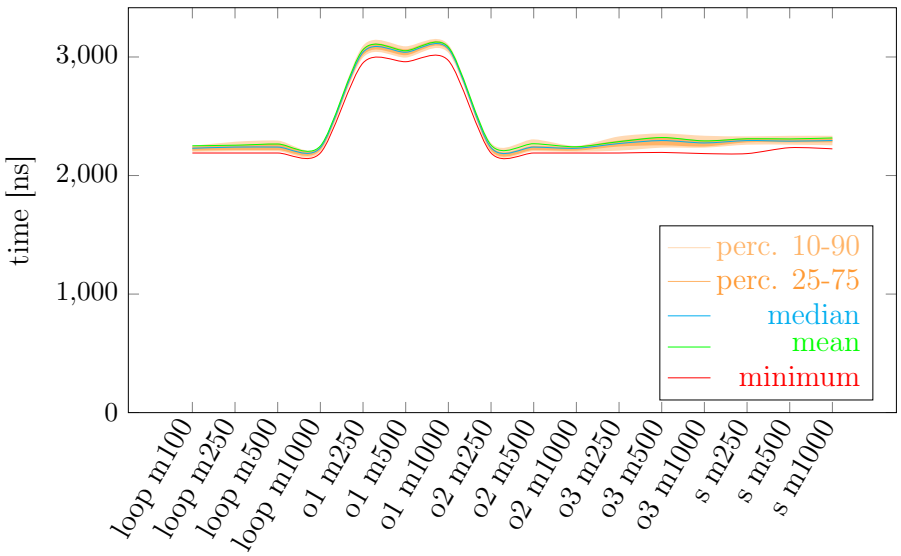

Figura 4.3: Campione Pogoplug Opensslmorpher Function

Per la piattaforma hardware PogoPlug con ARMv5 sono mostrati in figura 4.1 i tempi di esecuzione della funzione AES sia nella versione schoolbook che openssl, e quelli delle rispettive varianti con il framework di morphing nelle figure successive 4.2.

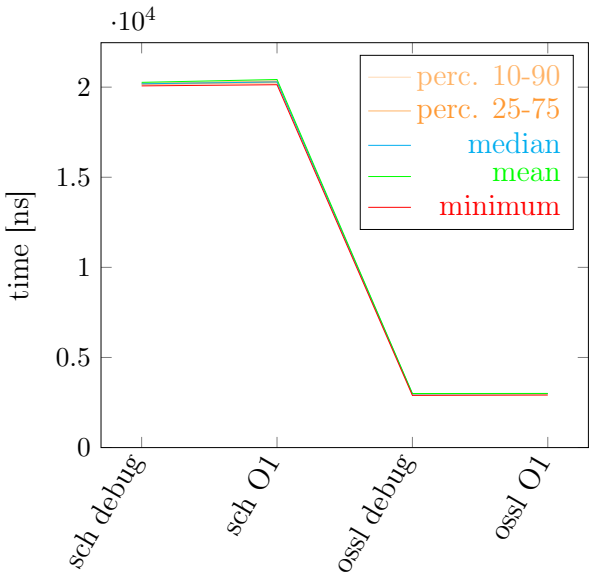

Figura 4.4: Campione Pogoplug Function, livelli di ottimizzazioni O1 e debug

Da questi grafici si possono riscontrare per ARMv5 che i tempi di esecuzione della funzione AES sono leggermente maggiori per le ottimizzazioni **size**; il livello di ottimizzazione **O1** ha tempi completamente differenti e come si può vedere in figura 4.4 del tutto paragonabili ad un eseguibile compilato in modalità debug.

Consultando anche i dati presentati nelle tabelle nella Sezione 4.4, che mostrano una vista aggregata dei risultati dell'analisi degli overhead, si può concludere che al variare del livello di compilazione, i risultati migliorano o restano costanti.

L'assunto sulla consistenza dei risultati con il crescere del livello di ottimizzazione è verificato tanto che risultano indistinguibili i tempi rilevati per le ottimizzazioni **O2**, **O3**, **loop** e **size** sulle piattaforme e, dove questo non è vero, i tempi rilevati per le ottimizzazioni **O2** e **O3** risultano migliori rispetto alle altre ottimizzazioni.

4.3.2 Influenza della operazione di cache flush sull'esecuzione

Considerando che il modulo di morphing per un corretto funzionamento richiede l'operazione di flush della cache (Sezione 3.3.3), si prevedeva che questa avesse un effetto penalizzante sul tempo medio di esecuzione della funzione AES, costringendo la CPU a ricostruire la cache delle istruzioni in corrispondenza dell'operazione di flush.

Il tempo dovuto all'inserimento di un cache flush è un costo, dipendente sia dall'architettura che dall'implementazione, ma comunque da pagare indipendentemente dal livello di ottimizzazione a cui sono stati sottoposti i moduli.

Esso rappresenta un costo fisso nell'esecuzione del morphing; infatti, tale costo è probabilmente da mantenere anche per un'esecuzione in parallelo del sistema del modulo di morphing (5.2.2), nella quale si preveda quindi ottimisticamente di avere sempre pronta una nuova versione riscritta della funzione da proteggere.

L'operazione di cache flush è infatti necessaria a garantire la coerenza della cache istruzioni, allineando quindi il contenuto della cache dati dove è stata appena modificata la funzione con quello della cache istruzioni che conterrà le istruzioni da eseguire.

Per il campione di test eseguiti, si prevedeva di osservare nei tempi di esecuzione raccolti una correlazione biunivoca tra impiego del modulo di morphing e aumento dei tempi di esecuzione della funzione modificata.

Si presumeva di osservare un aumento dei tempi medi di esecuzione anche confrontando tra loro campioni parametrizzati con esecuzioni via via più frequenti del modulo di morphing, come conseguenza dell'aumento di operazioni di cache flush.

I dati presenti nella corrispondente nella tabella 4.3 e riportati sinteticamente nella figura 4.5 mostrano, invece, un aumento dei tempi di esecuzione medi al diminuire della frequenza di esecuzione del modulo di morphing, precisamente nel caso degli intervalli di esecuzione maggiori (1000, 2500).

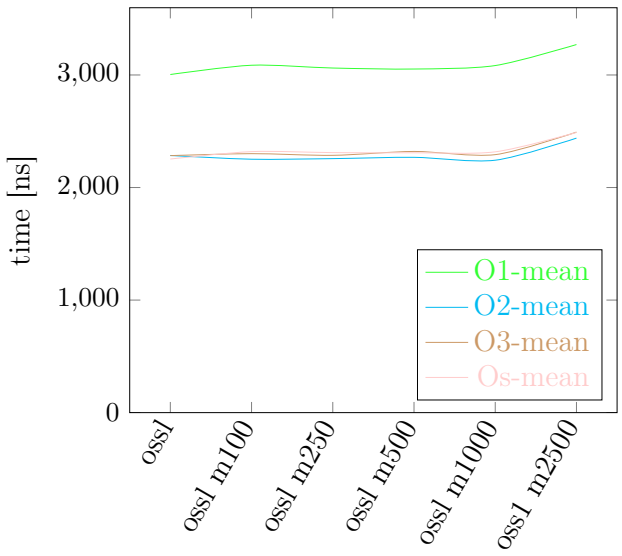

Figura 4.5: Campione Pogoplug Function, al variare del livello di ottimizzazione e della parametrizzazione del modulo di morpher

4.3.3 Dipendenza del framework dalla parametrizzazione

Nella sezione precedente si è ricercata una relazione che legasse alla crescita dell'intervallo di esecuzione del framework una diminuzione dei tempi di esecuzione della funzione AES. In questa sezione è d'interesse la ricerca di quei comportamenti dipendenti dalla parametrizzazione in modo più generico.

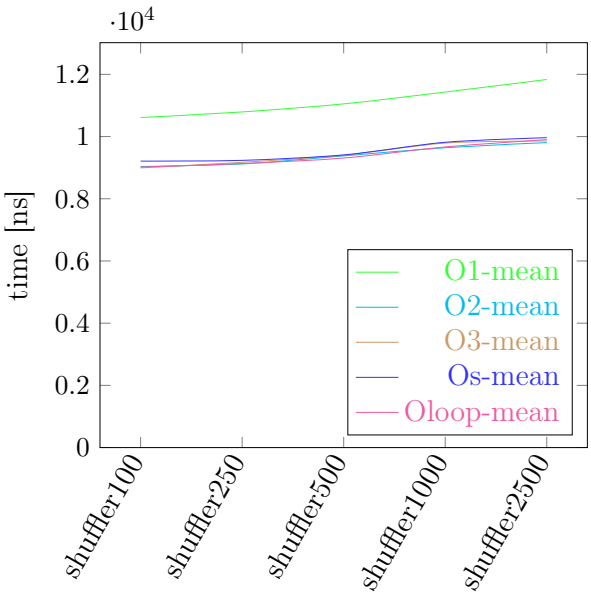

Figura 4.6: Campione Pogoplug modulo shuffler: al variare della parametrizzazione e del livello di ottimizzaziones

La figura 4.6 mostra i tempi rilevati per il campione del modulo che esegue lo shuffling, nello specifico su piattaforma PandaBoard e per l'implementazione scholar.

È possibile riscontrare, osservando i campioni rilevati, una dipendenza dei tempi di esecuzione del modulo di shuffler dalla parametrizzazione dell'intervallo di esecuzione del framework; nello specifico al crescere dell'intervallo, da una esecuzione ogni 100 esecuzione della funzione AES a una ogni 2500, cresce il tempo con cui viene eseguita la funzione di shuffling.

Questo comportamento è comune per ognuno dei differenti livelli di ottimizzazioni dell'eseguibile prodotta dal compilatore, infatti il comportamento è osservabile per il grafico della ottimizzazione **O1**, in verde nel grafico, quanto per le altre.

4.3.4 Problema: granularità del tempo di esecuzione su Pandaboard ES

La scheda Pandaboard ES utilizzata è fornita di un clock con granularità di $32kHz$, capace quindi di rilevare i tempi di esecuzione con una precisione approssimativamente di $33\mu s$; maggiori informazioni sono reperibili all'indirizzo [Launchpad, 2014].

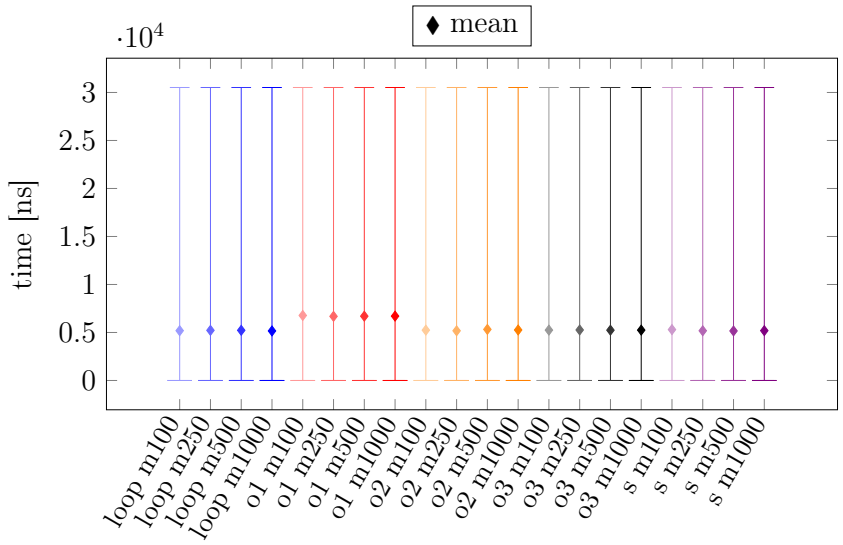

Figura 4.7: Campione dei tempi rilevati per l'esecuzione della funzione AES_openSSL in esecuzione con il modulo di morphing

È possibile rilevare questa condizione osservando i campioni analizzati per la Pandaboard ES; come si può osservare nella figura seguente 4.7, alcuni dei valori minimi assunti dal campione sono pari a zero.

Avendo i campioni raccolti per la Pandaboard valori minimi pari a zero, i tempi medi calcolati potrebbero risultare nettamente inferiori alla media reale. I campioni raccolti per l'esperimento mostrato in figura 4.7 presentano oltre allo zero dei valori equivalenti al minimo rilevabile dal clock del dispositivo.

Dal punto di vista statistico, il campione presenta un tempo di esecuzione del tutto inferiore alla soglia di rilevazione; probabilmente il valore medio rilevato è ancora significativo perché rispetta la definizione di tempo medio come $\frac{tempo\ totale}{numero\ di\ campioni}$.

Dal punto di vista delle statistiche di ordine superiore, questa soglia di rilevazione minima comporta la presenza di un valore zero in luogo di valori reali, introducendo quindi dei valori fuori scala rispetto al valore effettivo; valori che rendono l'analisi dello scarto quadratico medio e quindi dello deviazione standard inconsistente.

Il grafico, nonostante sia un boxplot con informazioni anche sui quartili, perde di significatività e si riduce a mostrare il range di valori osservati: tra 0 e il valore minimo rilevabile.

A seguito di queste considerazioni si è provato a filtrare l'elenco dei valori rilevati rimuovendo dal campione gli 0, ma questo procedimento altera completamente l'analisi come mostra la figura 4.8.

Infatti a titolo di verifica di quanto riscontrato sono mostrati il numero di elementi con valore 0 sul totale del campione di 10,000 valori rilevati, variando sia le opzioni di compilazione che quelle dell'intervallo di attivazione del modulo di morphing.

Per ognuno degli esperimenti sulla funzione AES di OpenSSL e in esecuzione con il modulo di morpher, in esecuzione naturalmente su Pandaboard, è mostrato il corrispondente numero di campioni dei tempi rilevati con valore 0; sempre su un massimo di valore 10,000.

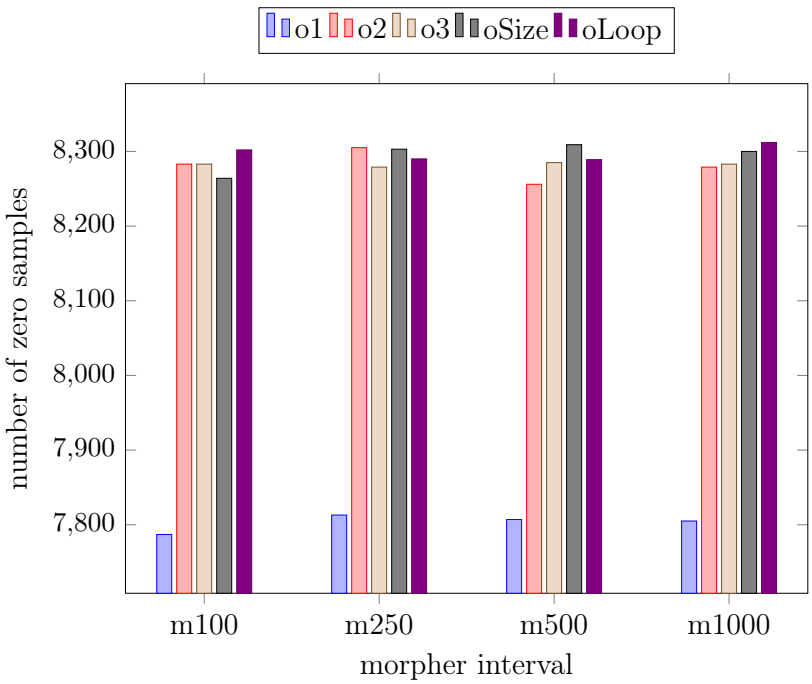

Figura 4.8: Pandaboard AES_function openssl-morpher: Numero di campioni con valore 0 su un totale di 10.000

La granularità del clock genera una soglia di rilevazione dei tempi di esecuzione; per mostrare la soglia è stato realizzato ed è stato sottoposto a benchmark un prototipo che esegue un numero variabile di istruzioni tra una rilevazione e l'altra al solo scopo di identificarla visivamente sul grafico.

Il grafico in figura 4.9 corrisponde ai tempi del prototipo in esecuzione su Pandaboard, al crescere del numero di istruzioni aritmetiche eseguite per ogni campione di 10,000 rilevazioni. Quindi sull'asse X sono indicate le operazioni eseguite tra una rilevazione e l'altra.

È di facile individuazione sul grafico dei dati mostrato in figura 4.9 la soglia minima rilevabile; si può notare come oltrepassando i 31000 nano secondi di media i campioni non presentino più minimi con valore 0.

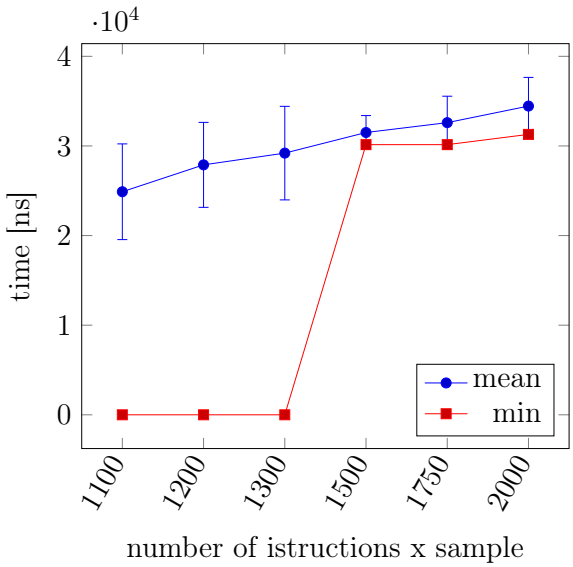

Figura 4.9: Pandaboard, grafico dei tempi di esecuzione rilevati al variare del numero di istruzioni eseguite per campione

4.3.5 Standard Error mostra lo spostamento dei campioni

Sono stati preparati dei grafici che mostrano la distribuzione del campione raccolto precisamente per architettura ARMv5, in esecuzione quindi sulla Pogoplug (Sezione 4.2.4), che illustrano i tempi di esecuzione della funzione AES sia nel caso dell'implementazione con algoritmo OpenSSL che in quello della funzione Scholar.

I dati mostrati corrispondono ai tempi di esecuzione per la funzione anche con l'utilizzo del framework di morphing, con una parametrizzazione dell'intervallo di attivazione rispettivamente di 500 e 1000, quindi con una iterazione del framework (morpher e shuffler) ogni 500 o 1000 esecuzioni della funzione AES; sono stati tralasciati i grafici delle altre parametrizzazioni in quanto del tutto simili.

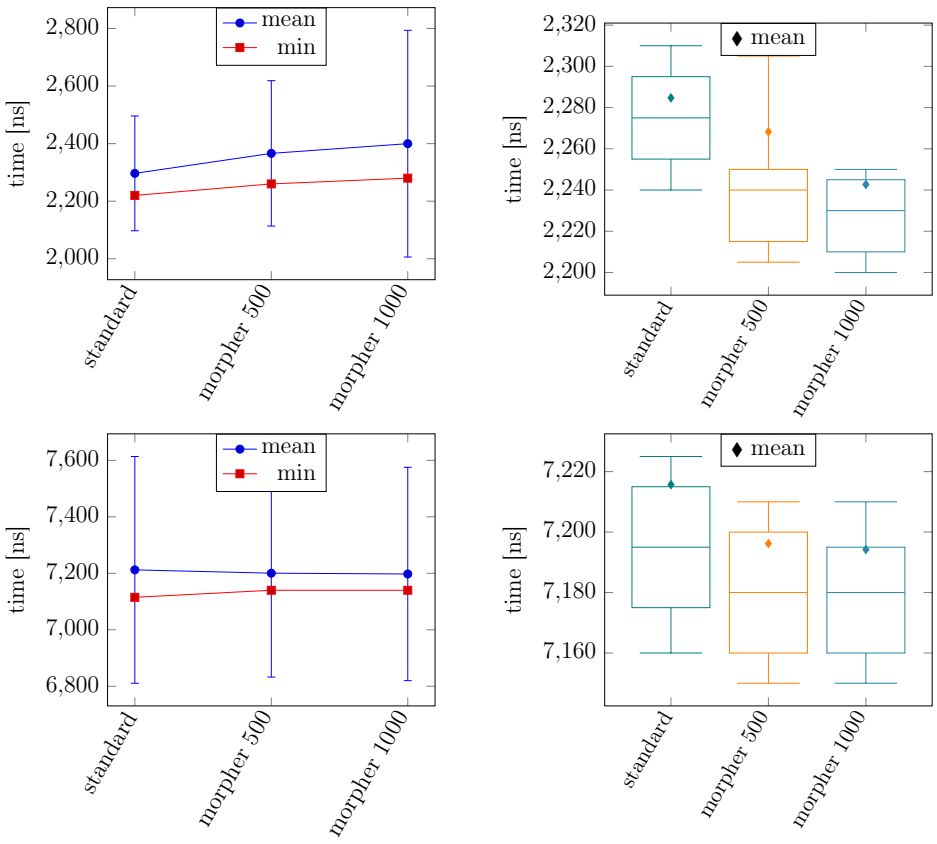

Figura 4.10: In alto: Openssl_AES spostamento dello deviazione standard. In basso: Scholar_AES spostamento dello deviazione standard

Le figure 4.10 e 4.10 mostrano la distribuzione dei campioni raccolti nel caso di una ottimizzazione del compilatore standard (livello 2).

Ognuna delle due figure è composta da due grafici che presentano informazioni di natura statistica sui campioni.

Il primo grafico mostra la media e il minimo rilevato per il campione ed è evidenziato l'intervallo dell'errore standard; il secondo grafico è una rappresentazione a boxplot del campione.

Per quest'ultimo sono state generate la posizione della mediana, del 25-esimo e 75-esimo quartile e i whiskers posizionati al 10-esimo e 90-esimo quartile. Per ognuno dei grafici boxplot è stata calcolata la media aritmetica

del campione che nel grafico è rappresentata con la forma di rombo.

Per definizione sappiamo che l'errore standard è pari alla metà della deviazione standard e cresce con il quadrato della differenza tra il campione e la media.

Dall'analisi nel primo grafico e in particolare della posizione del minimo rispetto sia all'estremo inferiore dell'intervallo generato dall'errore standard che alla posizione della media, si intuiscono sia la concentrazione del campione che la natura di un errore standard così ampio.

Non essendoci, ovviamente, valori al di sotto del minimo, l'ampiezza dell'intervallo dell'errore standard è dovuta a valori maggiori della media che potremmo definire fuori scala.

Il secondo grafico mostra ancora più in dettaglio la concentrazione dei valori del campione nel range mediana-media; infatti la media si trova in prossimità del 75-esimo quantile e la posizione del 90-quantile è prossima, mostrando quindi una compattezza nei valori assunti dal campione attorno alla mediana.

4.4 Stima overhead

Il capitolo si conclude con questa sezione che riassume l'analisi degli overhead dei moduli del framework rispetto al tempo della funzione.

La seguente tabella mostra i risultati degli overhead, nel caso di ottimizzazione massima, del valore medio di uno dei moduli del framework rispetto al tempo medio stimato di esecuzione della funzione AES corrispondente.

4.4.1 Tabelle di overhead

Il dato espresso identifica il rapporto tra il tempo medio di esecuzione della funzione di un modulo (morpher, shuffler) rispetto al tempo della funzione AES. Per esempio un valore di 45 per il modulo morpher, implica che la funzione del modulo ha un tempo di esecuzione pari a 45 volte il tempo della funzione AES. Le tabelle mostrano i dati al variare delle architetture hardware e dell'algoritmo di implementazione dell'AES.

	openssl_morpher		scholar_morpher_shuffler			
	m1000	m2500	m1000	s1000	m2500	s2500
Armv5	137.6	125.5	88.3	26	84.9	25.1
Armv7a	54.9	53.3	54.6	16.1	56.3	15.2
Armv7h	45	43.7	58.7	13.3	60	12.7

Tabella 4.1: Overhead dei tempi di esecuzione dei moduli rispetto ai tempi della sola funzione AES, rispettivamente per openssl e scholar

Nelle figure seguenti i dati sono suddivisi in colonne in base, o al livello di ottimizzazione di compilazione, o alla frequenza di utilizzo dei moduli; tali dati mostrano il rapporto tra i tempi di esecuzione dei moduli (init,morpher,shuffler) e il tempo di esecuzione della funzione AES. Il range dei valori rilevati va da 1 fino a valori positivi pari a 100 e più volte il tempo della funzione AES.

A seguito di un'ispezione visiva è stato possibile individuare, per alcune parametrizzazioni, delle caratteristiche o peculiarità ricorrenti nei dati dei campioni rilevati. Si parla di variazioni consistenti dal 10% al 40%

del tempo di esecuzione rispetto al tempo rilevato per altri campioni con parametrizzazioni differenti.

Sono state quindi individuate le seguenti combinazioni di parametro/modulo: *function_AES_m2500*, *morpher_m100*, *morpher_m2500*; la tabella 4.4.1 ne riassume le evidenze.

hardware	sample	aes	morpher	
		m2500	m100	m2500
armv7h	opensslmorpher	+ 4/6		- 6/6
	scholarmorphershuffler		+ 4/6	
armv7a	opensslmorpher	+ 6/6		- 5/6
	scholarmorphershuffler		+ 4/6	
armv5tel	opensslmorpher	+ 6/6		- 6/6
	scholarmorphershuffler		+ 4/6	

Tabella 4.2: Riepilogo peculiarità

I simboli + o - individuano la differenza dei tempi, maggiori o minori, rispetto agli altri campioni; il rapporto (ad esempio 4/6) rappresenta il numero di campioni per i quali questa peculiarità è stata riscontrata.

Una certa attenzione va utilizzata nella lettura del rapporto 4/6, perché corrisponde sempre al campione compilato con i livelli migliori di ottimizzazione, quindi rappresenta il campione più significativo dal punto di vista applicativo.

La differenza tra i livelli di ottimizzazione della compilazione genera invece una suddivisione del campione analizzato in due gruppi, distinguibili facilmente per la sostanziale differenza dei tempi di esecuzione. Da una parte si trovano le ottimizzazioni loop, size, o2 e o3 e dall'altra o1 e ovviamente debug. Il campione corrispondente al livello di ottimizzazione debug, inutile dal punto di vista applicativo, è stato lasciato nel campione dei risultati per poter fornire un termine di paragone per il risultato ottenuto dall'opzione o1 e ottenere una più accurata stima dell'importanza delle ottimizzazioni.

4.4.2 Overhead modulo Morpher

L'intervallo di overhead [minimo, massimo] rilevato per il modulo morpher è mostrato nella tabella 4.3 ed è limitato alle opzioni di compilazione più comuni.

I campioni con valori peggiori di overhead rilevato hanno tempi circa 1.5 volte peggiori del valore migliore rilevato. Queste discrepanze, all'interno di campioni relativi allo stesso tipo di hardware e algoritmo, dipendono in questo caso solo dal valore del parametro di frequenza utilizzato e sono la "cartina tornasole" delle peculiarità analizzate nella sezione precedente e riassunte nella tabella 4.4.1.

hardware	sample	o1	o2, o3
armv7h	opensslmorpher	116.2 - 127	43.7 - 46.8
	scholarmorphershuffler	24.0 - 26.5	55.9 - 78.6
armv7a	opensslmorpher	133.3 - 138.8	53.3 - 58.0
	scholarmorphershuffler	29.9 - 30.9	53.7 - 85.0
armv5tel	opensslmorpher	267.7 - 286.2	125.5 - 141.6
	scholarmorphershuffler	37.6 - 40.2	84.9 - 131.9

Tabella 4.3: Riepilogo peculiarità

4.4.3 Overhead modulo Shuffler

Il costo, presunto basso in fase di design dello shuffling, si attesta su valori pari a 10 volte la funzione AES, con risultati peggiori in caso di scarsa ottimizzazione, con 25 e più volte il costo della funzione AES (a sua volta già con tempi di esecuzione di almeno 2 volte maggiori).

I dati riguardanti il valore di overhead rilevato per il modulo di shuffling sono riportati nella tabella 4.4 in forma di intervallo [minimo, massimo] del rapporto tra i tempi di esecuzione.

hardware	o1	o2,o3
armv7h	22.9 - 23.2	12.5 - 14.0
armv7a	33.9 - 35.8	14.9 - 16.1
armv5tel	44.8 - 45.4	25.1 - 26.0

Tabella 4.4: Overhead del modulo di shuffle per i campioni scholar_morpher_shuffler

4.4.4 Analisi campione Pandaboard

L'analisi degli overhead per la Pandaboard ES, con quanto espresso nella sezione 4.3.4, è soggetta all'errore di stima dovuto alla granularità della sorgente di clock, manifestata dalla presenza nel campione rilevato di tempi di esecuzione pari a zero. Rispetto a questo problema evidenziato dai campioni rilevati, solo i campioni corrispondenti al modulo del morpher per parametrizzazioni o1_m500 e o1_m1000 presentano valori rilevati sempre diversi da zero.

Grazie alla presenza di questi due campioni significativi e al comportamento del tutto simile dei rapporti tra funzione e moduli rispetto alle altre due architetture hardware, è possibile considerare significativi i rapporti rilevati per la Pandaboard ES.

4.4.5 Parametri Consigliati

In [Agosta et al., 2012] si è proceduto alla stima dell'overhead computazionale dell'intero sistema, accettabile nel caso di un impiego industriale. Si è scelto un livello di protezione accettabile e quindi il numero di iterazioni minime dell'algoritmo di morphing, con un costo computazionale complessivo minore, tale da mantenere il livello di sicurezza pressoché invariato.

I dati raccolti mostrano che, per un intervallo pari a 2000-3000 iterazioni della funzione per ogni iterazione del morphing, il livello di resistenza del sistema da un attacco di DPA rimaneva attorno al 79% e l'overhead computazionale totale del sistema si aggirava tra il 20% e il 27%.

Code Morphing Interval	Confidence Intervals Overlap	Average Execution Time
1000	78.98	x1.46
2000	79.76	x1.27
3000	79.04	x1.20
4000	75.16	x1.17

Tabella 4.5: Estratto tabella 1 in [Agosta et al., 2012]: impatto dell'intervallo di morphing sui tempi di esecuzione e sulla protezione da attacchi di DPA

La seguente tabella è invece stata compilata con l'obbiettivo di fornire una mera stima dei tempi di esecuzione con questa implementazione per il range di intervallo di attivazione del framework 1000-2500, compatibilmente quindi con quanto previsto per i livelli di protezione.

Si ricorda che in questo lavoro non sono state eseguite analisi sulla protezione fornita dal morphing all'esecutore hardware nei confronti di attacchi DPA.

La tabella è stata costruita in modo teorico con i dati empirici della tabella 4.1, applicando una formula che stima il costo totale a partire dal costo dei singoli moduli (morpher e shuffler), comprendendo il costo medio della funzione AES.

$$overhead_{total} = 1 + \frac{\sum_{m \in modules} m_{overhead}}{interval}$$

	openssl_morpher		scholar_morpher_shuffler	
Architecture	1000	2500	1000	2500
Armv5	1.138	1.050	1.114	1.044
Armv7a	1.055	1.021	1.071	1.029
Armv7h	1.045	1.017	1.072	1.029

Tabella 4.6: Stima del tempo di esecuzione per gli intervalli 1000 e 2500

Capitolo 5

Conclusioni ed Estensioni

5.1 Conclusioni

Il lavoro ha realizzato l'obbiettivo di integrare in un framework, alla portata dello sviluppatore, delle metodologie per applicare delle contromisure agli algoritmi crittografici contro attacchi di tipo power analysis.

L'integrazione e l'esecuzione in fase di verifica di dispositivi con instruction set moderne sono state realizzate garantendo, allo stesso tempo, anche il supporto al codice modificabile a runtime per le piattaforme che eseguono instruction set ARMv5.

L'introduzione di una fase di inizializzazione che svolge il precalcolo di strutture dati complesse alleggerisce e semplifica il lavoro svolto dal morpher; in questa direzione si può ottenere un risultato ancora migliore aggiungendo una fase di analisi del codice binario, con l'obiettivo di individuare le istruzioni sostituibili lasciando in fase di esecuzione solo la sostituzione casuale. Nel caso più pragmatico di *random instruction insertion*, cioé dell'introduzione di istruzioni che aggiungono solo rumore di sottofondo, questo approccio non sarebbe però migliorativo.

È immaginabile il porting verso esecutori meno potenti come micro-controllori con delle scelte e delle soluzioni, per quanto riguarda il morphing, differenti rispetto a quanto possibile su ARM, visto il costo computazionale del morphing. Per esempio nella sezione 5.2.3 è presente un approccio che sposta a

compile time la gran parte dei costi.

Questo approccio ha come naturale evoluzione una soluzione ancora più verticale e, dovendo comunque fare affidamento su una non banale modifica ad-hoc del compilatore, è immaginabile, a questo punto, la creazione di una libreria integrata nel compilatore sviluppata per lo stesso scopo.

5.2 Possibili Estensioni

Le seguenti modifiche all'implementazione corrente del framework sono state individuate in fase di sviluppo e si possono considerare come idee e spunti di partenza la cui fattibilità è stata analizzata.

5.2.1 Modulo di Shuffling

Una delle possibili estensioni del modulo di shuffling è a carico della parte implementata in Clang del gestore dell'attributo. Una delle prime modifiche potrebbe consistere nell'introduzione nel compilatore della verifica di applicabilità dell'attributo di shuffling al codice del metodo. Ricordiamo velocemente che l'attributo opera su tre parametri: i) il nome della funzione su cui è attiva la contromisura, ii) il nome della variabile locale al metodo o ad un suo sotto-blocco, iii) il nome della variabile globale che contiene le permutazioni dei suoi indici. Sarebbe quindi utile introdurre la verifica delle condizioni di applicabilità per la variabile locale durante la compilazione, modificando quindi il gestore dell'attributo in LLVM per verificare se la variabile locale al metodo è usata come variabile di induzione in un ciclo e se è usata nel blocco solo per definire degli offset per un accesso ad array.

L'attuale implementazione accetta una sola variabile come variabile di induzione, mentre, a fronte di una modifica aggiuntiva, il modulo dovrebbe poter operare su diversi cicli con variabili di induzione differenti.

5.2.2 Parallelizzazione

La funzione *substitute*, di sostituzione del codice binario, potrebbe essere eseguita in un thread lanciato dallo stesso processo, ottenendo così un impatto significativo sul costo computazionale e sulla generazione del rumore. L'applicazione di questo approccio dovrebbe essere un'opzione di avvio/compilazione, perché è naturalmente applicabile nelle architetture multiprocessore, ma è anche conveniente in quelle applicazioni non CPU-bound.

Il modulo di morphing acquisirebbe un'ulteriore responsabilità, dovendo gestire la tempistica con la quale si sostituisce alla funzione corrente una versione modificata, calcolata dal thread incaricato di eseguire in background il morphing della funzione di substitute.

Questo approccio permetterebbe anche di aprire la strada all'applicabilità di processi di morphing più raffinati e complessi, e di introdurre dell'ulteriore rumore nell'esecuzione del sistema, sia come potenza assorbita che come alterazione temporale.

5.2.3 Morphing Ricombinante

Immaginando di generare a runtime o anche a compile-time versioni differenti dello stesso macro blocco di codice, si potrebbe pensare di passare da un morphing ottenuto attraverso alterazione diretta ad un morphing di tipo ricombinante, che ricostruisce la funzione scegliendo tra blocchi di codice equivalenti.

Naturalmente una soluzione che preveda la generazione delle varianti di morphing a compile-time avrebbe il pregio di avere costi computazionali molto bassi a runtime. D'altra parte, nonostante questo implichi un set limitato di alternative, queste bastano a generare sufficiente rumore da costringere un attacco di tipo DPA all'utilizzo di risorse computazionali eccessive per raccogliere le power trace e/o eseguire l'analisi delle stesse per ricostruire la chiave.

Tale soluzione potrebbe essere applicata a compile-time e lasciata gestire ad un nuovo modulo di morphing, che sarebbe informato attraverso una struttura dati delle varianti di codice e riuscirebbe a ricombinarle a runtime.

Questo equivale all'introduzione di un runtime environment per il modulo del morpher.

Essendo LLVM dotato di un runtime Jit, questa modifica potrebbe essere più semplice da realizzare se implementata facendo uso dello stesso Jit e quindi con gli strumenti messi a disposizione da LLVM.

Appendice A

Dati tecnici sulla CPU

PogoPlug Marvell

Figura A.1: PogoPlug SO2 Cpuinfo

```
processor        : 0
model name       : Feroceon 88FR131 rev 1 (v5l)
BogoMIPS         : 1191.11
Features         : swp half thumb fastmult edsp
CPU implementer  : 0x56
CPU architecture : 5TE
CPU variant      : 0x2
CPU part         : 0x131
CPU revision     : 1

Hardware         : Marvell SheevaPlug Reference Board
Revision         : 0000
```

PandaBoard Omap 4460

Figura A.2: Pandaboard-ES Cpuinfo

```
Processor        : ARMv7 Processor rev 10 (v7l)
processor        : 0
BogoMIPS         : 695.32

processor        : 1
BogoMIPS         : 698.99

Features         : swp half thumb fastmult vfp edsp thumbee neon
                   vfpv3 tls
CPU implementer  : 0x41
CPU architecture : 7
CPU variant      : 0x2
CPU part         : 0xc09
CPU revision     : 10

Hardware         : OMAP4 Panda board
Revision         : 0020
```

ODroid-xu: Exynos 5

Figura A.3: ODroid-xu Cpuinfo

```
Processor : ARMv7 Processor rev 3 (v7l)
processor : 0
BogoMIPS  : 1785.85

processor : 1
BogoMIPS  : 3174.85

processor : 2
BogoMIPS  : 3174.85

processor : 3
BogoMIPS  : 1587.42

Features  : swp half thumb fastmult vfp edsp neon vfpv3
            tls vfpv4 idiva idivt

Hardware  : ODROIDXU
Revision  : 0000
```

Bibliografia

[7-cpu.com, 2014] 7-cpu.com (2014). 7-cpu.com: Marvel kirkwood 88f6281. `http://www.7-cpu.com/cpu/Kirkwood.html`.

[Agosta et al., 2012] Agosta, G., Barenghi, A., and Pelosi, G. (2012). A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 77–82, New York, NY, USA. ACM.

[Barenghi et al., 2010] Barenghi, A., Pelosi, G., and Teglia, Y. (2010). Improving first order differential power attacks through digital signal processing. In *Proceedings of the 3rd international conference on Security of information and networks*, pages 124–133. ACM.

[Bayrak et al., 2011] Bayrak, A. G., Regazzoni, F., Brisk, P., Standaert, F.-X., and Ienne, P. (2011). A first step towards automatic application of power analysis countermeasures. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 230–235. IEEE.

[Biham and Shamir, 1991] Biham, E. and Shamir, A. (1991). Differential cryptanalysis of des-like cryptosystems. *Journal of CRYPTOLOGY*, 4(1):3–72.

[Coppersmith, 1994] Coppersmith, D. (1994). The data encryption standard (des) and its strength against attacks. *IBM journal of research and development*, 38(3):243–250.

[Curtin and Dolske, 1998] Curtin, M. and Dolske, J. (1998). A brute force search of des keyspace. In *8th Usenix Symposium, January*, pages 26–29. Citeseer.

[Daemen and Rijmen, 2002] Daemen, J. and Rijmen, V. (2002). *The design of Rijndael: AES-the advanced encryption standard*. Springer.

[EFF, 2014] EFF (2014). DES Cracker Machine. `https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html`.

[Gcc, 2014] Gcc (2014). Gcc: Optimization options. `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[Joye et al., 2005] Joye, M., Paillier, P., and Schoenmakers, B. (2005). On second-order differential power analysis. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 293–308. Springer.

[Kaliski and Robshaw, 1994] Kaliski, BurtonS., J. and Robshaw, M. (1994). Linear cryptanalysis using multiple approximations. In Desmedt, Y., editor, *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 26–39. Springer Berlin Heidelberg.

[kernel.org, 2014] kernel.org (2014). kernel.org: Documentation arm marvell. `https://www.kernel.org/doc/readme/Documentation-arm-Marvell-README`.

[Knuth, 1998] Knuth, D. E. (1998). The Art of Computer Programming, 3rd edn., vol. 2. *Seminumerical Algorithms*.

[Kocher et al., 1998] Kocher, P., Jaffe, J., and Jun, B. (1998). Introduction to differential power analysis and related attacks.

[Kocher et al., 1999] Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Advances in Cryptology CRYPTO 1999*, pages 388–397. Springer.

[Launchpad, 2014] Launchpad, U. (2014). Ubuntu launchpad: Pandaboard clock bug. `https://bugs.launchpad.net/linaro-ubuntu/+bug/873453`.

[Mangard et al., 2007] Mangard, S., Oswald, E., and Popp, T. (2007). *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer.

[marvell.com, 2014] marvell.com (2014). marvell.com: Marvell 88f6281 soc with sheeva technology. `http://www.marvell.com/embedded-processors/kirkwood/assets/88F6281-004_ver1.pdf`.

[Matsui, 1994] Matsui, M. (1994). Linear cryptanalysis method for des cipher. In *Advances in Cryptology—EUROCRYPT'93*, pages 386–397. Springer.

[Messerges et al., 1999] Messerges, T. S., Dabbish, E. A., and Sloan, R. H. (1999). Investigations of power analysis attacks on smartcards. In *USENIX workshop on Smartcard Technology*, volume 1999.

[numpy.org, 2014] numpy.org (2014). Numpy. `http://www.numpy.org`.

[Oswald et al., 2006] Oswald, E., Mangard, S., Herbst, C., and Tillich, S. (2006). Practical second-order dpa attacks for masked smart card implementations of block ciphers. In *Topics in Cryptology–CT-RSA 2006*, pages 192–207. Springer.

[Prouff et al., 2009] Prouff, E., Rivain, M., and Bévan, R. (2009). Statistical analysis of second order differential power analysis. *Computers, IEEE Transactions on*, 58(6):799–811.

[Quisquater and Samyde, 2001] Quisquater, J.-J. and Samyde, D. (2001). Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, pages 200–210. Springer.

[Rutenberg, 2014] Rutenberg, G. (2014). Profiling code using clock_gettime. `http://www.guyrutenberg.com/2007/09/22/profiling-code-using-clock_gettime/`.

[ti.com, 2014] ti.com (2014). ti.com: List of arm cores. `http://www.ti.com/lit/ds/symlink/omap4460.pdf`.

[Tillich and Großschädl, 2007] Tillich, S. and Großschädl, J. (2007). Power analysis resistant aes implementation with instruction set extensions. In *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 303–319. Springer.

[wikipedia.org, 2014a] wikipedia.org (2014a). Wikipedia: Arm9e. `http://en.wikipedia.org/wiki/ARM9E#ARM9E`.

[wikipedia.org, 2014b] wikipedia.org (2014b). Wikipedia: List of arm cores. `http://en.wikipedia.org/wiki/List_of_ARM_cores`.