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

TITOLO ITALIANO

TITOLO INGLESE

MARCO BURACCHI

Relatore: Prof. *Michele Boreale*

Anno Accademico 2017-2018

INDICE

ELENCO DELLE FIGURE

LISTINGS

ELENCO DELLE TABELLE

"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 trasforma con l'utilizzo di una chiave, 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 (figura 1) che riceveva un plaintext e restituiva un ciphertext (encryption) e viceversa (decryption). 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,

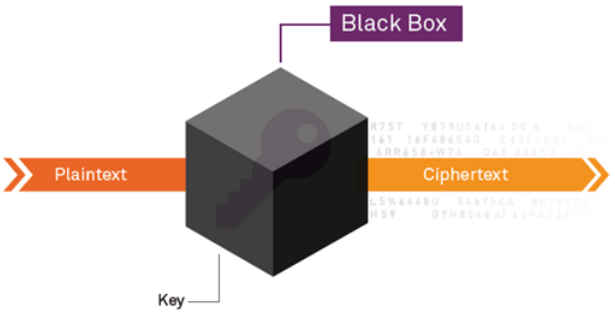

Figura 1: Black-box encryption

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 l'attacco *SPECTRE* che ha afflitto tutti i recenti processori AMD, ARM e Intel. Nel capitolo 4 verrà infine presentato un attacco che sfrutta i concetti dell'attacco SPECTRE in grado di ottenere dati protetti da password senza conoscere tale informazione.

1

SIDE-CHANNEL ATTACKS

I *side-channel attacks* sono metodi di criptanalisi 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. Da una parte troviamo il dispositivo che implementa la funzione crittografica con accanto lo strumento utilizzato per rilevare le informazioni che trapelano dal dispositivo attaccato. La cosa fondamentale è che gli attacchi di questo tipo non vanno a colpire direttamente la funzione crittografica ma sfruttano le informazioni fisiche dell'ambiente intorno al dispositivo.

L'analisi di questi metodi ha acquisito notevole interesse dato che questo tipo di attacchi possono essere montati velocemente e molto spesso non richiedono hardware particolare e 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 però molto variegata e disomogenea quindi, come prima cosa, cerchiamo 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.

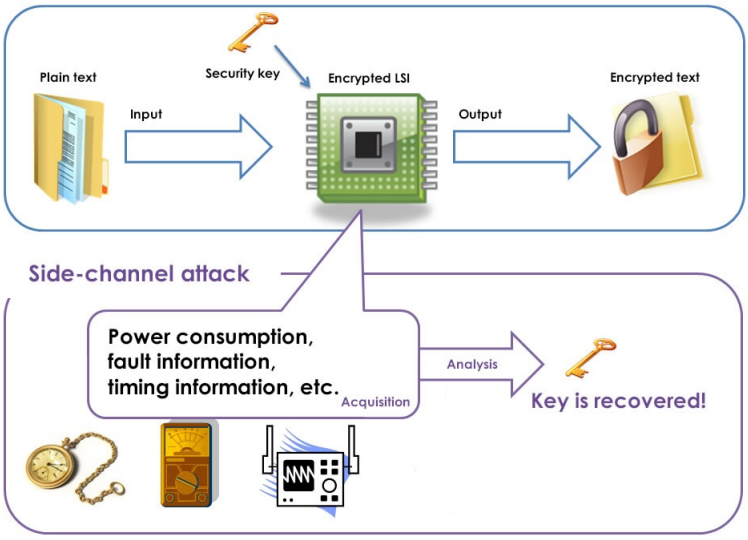

Figura 2: Esempio di side-channel attack

Tipi di canali

Nel lavoro di *Ge, Yarom, Cock e Heiser*[5] vengono fornite alcune definizioni che utilizzeremo nel prosieguo di questa tesi. La prima distinzione che è necessario 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 quelli creati e sfruttati dall'attaccante ad esempio tramite l'utilizzo di Trojan e che *deliberatamente* lasciano filtrare le informazioni. 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 si prestano maggiormente rispetto ad altre.

Il tempo e il consumo energetico sono le più comunemente utilizzate ma non sono di 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

Un'ultima classificazione può essere effettuata andando a discriminare gli attacchi secondo l'algoritmo crittografico attaccato. In questo caso i due maggiori algoritmi attaccati sono senza dubbio Advanced Encryption Standard (AES)[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 con i criteri sopra definiti i principali attacchi conosciuti.

Passiamo adesso ad una breve panoramica sui maggiori attacchi per ogni tipo di grandezza fisica attaccata. 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 '900, il governo degli Stati Uniti d'America era a conoscenza del fatto che i computer producessero "emanazioni compromettenti" e che queste potessero essere catturate ed utilizzate. I ricercatori dell'esercito dimostrarono che era possibile catturare queste emanazioni a distanza e rivelare le informazioni (classificate) ad esse associate. In relazione a questo problema fu attivato il progetto Transient Electromagnetic

Pulse Emanation STandard (TEMPEST).

TEMPEST è uno standard creato dal National Communications Security Committee Directive 4 (NCSCD4) ed i requisiti richiesti alle periferiche TEMPEST-compliant sono specificati nel documento riservato NACSIM5100A. Anche la North Atlantic Treaty Organization (NATO) 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 una televisione modificata con 15 dollari di accessori (al cambio attuale corrisponderebbero a meno di 40 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 di suoni prodotti da dispositivi meccanici, specialmente in campo militare, risalgono 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 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

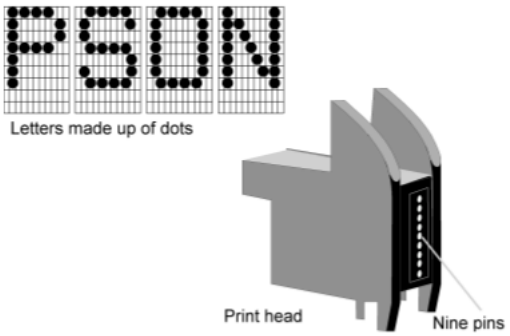

Figura 3: Modalità di stampa di una stampante ad aghi

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 sia ormai sparito dall'utilizzo del privato cittadino non deve far pensare ad un attacco anacronistico. Questa scelta è infatti dovuta al fatto che in quell'anno circa il 60% dei medici in Germania e il 30% delle banche utilizzavano ancora quel tipo di stampanti. 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 provoca 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 4 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, 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 Centre National

d'Etudes Spatiales (CNES) 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 Picosecond Imaging Circuit Analysis (PICA)[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.

Un ulteriore problema è dovuto proprio a questa ultima osservazione insieme alla necessità di sincronizzare Optica con il codice che sta eseguendo il dispositivo. 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 Simple Power Analysis (SPA) e la Differential Power Analysis (DPA) 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 Data Encryption Standard (DES) 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 dati forniti in input. Le permutazioni di DES come le moltiplicazioni o le esponenziazioni di RSA sono vittime tipiche di questi attacchi.

Se ad esempio prendiamo 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

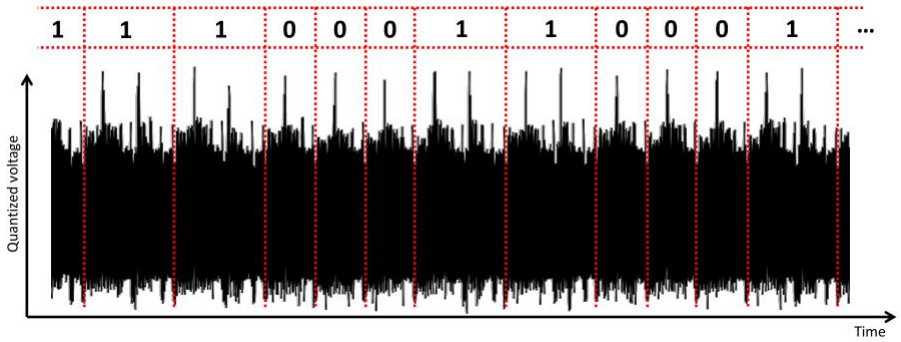

Figura 4: SPA contro RSA

esegue solamente la sequenza square-reduce. É 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 da possibili sorgenti di rumori e dall'altra può permettere di estrarre informazioni maggiori rispetto alla semplice esecuzione delle singole operazioni.

1.6 ATTACCHI BASATI SULLA TEMPERATURA

Di attacchi basati sulla temperatura se ne parla molto nella letteratura[32, 33, 34] ma la maggior parte delle pubblicazioni sull'argomento menzionano l'esistenza e la possibilità di sfruttare questo canale senza approfondire l'argomento. 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 su 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 modificando maliziosamente la normale esecuzione di un algoritmo crittografico. Tale esecuzione errata può far trapelare informazioni che possono essere utilizzate per recuperare la chiave. In generale gli attacchi fault-based si dividono nelle seguenti quattro categorie[37].

Differential Fault Analysis (DFA)

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. Lo stato dell'arte degli attacchi DFA permette di recuperare l'intera chiave a 128 bit di AES con l'inserimento di un singolo fault[38].

Fault Sensitivity Analysis (FSA)

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 .

Differential Fault Intensity Analysis (DFIA)

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.

Safe-Error Attacks (SEA) e Differential Behavior Analysis (DBA)

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 Fault Behavior Analysis (FBA) tecnica introdotta da *Li et al.* in [43]. Questi attacchi osservano solamente il comportamento del dispositivo crittografico durante l'iniezione dei fault e non richiedono di conoscere il valore del ciphertext.

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 e 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.

2

TIMING ATTACKS

Come anticipato nel capitolo precedente, approfondiremo adesso la tipologia di attacchi basati sul tempo focalizzandoci maggiormente su quelli che hanno come obiettivo la cache del processore.

L'idea di base che sta sotto i timing attacks si basa sul fatto che l'esecuzione di un determinato programma, al variare delle operazioni che vengono eseguite e al variare degli input, impiega tempi diversi per portare a termine il proprio compito.

```
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. Per portare

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

questo concetto al livello che ci interessa vediamo prima delle nozioni fondamentali sulla cache del processore.

2.1 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 [49], 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 vengono utilizzate le caches.

La cache è infatti un piccolo banco di memoria molto veloce sito all'interno di ogni core che il processore utilizza per immagazzinare i valori delle celle di memoria accedute più recentemente.

2.1.1 *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 Last Level Cache (LLC) o L3 condiviso tra i tutti i core (nella tabella 2 sono riportate le dimensioni delle varie memorie di alcuni processori).

Considerando che l'accesso alla memoria principale in media impiega dai 50 ai 150 *ns* mentre l'accesso alla cache L1 utilizza un tempo nell'or-

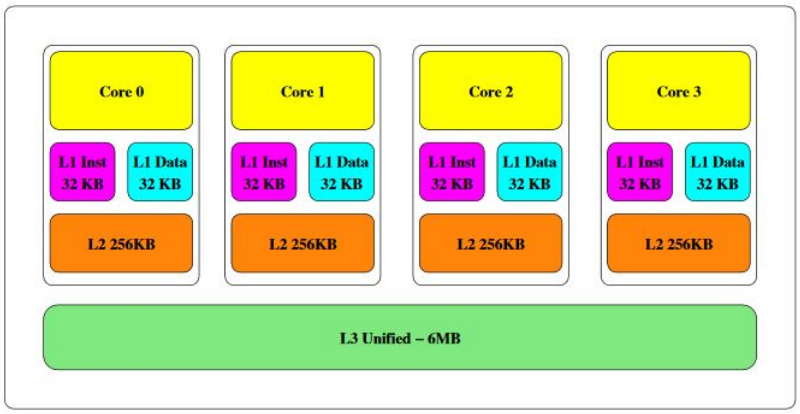

Figura 5: Architettura del processore Intel Core i5-3470

dine degli 0.3 *ns* si può capire l'enorme differenza di prestazioni che possono essere raggiunte utilizzando questo tipo di memoria.

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 2) 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 dalla 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 e i cache miss si hanno solamente quando non c'è più spazio libero nella cache. In pratica però questo 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 2 e 24 lines) in cui ogni indirizzo viene controllato in parallelo come in una fully-associative cache. In quale set viene mappato un blocco di memoria viene calcolato come per una direct-mapped cache da una funzione del suo indirizzo. Una cache con n line sets viene chiamata *n-way associative*.

In figura 6 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.

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 in cache, 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 linee 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 dal livello direttamente inferiore. Per mantenere questa caratteristica,

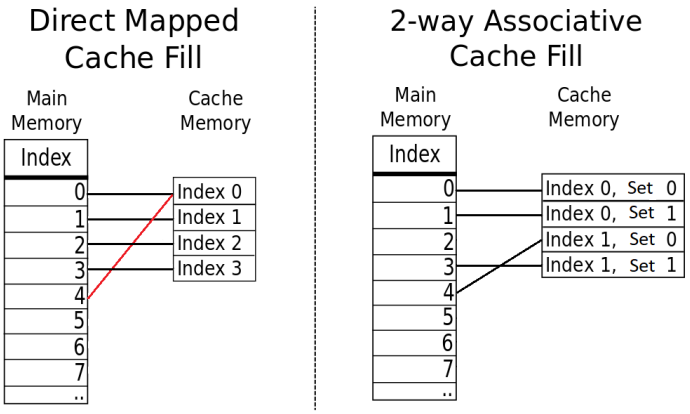

Figura 6: Schemi di associatività della cache.

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.

2.2 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 acceduto possono verificarsi una di queste due condizioni:

1. Il dato è presente in cache, si verifica una hit e viene recuperato molto velocemente.
2. Il dato non è presente in cache, si verifica una miss e bisogna aspettare che venga recuperato dalla memoria principale.

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

2.2.1 *Tassonomia*

Una prima classificazione dei cache attacks si basa sullo stato della cache al momento dell'attacco[50].

- *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.

- *Cold start misses*: questo tipo di miss si ottiene quando il dato viene acceduto 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 che quindi non possono essere presenti 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*[51]: Questo è un attacco di tipo forged initial state. L'attaccante precarica uno o più set della cache con dati propri. Dopo l'esecuzione della funzione vittima prova a riaccedere ai propri dati. Se la funzione vittima non ha utilizzato lines 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 ha utilizzato la vittima. Lo schema di questo attacco e del seguente è visibile in figura 7.
- *Evict+Time*[51]: 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. Con questa base, l'attaccante fa eseguire la funzione alla vittima calcolandone il tempo di esecuzione. Successivamente esegue una evict di un cache set caricando dati

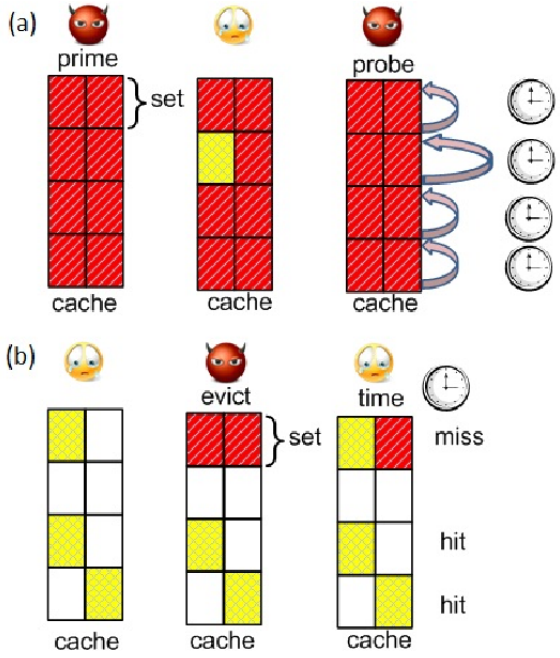

Figura 7: Schema di attacco Prime+Probe (a) e Evict+Time(b)

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 linea a cui è interessato utilizzando l'istruzione *clflush*[52] 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 linea che aveva rimosso. Se la risposta è veloce vuol dire che la vittima l'ha portata in cache durante l'esecuzione della sua funzione.

- *Evict+Reload*[53]: 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*[54]: Diversamente da tutti i precedenti approcci, in questo caso non si esegue nessun accesso alla memoria ma 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 in cache o meno.

2.3 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 [51].

2.3.1 *AES*

AES è un algoritmo di cifratura a blocchi, a chiave simmetrica, scelto come standard dagli Stati Uniti d'America dalla Federal Information Processing Standards (FIPS) nel documento PUB 197[58]. Viene considerato a tutti gli effetti il successore di DES, in via di abbandono a causa della sua ormai provata insicurezza[55, 56]. La rimanente spiegazione dell'algoritmo è presa da [57].

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 4 byte) ricavando 44 word dai 128 bit di partenza. Ad ogni round verranno prese di volta in volta 4 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:

- *Substitute byte*: tramite una tabella (*S-Box*) vengono sostituiti, byte per byte, tutti i byte del blocco.

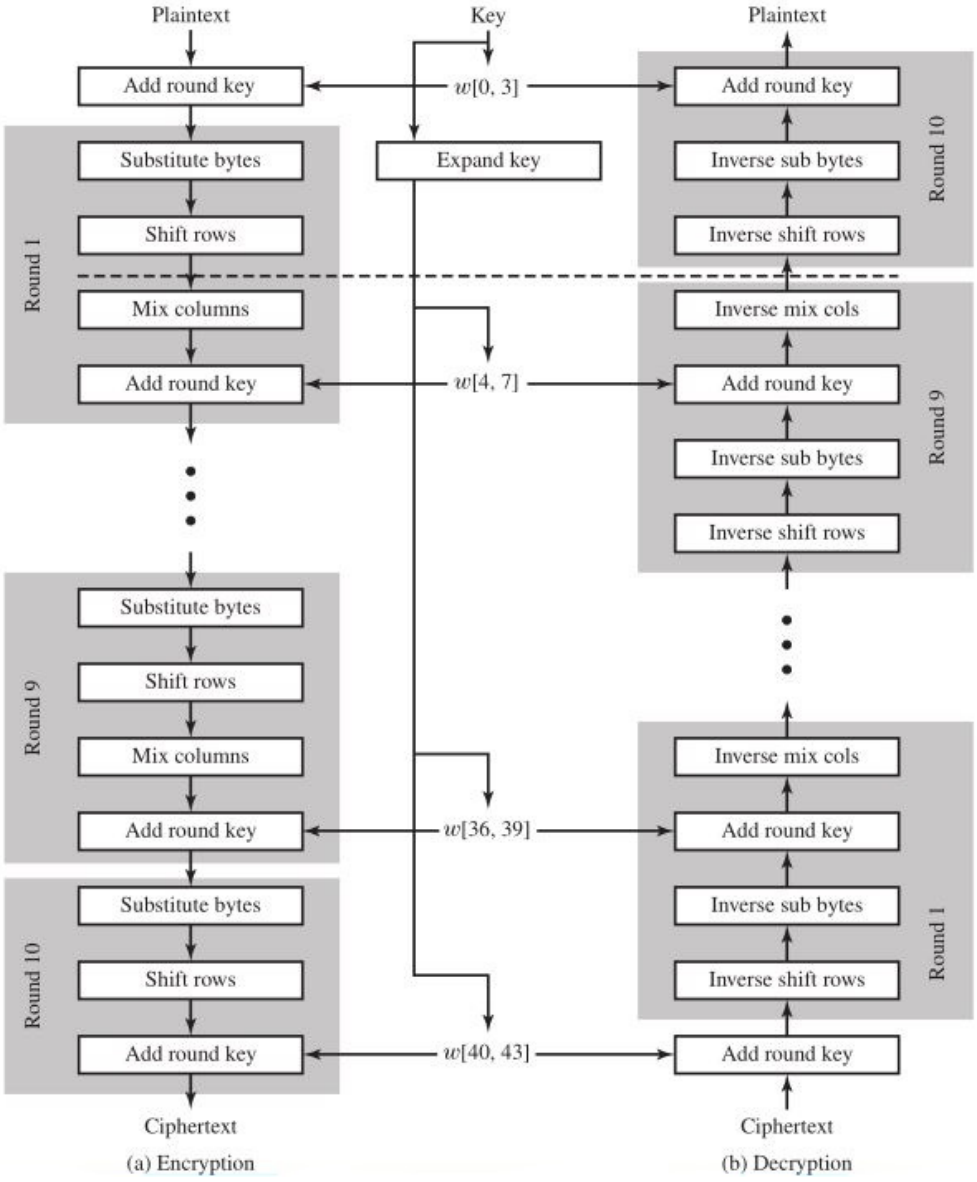

Figura 8: Schema di funzionamento di AES

- *Shift row*: viene effettuata una permutazione delle righe della matrice
- *Mix columns*: vengono modificate le colonne con una funzione su ogni bytes di ogni colonna.
- *Add round key*: viene effettuato uno XOR tra il blocco corrente e la chiave di round.

Come si vede in figura 8 l'algoritmo inizia con un *add round key* seguito da nove round in cui vengono effettuate tutte e quattro le operazioni. L'ultimo round (il decimo) è composto da solo tre di queste in quanto non viene effettuata la *Mix columns*.

2.3.2 *Implementazione e uso della memoria*

Quello descritto fino a qui è il funzionamento teorico di AES; in effetti, in teoria, tutto l'algoritmo è eseguibile tramite semplici operazioni algebriche ma in realtà, per migliorare le performance, vengono create (dal programmatore o durante l'inizializzazione del sistema) otto tabelle di lookup che chiameremo $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ 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 4 byte. La chiave (16 byte) $\mathbf{k} = (k_1, \ldots, k_{16})$ viene estesa per formare 10 chiavi (una per ogni round) $\mathbf{K}^{(r)}$ per $r = 1, \ldots, 10$ che vengono suddivise in 4 word di 4 byte ciascuna $\mathbf{K}^{(r)} = \left(K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)}\right)$. Dato un plaintext di 16 byte $\mathbf{p} = (p_0, \ldots, p_{15})$ la funzione di encryption calcola uno stato intermedio $\mathbf{x}^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)})$ ad ogni round r. Lo stato iniziale $\mathbf{x}^{(0)}$ viene calcolato come $x_i^{(0)} = p_i \oplus k_i$ con $(i = 0, \ldots, 15)$. I nove round seguenti vengono calcolati aggiornando lo stato intermedio secondo le seguenti equazioni (per $r = 0, \ldots, 8$):

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

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 $\mathbf{x}^{10}$ 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, solamente in questo, l'operazione *mix columns* non viene eseguita.

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.

2.4 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 cercare 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 a dati casuali dalla 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.*[59] hanno dimostrato che la correzione a tempo costante adottata per mitigare l'attacco *Lucky 13*[60] 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[61]. Questa soluzione, in teoria, riesce a risolvere completamente il problema ma è stato dimostrato[59] 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 [62] 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 uno dei quali è lo svuotamento completo della cache ad ogni context switch (*cache flushing*[63]). Questo ovviamente porta ad una perdita in prestazioni molto grande e si è passati al *lattice scheduling*[64] che esegue il flushing della cache non ad ogni context switch ma solo nel passaggio da processi sensibili a processi inaffidabili. Un'altra soluzione[65] mira a sfruttare la necessità di analizzare molto spesso lo stato della cache della vittima. Tale necessità è richiesta dagli attacchi di tipo Prime+Probe ad esempio e non viene soddisfatta imponendo un tempo minimo di esecuzione per le componenti vulnerabili entro il quale non possono essere prelazionate.

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[66] 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[67] 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).

3

SPECTRE ATTACKS

Figura 9: Il logo di SPECTRE

In questo capitolo verrà presentato il progetto *SPECTRE*[19] (il cui logo è rappresentato in figura 9), 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 Common Vulnerabilities and Exposures (CVE) 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.

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

3.1 ESECUZIONE SPECULATIVA

L'esecuzione speculativa è una tecnica utilizzata dai processori per ottenere un miglioramento delle prestazioni; essa consiste nel cercare di "indovinare" il risultato di un 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 10 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 angolo 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 del 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, si risparmiano molti cicli di clock mentre in caso di insuccesso il risultato

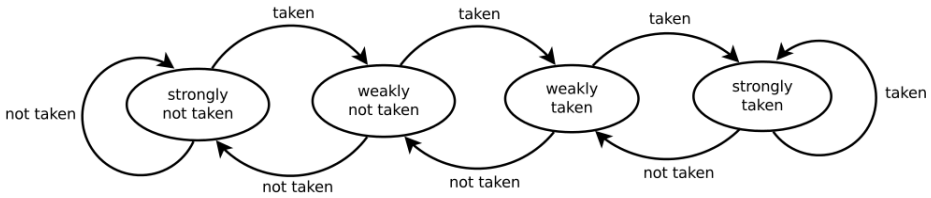

Figura 10: Automa di decisione di un one-level branch predictor a 2 bit

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 Branch Predictor (BP).

3.1.1 *Branch predictor*

Esistono svariati tipi di branch predictor; analizziamo il funzionamento di uno dei più semplici, il *one-level branch predictor* a 2 bit.

Come da schema in figura 10 un one-level branch predictor può essere descritto con un semplice automa a 4 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 2.

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 1 ma se si rivelerà true andrà allo stato 3 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 2, altrimenti passeremo al 4.

4. *Strongly taken:* questo stato è il duale dello stato 1. In questa situazione il BP sceglierà il ramo then rimanendo in questo stato se otterrà un true e tornando allo stato 3 se otterrà un false (continuando comunque ad eseguire il ramo then).

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 un certo branch.

3.2 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 esporrano 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.

3.2.1 *Esempio*

Consideriamo il caso di una funzione che riceve un intero x da una fonte non fidata (come ad esempio il codice 3.1).

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

Codice 3.1: 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 maniera malevola in maniera 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 11).
- *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 in maniera normale 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 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

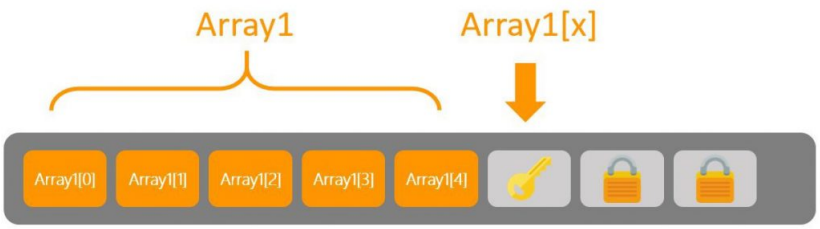

Figura 11: Recupero del valore al di fuori di *array1*

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 $n \in (0 \dots 255)$. 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.

3.3 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 [68, 69], 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 3.2.

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

Codice 3.2: 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 3.3 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];
```

Codice 3.3: 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[68].

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[70].

Utilizzare variabili "volatile"

Un'altra possibile contromisura può essere quella di utilizzare variabili dichiarate *volatile* come nel codice 3.4:

```
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];
```

Codice 3.4: Utilizzo di variabili dichiarate volatile

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

Ovviamente anche questo metodo risolve il problema ma, come 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 3.5:

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

Codice 3.5: 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.

4

PROOF OF CONCEPT

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque

felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

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, 22, 25, and 30.)

[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 13.)

[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:18o1.o1203*, 2018. (Cited on pages 11 and 32.)

[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, 22, and 26.)

[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, 25, and 27.)

[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 Brouchier, Tom Kean, Carol Marsh, and David Naccache. Temperature attacks. *IEEE Security & Privacy*, 7(2):79–82, 2009. (Cited on page 16.)

[33] Julien Brouchier, 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 16.)

[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 17.)

[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 18.)

[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 18.)

[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 18.)

[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 18.)

[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 18.)

[49] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. (Cited on page 21.)

[50] Anne Canteaut, Cedric Lauradoux, and Andre Seznec. *Understanding cache attacks*. PhD thesis, INRIA, 2006. (Cited on page 24.)

[51] 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 25 and 27.)

[52] Intel Intel. and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part*, 1(64):64, 64. (Cited on page 26.)

[53] 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 26.)

[54] 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 27.)

[55] 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 27.)

[56] John Gilmore. Cracking des: Secrets of encryption research, wiretap politics & chip design, 1998. (Cited on page 27.)

[57] William Stallings, Lawrie Brown, Michael D Bauer, and Arup Kumar Bhattacharjee. *Computer security: principles and practice*. Pearson Education, 2012. (Cited on page 27.)

[58] NIST FIPS Pub. 197: Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441):0311, 2001. (Cited on page 27.)

[59] 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 page 30.)

[60] 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 30.)

[61] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992. (Cited on page 30.)

[62] 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 31.)

[63] 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 31.)

[64] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. (Cited on page 31.)

[65] 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 31.)

[66] Colin Percival. Cache missing for fun and profit, 2005. (Cited on page 31.)

[67] 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 31.)

[68] 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 pages 37 and 38.)

[69] 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 37.)

[70] Paul Kocher. "spectre mitigations in microsoft's c/c++ compiler, 2018. (Cited on page 38.)

ACRONIMI

AES Advanced Encryption Standard

BP Branch Predictor

CNES Centre National d'Etudes Spatiales

CVE Common Vulnerabilities and Exposures

DBA Differential Behavior Analysis

DES Data Encryption Standard

DFA Differential Fault Analysis

DFIA Differential Fault Intensity Analysis

DPA Differential Power Analysis

FBA Fault Behavior Analysis

FIPS Federal Information Processing Standards

FSA Fault Sensitivity Analysis

GPS Global Positioning System

HTTPS HyperText Transfer Protocol over Secure Socket Layer

LED Light Emitting Diode

LLC Last Level Cache

NATO North Atlantic Treaty Organization

NCSCD4 National Communications Security Committee Directive 4

PICA Picosecond Imaging Circuit Analysis

QIF Quantitative Information-flow Analysis

SEA Safe-Error Attacks

SPA Simple Power Analysis

TEMPEST Transient Electromagnetic Pulse Emanation STandard

TOR The Onion Router

USB Universal Serial Bus

INDICE ANALITICO