

Tecniche di Protezione del SW - Papers

Lorenzo Naturale

5 luglio 2025

Indice

1 Low-Fat Pointers (LFP)	3
1.1 Definizione del Metodo	3
1.2 Confronto tra AddressSanitizer e Low-Fat Pointers	3
1.3 Risultati Sperimentali	3
1.4 Codice: LFP instrumentation	3
1.5 Conclusioni	4
2 Blind Return Oriented Programming (BROP)	5
2.1 Osservazioni chiave	5
2.2 Fasi dell'attacco	5
2.3 Automazione con Braille	5
2.4 Contromisure proposte	5
3 Control-Flow Integrity (CFI)	6
3.1 Principi e Implementazione	6
3.2 Applicazioni e Vantaggi	6
3.3 Lavoro Correlato	6
3.4 Codice: CFI Instrumentation	6
3.5 Conclusioni	7
4 Counterfeit Object-Oriented Programming (COOP)	8
4.1 Introduzione	8
4.2 Principi di COOP	8
4.3 Vulnerabilità delle Difese Esistenti	8
4.4 Implementazione e Dimostrazioni	8
4.5 Difese contro COOP	8
4.6 Conclusioni	9
5 LibAFL	10
5.1 Introduzione	10
5.2 Problema Affrontato	10
5.3 Obiettivo del Framework	10
5.4 Architettura e Caratteristiche	10
5.5 Tecniche Avanzate Supportate	11
5.6 Esperimenti e Valutazioni	11
5.7 Conclusione	11
6 Meltdown	12
6.1 Introduzione	12
6.2 Meccanismo di Attacco	12
6.3 Caratteristiche Principali dell'Attacco	12
6.4 Implicazioni di Sicurezza	12
6.5 Soluzioni e Mitigazioni	12
6.6 Meltdown PseudoCodice	13

6.7	Conclusioni	13
7	Spectre	14
7.1	Introduzione	14
7.2	Meccanismo di Attacco	14
7.3	Varianti dell'Attacco	14
7.4	Implicazioni di Sicurezza	14
7.5	Soluzioni e Mitigazioni	15
7.5.1	Approcci Software	15
7.5.2	Approcci Hardware	15
7.5.3	Limitazioni	15
7.6	Spectre PseudoCodice	15
7.7	Conclusioni	15

1 Low-Fat Pointers (LFP)

Il presente lavoro introduce un metodo innovativo per la protezione dei limiti dello stack mediante l'impiego dei *Low-Fat Pointers* (LFP). Questo approccio mira a mitigare vulnerabilità comuni legate alla corruzione della memoria, come i *buffer overflow*, mantenendo un impatto minimo sulle prestazioni del sistema.

1.1 Definizione del Metodo

I Low-Fat Pointers (LFP) rappresentano una tecnica avanzata di gestione della memoria, che incorpora le informazioni sui limiti (*bounds*) direttamente nei puntatori stessi. A differenza delle soluzioni tradizionali, che utilizzano tabelle di metadati esterne per tracciare tali informazioni, i LFP codificano i dati di bound direttamente nella rappresentazione binaria del puntatore. Questo approccio consente di rilevare accessi fuori dai limiti in modo efficiente, con un impatto minimo sulle prestazioni del sistema, riducendo sia il sovraccarico di memoria sia il tempo di accesso ai dati.

1.2 Confronto tra AddressSanitizer e Low-Fat Pointers

AddressSanitizer (ASan) è una tecnica che utilizza una *shadow memory* per rilevare errori di accesso alla memoria, come *buffer overflow* e *use-after-free*. ASan inserisce metadati in una memoria separata e strumenta il codice per eseguire controlli durante ogni accesso alla memoria. Sebbene sia estremamente efficace, introduce un elevato overhead in termini di prestazioni e utilizzo di memoria.

Al contrario, i Low-Fat Pointers incorporano direttamente le informazioni sui limiti all'interno dei puntatori stessi, eliminando la necessità di strutture di metadati esterne. Questo consente verifiche di bound più rapide e meno costose computazionalmente. Inoltre, i controlli vengono eseguiti mediante operazioni dirette sui puntatori, con un overhead nettamente inferiore rispetto ad ASan.

- **AddressSanitizer (ASan):** utilizza una *shadow memory* per tracciare lo stato della memoria e rilevare *buffer overflow* (BOF), *use-after-free* (UAF) e accessi illegali. Protegge la memoria con *redzones* e controlla ogni accesso, bloccando quelli non validi. Introduce un alto overhead in termini di memoria e prestazioni.
- **Low-Fat Pointers (LFP):** incorporano i bound direttamente nei puntatori, memorizzando l'indirizzo e la dimensione dell'oggetto. Consentono il controllo degli accessi senza l'uso di *shadow memory*, con un overhead minore. Tuttavia, richiedono modifiche al compilatore o supporto hardware specifico.

1.3 Risultati Sperimentali

Gli esperimenti condotti dimostrano che i LFP offrono una protezione efficace con un impatto minimo sulle prestazioni. Questa caratteristica rende la tecnica particolarmente adatta a sistemi in cui sicurezza e velocità costituiscono requisiti fondamentali, come nei sistemi embedded o critici in tempo reale.

1.4 Codice: LFP instrumentation

Un metodo efficace per prevenire errori di tipo **Out-Of-Bounds (OOB)** è l'instrumentazione dei controlli di limite → **bounds checking instrumentation**. L'idea di base è la seguente: dato un puntatore p associato a un oggetto O con indirizzo di base (base) e dimensione (size), allora p è fuori dai limiti rispetto a O se il seguente test (chiamato isOOB) ha esito positivo:

```
(p < base) || (p > base + size - sizeof(*p)) // isOOB
```

Gli errori di accesso fuori dai limiti possono essere prevenuti inserendo codice di controllo (strumentazione) per ogni operazione di lettura o scrittura di memoria che coinvolge p, come segue:

```
if (isOOB(p, base, size))
    error();
v = *p; // oppure *p = v;
```

Qui la funzione error() segnala l'errore di accesso fuori dai limiti e interrompe l'esecuzione del programma, prevenendo così qualsiasi attacco di tipo data/control o information leakage. Le informazioni sui limiti dell'oggetto, cioè la dimensione e la base, sono note come metadati dei limiti (→ **bounds meta information**) sono memorizzabili nel seguenti modi:

```
struct { void *ptr; void *base; size_t size; } // fat pointers

union { void *ptr;
        struct {uintptr_t size:10;
                 uintptr_t unused:54; } meta;} p; // low-fat pointers
```

1.5 Conclusioni

L'approccio basato sui Low-Fat Pointers rappresenta un significativo miglioramento rispetto alle soluzioni esistenti per la protezione dello stack. La tecnica offre un compromesso ottimale tra efficienza e sicurezza, con un ridotto overhead in termini di memoria e prestazioni. La capacità di codificare i bound direttamente nei puntatori permette una protezione leggera e performante, rendendo i LFP una soluzione interessante e promettente rispetto a metodi tradizionali come AddressSanitizer.

2 Blind Return Oriented Programming (BROP)

Il presente lavoro introduce la tecnica *Blind Return Oriented Programming* (BROP), che consente di scrivere exploit per vulnerabilità di *buffer overflow* (BOF) su server remoti senza disporre del codice sorgente né del file binario del target. Questo approccio è particolarmente utile contro servizi proprietari o binari compilati manualmente, in cui l'attaccante non ha accesso diretto all'eseguibile.

2.1 Osservazioni chiave

BROP si basa su due osservazioni fondamentali:

1. Alcuni server riavviano automaticamente i processi dopo un crash.
2. È possibile dedurre informazioni utili dal comportamento del server in risposta a input specifici (ad esempio: crash o assenza di crash).

2.2 Fasi dell'attacco

L'attacco sfrutta una vulnerabilità di *stack overflow* e si articola in tre fasi principali:

1. **Stack Reading:** lettura dello stack byte per byte, allo scopo di recuperare valori critici come il *stack canary*, il *frame pointer* e il *return address*, permettendo così di bypassare l'ASLR.
2. **Blind ROP:** individuazione remota di *gadget* utili alla costruzione di una *syscall write*, senza poter osservare direttamente la memoria o il codice.
3. **Dumping dell'eseguibile:** trasferimento del file binario dal server al client attraverso la *write syscall*, completando l'exploit con tecniche tradizionali in locale.

2.3 Automazione con Braille

Per automatizzare l'intero processo è stato sviluppato uno strumento chiamato *Braille*. Esso richiede unicamente una funzione in grado di inviare input al server e rilevare se quest'ultimo va in crash. L'attacco è stato eseguito con successo in tre scenari:

- **nginx** (CVE-2013-2028)
- **yaSSL + MySQL** (vulnerabilità di stack overflow)
- **Server proprietario** sviluppato da un collega degli autori

In media, un exploit completo è stato ottenuto con meno di 4.000 richieste e in circa 20 minuti. L'attacco è efficace anche contro sistemi moderni dotati di ASLR, NX e canary, a condizione che:

- il server sia *restartable*,
- l'ASLR non venga randomizzato nuovamente ad ogni crash.

2.4 Contromisure proposte

Il paper suggerisce alcune contromisure per mitigare o impedire l'efficacia dell'attacco:

- Compilare i binari con *Position Independent Executable* (PIE) e forzare la ri-randomizzazione dell'ASLR ad ogni crash del processo.
- Evitare overflow dello stack tramite una corretta validazione degli input.
- Impedire l'accesso remoto a informazioni sul crash o comportamento anomalo del server.

3 Control-Flow Integrity (CFI)

Il paper presenta il concetto di *Control-Flow Integrity* (CFI), una tecnica di sicurezza progettata per garantire che l'esecuzione di un programma segua un grafo di flusso di controllo (CFG) predeterminato. CFI ha l'obiettivo di prevenire attacchi che sfruttano vulnerabilità del software per alterare arbitrariamente il flusso di esecuzione, come i *buffer overflow* e gli attacchi *jump-to-libc*.

3.1 Principi e Implementazione

- **CFG (Control Flow Graph):** definito tramite analisi statica o dinamica, rappresenta i percorsi di esecuzione validi nel programma.
- **Assunzioni chiave:**
 - **UNQ (Unique IDs):** gli identificatori utilizzati per i salti di controllo devono essere univoci nel codice.
 - **NWC (Non-Writable Code):** il codice non deve poter essere modificato durante l'esecuzione.
 - **NXD (Non-Executable Data):** i dati non devono essere eseguibili come codice.
- **Strumentazione:** il codice viene riscritto per inserire controlli dinamici (ID-check) che verificano che ogni salto di controllo rispetti il CFG. Questi controlli vengono implementati tramite riscrittura del codice macchina.

3.2 Applicazioni e Vantaggi

- **Protezione:** CFI è efficace contro attacchi comuni come la sovrascrittura di puntatori e gli exploit basati sull'heap.
- **Efficienza:** il costo in termini di prestazioni è moderato, con un overhead medio del 16% nei benchmark SPEC.
- **Fondamento per politiche avanzate:** CFI può essere esteso per supportare meccanismi di sicurezza più sofisticati, tra cui:
 - *Shadow Call Stack*: una copia protetta dello stack per la verifica dei `return`.
 - *Software Memory Access Control* (SMAC): un sistema di controllo degli accessi alla memoria a livello software.

3.3 Lavoro Correlato

CFI si distingue da altre tecniche di mitigazione, come *stack canaries* o *ASLR*, per i seguenti motivi:

- **Semplicità e Verificabilità:** le garanzie di sicurezza offerte da CFI possono essere formalmente dimostrate.
- **Compatibilità:** può essere applicato a codice esistente senza necessità di modifiche hardware.
- **Robustezza:** offre una protezione efficace anche contro attaccanti con pieno controllo della memoria, impedendo la deviazione del flusso di esecuzione.

3.4 Codice: CFI Instrumentation

Questa tecnica protegge i salti indiretti confrontando un **identificatore unico (ID)** nella memoria di destinazione prima di effettuare il salto. Se l'ID non è quello atteso, viene generato un errore.

Vantaggi: *Sicurezza, Compatibilità e Precisione*

&

Svantaggi: *Overhead, Gestione degli ID, Interferenze con altre Tecniche*

Esempio applicazione tecnica CFI:

```
jmp ecx          ; computed jump
mov eax, [esp+4] ; dst
```

can be instrumented as:

```
cmp [ecx], 12345678h ; comp ID & dst
jne error_label      ; if != fail
lea ecx, [ecx+4]     ; skip ID at dst
jmp ecx              ; jump to dst

; data 12345678h ; ID
mov eax, [esp+4] ; dst
```

3.5 Conclusioni

CFI rappresenta una soluzione pratica ed efficace per garantire l'integrità del flusso di controllo di un programma, con un impatto contenuto sulle prestazioni. La sua semplicità, combinata con la possibilità di integrazione con altri meccanismi di protezione, lo rende un elemento fondamentale per migliorare la sicurezza del software moderno.

4 Counterfeit Object-Oriented Programming (COOP)

4.1 Introduzione

Il paper introduce *Counterfeit Object-Oriented Programming* (COOP), una tecnica di attacco che sfrutta il riutilizzo di codice nelle applicazioni C++ per eludere meccanismi di protezione del controllo di flusso. COOP dimostra che molte difese, tra cui la *Control-Flow Integrity* (CFI), risultano inefficaci se non tengono conto della semantica orientata agli oggetti tipica del linguaggio C++.

4.2 Principi di COOP

- **Vfgadgets:** COOP utilizza funzioni virtuali esistenti (*Virtual Function Call Gadgets*) come blocchi di codice riutilizzabili. Queste funzioni sono invocate attraverso siti di chiamata legittimi, rendendo l'attacco difficile da rilevare. Sebbene originariamente pensate per il polimorfismo, possono essere abusate da un attaccante per deviare il flusso di esecuzione.
- **Oggetti contraffatti:** L'attaccante crea oggetti C++ falsificati, dotati di puntatori a tabelle virtuali (vptr) manipolati. Quando una funzione virtuale viene chiamata su questi oggetti, il flusso di esecuzione può essere rediretto verso codice arbitrario.
- **Flusso di dati:** I dati sono trasferiti tra i *vfgadgets* attraverso la sovrapposizione degli oggetti contraffatti o tramite registri e stack, simulando il comportamento lecito del programma.
- **Esempio:** In C++, quando si chiama una funzione virtuale su un oggetto, il programma consulta la *vtable* per determinare l'indirizzo della funzione. Se un oggetto di tipo A ha il proprio *vptr* sovrascritto con quello della classe B, una chiamata a `doSomething()` potrebbe in realtà invocare `openShell()`.

4.3 Vulnerabilità delle Difese Esistenti

- **Control-Flow Integrity (CFI):** Le implementazioni CFI che ignorano la semantica orientata agli oggetti del C++ non riescono a distinguere tra chiamate virtuali legittime e malevoli.
- **Shadow Call Stack:** COOP non altera direttamente gli indirizzi di ritorno, rendendo inefficaci le difese basate sul controllo dello stack.
- **Randomizzazione del codice:** COOP elude anche tecniche di randomizzazione fine-grained, poiché fa uso di intere funzioni virtuali anziché brevi sequenze di istruzioni.

4.4 Implementazione e Dimostrazioni

- **Esempi pratici:** Gli autori hanno realizzato exploit funzionanti per Internet Explorer 10 (32 e 64 bit) e Firefox 36 su Linux.
- **Turing completezza:** Gli autori dimostrano che COOP è Turing completo, consentendo la costruzione di payload arbitrariamente complessi.

4.5 Difese contro COOP

- **CFI consapevole del C++:** Tecniche che analizzano la gerarchia delle classi e i siti di chiamata virtuale possono mitigare COOP, sebbene richiedano accesso al codice sorgente o avanzate analisi binarie.
- **Verifica dei vptr:** Controllare che i *vptr* puntino a *vtable* valide e non manipolate può prevenire l'attacco.
- **Memory safety:** Garantire l'integrità spaziale e temporale della memoria è una contromisura efficace, anche se spesso accompagnata da un elevato overhead prestazionale.

4.6 Conclusioni

COOP mette in luce le limitazioni delle difese attuali contro il riutilizzo di codice, in particolare nelle applicazioni C++. La tecnica è potente, stealthy, e in grado di bypassare protezioni come CFI, shadow stacks e randomizzazione fine-grained. Gli autori sottolineano l'importanza di sviluppare difese consapevoli della semantica del C++ e propongono soluzioni basate su analisi binaria avanzata o accesso al codice sorgente. I risultati del paper contribuiscono significativamente al dibattito sulla sicurezza del software moderno.

5 LibAFL

5.1 Introduzione

Il paper presenta **LibAFL**, un framework open-source scritto in Rust progettato per costruire fuzzers modulari, riutilizzabili e altamente estendibili. LibAFL nasce per superare i limiti degli strumenti di fuzzing esistenti, spesso basati su fork non compatibili di AFL (American Fuzzy Lop), che ostacolano la combinazione, il confronto e il riutilizzo delle tecniche accademiche.

5.2 Problema Affrontato

La comunità di ricerca sul fuzzing ha prodotto centinaia di tecniche innovative, spesso implementate in fork separati e incompatibili. Ciò ha portato a tre principali problemi:

1. Le tecniche ortogonali sono difficili da combinare.
2. È complesso valutare l'impatto isolato di una singola tecnica.
3. I confronti tra approcci diversi risultano poco equi e difficili da eseguire.

5.3 Obiettivo del Framework

LibAFL propone un'architettura componibile e configurabile per la creazione di fuzzers. Ogni componente essenziale del processo di fuzzing è modellato come un'entità astratta e indipendente.

Entità Principali in LibAFL

- **Input:** Rappresentazione interna degli input (es. sequenze di byte, AST).
- **Corpus:** Insieme di input “interessanti” trovati durante il fuzzing.
- **Scheduler:** Strategia di selezione dei test case dal corpus.
- **Stage:** Operazioni applicate a un test case (mutazioni, minimizzazione, tracciamento).
- **Observer:** Raccoglie informazioni sull'esecuzione del target (es. coverage).
- **Executor:** Gestisce l'esecuzione del programma target con l'input fornito.
- **Feedback:** Decide se un input è interessante in base agli osservatori.
- **Mutator:** Applica mutazioni agli input esistenti.
- **Generator:** Crea nuovi input da zero (es. approcci model-based).

5.4 Architettura e Caratteristiche

LibAFL è composto dai seguenti moduli principali:

- **LibAFL Core:** Implementazione delle entità base. Supporta ambienti senza libreria standard.
- **LibAFL Targets:** Strumenti per la strumentazione runtime (es. coverage).
- **LibAFL CC:** Wrapper per compilatori come Clang per integrare strumentazione automatica.
- **LibAFL Sugar:** API di alto livello per facilitare la costruzione di fuzzers.

Il framework è progettato per essere:

- **Estensibile:** Ogni componente è implementato come trait generico in Rust.
- **Portabile:** Funziona anche in ambienti embedded o bare-metal.
- **Scalabile:** Include un event manager per la comunicazione tra nodi paralleli.

5.5 Tecniche Avanzate Supportate

1. Roadblock Bypassing

- **Value-profile:** Massimizza i bit corrispondenti in confronti binari.
- **Cmplog:** Registra i valori nei confronti per abilitarne la sostituzione mirata.
- **Autotokens:** Estraе stringhe chiave dal binario per usarle come dizionario.

2. Structure-aware Fuzzing

- **Nautilus:** Grammar-based fuzzer che opera su AST.
- **Grimoire:** Impara la struttura degli input a partire dai test case osservati.
- **Token-level fuzzing:** Mutazioni effettuate a livello di token lessicale.

3. Corpus Scheduling

- **MinimizerScheduler:** Seleziona seed minimali con copertura massima.
- **WeightedScheduler:** Campionamento probabilistico basato su metriche.
- **AccountingScheduler:** Priorizza input ad alto impatto in termini di sicurezza.

4. Energy Assignment (Power Schedules)

- **AFLFast Schedules:** Include `coe`, `fast`, `explore`, `exploit`, `lin`, `quad`.
- **Plain:** Numero fisso di mutazioni.
- **Risultati:** Gli scheduler `explore` e `fast` sono i più efficaci su target a bassa latenza.

5.6 Esperimenti e Valutazioni

Gli autori hanno testato LibAFL su benchmark da FuzzBench, valutando 15 tecniche e combinazioni originali.

- **Bit-level generic fuzzer:** Combinando `cmplog`, `weighted scheduler`, `explore` e `M0pt`, ha superato AFL++, Honggfuzz e LibFuzzer in 22 benchmark.
- **Differential Fuzzing:** Implementazione di NeoDiff per Ethereum VM con feedback basato su hash dello stato.

5.7 Conclusione

LibAFL è una piattaforma potente, flessibile e modulare per lo sviluppo di fuzzers avanzati. Si distingue per:

- La capacità di combinare facilmente tecniche ortogonali.
- La possibilità di valutazioni e confronti equi su base comune.
- Il supporto sia per la ricerca accademica che per l'uso industriale.

Grazie alla sua architettura componibile e all'elevato grado di personalizzazione, LibAFL apre nuove opportunità nel campo della ricerca sul fuzzing e nella sicurezza del software in generale.

6 Meltdown

6.1 Introduzione

Il paper presenta **Meltdown**, una vulnerabilità hardware che consente a un processo in user space di accedere a memoria protetta del kernel, un comportamento teoricamente impossibile. Scoperta nel 2018, Meltdown ha messo in luce gravi problemi nella gestione dell'esecuzione speculativa nelle CPU moderne. L'esecuzione speculativa è una tecnica di ottimizzazione in cui la CPU esegue in anticipo istruzioni basate su predizioni. Se la predizione si rivela errata, le istruzioni vengono annullate. Tuttavia, esse possono lasciare tracce nello stato microarchitetturale del sistema, come la cache.

6.2 Meccanismo di Attacco

Meltdown sfrutta un comportamento delle CPU che implementano *out-of-order execution*, una tecnica in cui le istruzioni vengono eseguite nell'ordine più efficiente possibile, purché il risultato finale sia coerente con l'ordine previsto dal programma. Il meccanismo dell'attacco è il seguente:

1. Un processo tenta di leggere una cella di memoria protetta del kernel, operazione che normalmente causerebbe un'eccezione.
2. Prima che l'eccezione venga gestita, la CPU esegue speculativamente alcune istruzioni seguenti.
3. Durante questa finestra temporale, i dati protetti possono essere caricati nella cache L1.
4. Quando l'eccezione viene sollevata, le istruzioni speculative sono annullate, ma i dati restano nella cache.
5. Tramite un attacco side-channel, ad esempio *Flush+Reload*, l'attaccante può dedurre i dati speculativamente letti, basandosi sui tempi di accesso alla cache.

6.3 Caratteristiche Principali dell'Attacco

- Meltdown è efficace su molte CPU Intel, alcune ARM e un numero limitato di CPU AMD.
- Può essere eseguito da un qualsiasi processo non privilegiato.
- È in grado di leggere qualsiasi porzione della memoria del kernel mappata nello spazio virtuale del processo.

6.4 Implicazioni di Sicurezza

Meltdown compromette uno dei principi fondamentali della sicurezza dei sistemi operativi: la separazione tra user space e kernel space. Le implicazioni includono:

- Furto di password, chiavi di cifratura e altre informazioni sensibili contenute nella memoria del kernel.
- Violazione della protezione della memoria, consentendo accessi arbitrari al kernel da processi utente.

6.5 Soluzioni e Mitigazioni

Il paper propone diverse contromisure per mitigare la vulnerabilità, tra cui:

- **KAISER (Kernel Address Isolation to have Side-channels Efficiently Removed)**: tecnica che isola completamente lo spazio di memoria del kernel dallo user space. KAISER limita la mappatura del kernel nello spazio virtuale visibile ai processi utente, riducendo drasticamente la superficie d'attacco.
- **Aggiornamenti software e firmware**: sono fondamentali per applicare mitigazioni efficaci contro Meltdown.

Tuttavia, le soluzioni proposte hanno un impatto negativo sulle prestazioni, in particolare nei casi in cui sono frequenti le transizioni tra user space e kernel (es. system call intensive).

6.6 Meltdown PseudoCodice

```
1. probe = array(size = 256 * page_size)
2. flush_cache(probe)                      // flush 'probe' from cache
3. secret_byte = *(protected_address)      // causes exception
4. access probe[secret_byte * page_size]   // speculative execution
5. exception is handled (page fault)
6. for i = 0 to 255:
    measure access_time to probe[i * page_size]
    if access_time is low:
        likely secret value = i           // deduce the secret value
```

In pratica, anche se l'accesso alla memoria protetta fallisce, la CPU esegue speculativamente il codice successivo, lasciando tracce nella cache che possono essere misurate tramite side-channel.

6.7 Conclusioni

Meltdown dimostra che ottimizzazioni architetturali per le performance, come l'esecuzione speculativa e out-of-order, possono introdurre gravi vulnerabilità di sicurezza se non progettate tenendo conto delle possibili implicazioni sulla protezione dei dati. Questa vulnerabilità è tanto potente quanto semplice da sfruttare, e sottolinea come i problemi hardware possano avere conseguenze devastanti sulla sicurezza di interi sistemi software.

7 Spectre

7.1 Introduzione

Il paper presenta **Spectre**, una vulnerabilità che, come Meltdown, sfrutta meccanismi di esecuzione speculativa presenti nelle CPU moderne. Tuttavia, a differenza di Meltdown:

- Non viola direttamente la protezione hardware della memoria (es. separazione tra kernel e user space).
- Inganna un programma stesso a rivelare dati sensibili attraverso la sua esecuzione speculativa.
- È più generale e colpisce una gamma più ampia di processori, inclusi Intel, AMD e ARM.

7.2 Meccanismo di Attacco

Spectre si basa sull'inganno dei meccanismi di predizione dei salti (*branch prediction*) della CPU:

1. L'attaccante induce una vittima (di solito un processo) ad eseguire uno snippet di codice contenente un salto condizionale.
2. Tramite una tecnica detta *mistraining*, l'attaccante altera lo stato interno del predittore di ramo.
3. La CPU esegue speculativamente il ramo sbagliato, che accede a dati segreti che, in condizioni normali, non sarebbero accessibili.
4. Sebbene la speculazione venga annullata, i dati finiscono comunque nella cache.
5. L'attaccante può dedurre i dati tramite attacchi side-channel, come:
 - **Flush+Reload:** È un attacco side-channel che sfrutta la cache della CPU. L'attaccante "svuota" (flush) un blocco di memoria dalla cache, quindi osserva se un'altra parte del sistema "ricarica" (reload) quel dato. Se il dato è ricaricato, significa che è stato letto, permettendo all'attaccante di dedurre informazioni sensibili.
 - **Prime+Probe:** È un attacco side-channel in cui l'attaccante "prime" (riempie) la cache con i propri dati, quindi osserva se altri dati vengono "provati" (accessed) dalla vittima. Se i dati dell'attaccante vengono sostituiti nella cache, ciò indica che la vittima ha acceso la memoria contenente quei dati, permettendo la raccolta di informazioni.

7.3 Varianti dell'Attacco

Il paper identifica due principali varianti di Spectre:

- **Variant 1: Bounds Check Bypass (Spectre-BCB)**
Manipola il predittore di rami basato sulla *Pattern History Table (PHT)*, consentendo letture fuori dai limiti controllate dall'attaccante.
- **Variant 2: Branch Target Injection (Spectre-BTI)**
Manipola il *Branch Target Buffer (BTB)*, inducendo la CPU a saltare a indirizzi controllati dall'attaccante per eseguire codice speculativo malevolo (gadget).

7.4 Implicazioni di Sicurezza

Spectre è considerata più difficile da mitigare rispetto a Meltdown per vari motivi:

- È basata su comportamenti normali e leciti della CPU.
- Può essere eseguita tramite JavaScript in un browser, esponendo dati di altre tab o processi.
- Può colpire anche componenti privilegiati come kernel e hypervisor in scenari più sofisticati.

7.5 Soluzioni e Mitigazioni

7.5.1 Approcci Software

- Inserimento di barriere di memoria (*fence*) come LFENCE per impedire speculazioni indesiderate.
- Tecniche di **retpoline** per evitare salti speculativi a indirizzi controllati.
- Aggiornamenti ai compilatori per generare codice resistente agli attacchi Spectre.

Definizioni:

- *Fence*: istruzione che impone un ordine rigoroso nell'esecuzione delle istruzioni, bloccando la speculazione fino al completamento delle precedenti.
- *Retpoline*: tecnica che sostituisce salti indiretti con una sequenza di istruzioni che impedisce la speculazione dell'indirizzo di destinazione.

7.5.2 Approcci Hardware

- Alcuni vendor hanno aggiornato il microcode per introdurre controlli aggiuntivi nei predittori.
- Molte mitigazioni hardware richiedono però nuovi design architetturali, quindi sono applicabili solo a CPU future.

7.5.3 Limitazioni

- Le mitigazioni software e hardware possono comportare una riduzione significativa delle prestazioni.
- Non esistono soluzioni completamente risolutive per tutte le varianti note.

7.6 Spectre PseudoCodice

```
1. array1 = [x1, x2, x3, ..., xn]           // array of protected data
2. probe = array(size = 256 * page_size)      // array for measuring access time
3. flush_cache(probe)                       // flush 'probe' from cache
4. x = user_input                           // by user (potentially malicious)
5. if (x < array1_size):                   // bounds checking control
   temp = array1[x]                         // speculative read (branch mispredict)
   access probe[temp * page_size]          // read from cache
6. for i = 0 to 255:
   measure access_time to probe[i * page_size]
   if access_time is low:
     likely secret value = i               // deduce the secret value
```

In pratica, anche se l'accesso alla memoria protetta fallisce, la CPU esegue speculativamente il codice successivo, lasciando tracce nella cache che possono essere misurate tramite side-channel.

7.7 Conclusioni

Spectre ha rivelato come le ottimizzazioni per le performance nelle CPU (in particolare l'esecuzione speculativa) possano introdurre vulnerabilità complesse e pervasive. Contrariamente a Meltdown, Spectre non si basa su violazioni di protezione di memoria, ma sfrutta la stessa logica dei programmi e il loro comportamento lecito in condizioni speculativi. La vulnerabilità richiede una revisione profonda del design delle CPU e di tutto l'ecosistema software per garantire la sicurezza a lungo termine.