Sistemi Operativi Avanzati e Sicurezza dei Sistemi

Progetto A.A. 2025/2026

Report del Progetto:

SCT - Syscall Throttling Linux Kernel Module

0365258

Francesco Masci

Indice

I. Syscall Hooking

Per implementare le funzionalità di monitoraggio richieste, il primo passo fondamentale è l'intercettazione delle system call. Tramite questa tecnica, il modulo è in grado di porsi come intermediario nell'esecuzione delle chiamate di sistema effettuate dai thread, permettendo l'iniezione della logica di controllo.

Una volta stabilito l'hook, è necessario eseguire una serie di verifiche preliminari per determinare se l'esecuzione corrente debba essere sottoposta a throttling o lasciata proseguire indisturbata.

Per garantire flessibilità e compatibilità, il meccanismo di hooking è stato implementato seguendo due metodologie distinte, selezionabili dall'utente in fase di compilazione: la manipolazione diretta della *Syscall Table* e l'utilizzo del sottosistema *Ftrace*.

Indipendentemente dal metodo di iniezione scelto, il flusso di esecuzione viene deviato verso un wrapper comune che racchiude l'intera logica del monitor.

a. Syscall Table Discover Hooking

Questa metodologia prevede l'intercettazione tramite la manipolazione diretta della System Call Table. Il modulo esegue una scansione della memoria kernel a runtime (sfruttando *kprobes* su simboli come `__x64_sys_call` o simili) per localizzare l'indirizzo della tabella delle syscall.

Una volta individuata la tabella, l'indirizzo della funzione originale viene sostituito con quello del wrapper.

- **Vantaggi**: L'overhead a runtime è minimo, in quanto l'operazione consiste in un semplice salto indiretto modificato.
- **Svantaggi**: Questo approccio presenta criticità in termini di stabilità e portabilità. Su versioni recenti del kernel, i simboli necessari per la discovery e il patching non sono esportati, rendendo la patch complessa o impossibile se le probe vengono bloccate.

b. Ftrace Hooking

Per superare le limitazioni di stabilità e compatibilità del metodo precedente, è stata implementata una soluzione basata su *Ftrace*, il sottosistema standard del kernel Linux per il tracing delle funzioni.

Utilizzando le API di Ftrace, viene registrata una callback che intercetta l'esecuzione all'ingresso delle system call di interesse. All'interno della callback, viene modificato il registro Instruction Pointer nella struttura `pt_regs`, deviando il flusso di esecuzione verso il nostro wrapper.

Un dettaglio implementativo critico riguarda la gestione della ricorsione: quando il wrapper invoca la funzione originale, deve assicurarsi di saltare le prime istruzioni (corrispondenti alla dimensione del prologo di Ftrace, solitamente 5 byte su *x86_64*). Eseguire queste istruzioni causerebbe un nuovo trigger della callback Ftrace, innescando un loop infinito.

- **Vantaggi**: Questa tecnica è decisamente più stabile e sicura, basandosi su API supportate dal kernel, garantendo una maggiore compatibilità tra diverse versioni di Linux.
- **Svantaggi**: Introduce un overhead maggiore rispetto alla manipolazione diretta della tabella, dovuto alla gestione delle strutture interne di Ftrace e al salvataggio/ripristino dei registri necessari per la callback.

II. Filtri di Monitoraggio

Il cuore architetturale del modulo risiede nell'efficiente gestione dei filtri di monitoraggio. Una volta intercettato il flusso di esecuzione tramite l'hooking, il sistema deve determinare istantaneamente se la specifica syscall, invocata da un determinato utente per un dato programma, debba essere sottoposta a throttling.

Considerando che queste verifiche avvengono nel *critical hot-path* di ogni chiamata di sistema, la minimizzazione dell'overhead è il requisito non funzionale prioritario. Di seguito sono dettagliate le strutture dati e le ottimizzazioni adottate per ciascun criterio di filtro.

a. Utilizzo delle Primitive Kernel

Per garantire la massima stabilità e integrazione, l'implementazione delle strutture dati necessarie al filtraggio non è stata realizzata ex-novo, ma si basa interamente sulle primitive standard messe a disposizione dal kernel Linux.

Sviluppare strutture proprietarie in kernel space è una pratica rischiosa e spesso inefficiente; al contrario, l'adozione delle API native permette di sfruttare implementazioni *battle-tested*, altamente ottimizzate per la cache CPU e per la concorrenza (tramite operazioni atomiche e RCU).

b. Syscall Monitoring

Il primo livello di filtraggio determina se la syscall invocata è oggetto di interesse. Per evitare di degradare le prestazioni dell'intero sistema, è stata adottata una strategia di *Selective Hooking*.

Invece di intercettare indistintamente tutte le system call e filtrare internamente quelle non monitorate, l'hook viene installato sulla specifica entrata della *Syscall Table* solo nel momento in cui l'utente ne richiede esplicitamente il monitoraggio. Questa scelta architetturale garantisce che le syscall non monitorate mantengano le prestazioni native, non subendo alcuna deviazione del flusso di esecuzione.

All'interno del wrapper, per gestire eventuali race condition (es. una syscall che entra nel wrapper mentre il monitoraggio viene disabilitato), viene effettuato un controllo rapido su una Bitmap. La scelta della bitmap è dettata dalla natura statica e contenuta della tabella delle syscall: la verifica di un bit è un'operazione estremamente rapida e cache-friendly.

A livello di codice, questo controllo è annotato con la macro `unlikely()`, istruendo il compilatore a ottimizzare il flusso per il caso "monitoraggio attivo", dato che la disattivazione concorrente è un evento raro.

c. UID Monitoring

La verifica dell'EUID del processo chiamante pone sfide diverse rispetto alle syscall. Lo spazio dei possibili UID è vasto (potenzialmente 2^{32}) ma estremamente sparso (pochi UID attivi rispetto al totale). L'utilizzo di una bitmap sarebbe inefficiente in termini di memoria.

La soluzione implementata prevede l'utilizzo di una Hash Map.

L'EUID funge da chiave per l'inserimento e la ricerca. Le collisioni sono gestite tramite liste concatenate, ma dato che il confronto avviene tra valori interi primitivi, l'operazione di lookup risulta estremamente performante, mantenendo una complessità media vicina a $O(1)$.

d. Program Name Monitoring

Il monitoraggio basato sul percorso dell'eseguibile presenta criticità legate alla manipolazione delle stringhe (lentezza nei confronti) e alla persistenza (cambi di path o cancellazioni).

Per ovviare a questi problemi, il filtro non agisce sul path testuale, ma sulla coppia univoca *Inode* e *Device ID*.

All'interno del modulo, un programma monitorato è identificato da una chiave hash generata tramite la funzione kernel `jhash_2words`, prendendo in input l'inode number e il device ID. Questo approccio trasforma il problema del confronto di stringhe in un confronto tra interi all'interno di una Hash Map, garantendo elevate prestazioni.

- **Path Cache**: Di default, il modulo memorizza il path assoluto del file al momento della registrazione per scopi di logging. Tuttavia, è disponibile il flag di compilazione `LOW_MEMORY` che disabilita questo salvataggio, riducendo l'impronta di memoria per sistemi embedded o con risorse limitate.
- **Gestione File Cancellati**: Il modulo è progettato per non mantenere un lock persistente sull'inode (non incrementa il reference count), permettendo all'utente di cancellare o spostare il file monitorato. In caso di cancellazione, il file non sarebbe più raggiungibile tramite path per rimuovere il filtro. Per risolvere questo scenario, le API del modulo supportano una sintassi speciale `<inode>:<device>` che permette di rimuovere una regola di monitoraggio specificando direttamente gli identificativi numerici, bypassando la risoluzione del percorso.

e. Concorrenza e Sincronizzazione

Le strutture dati dei filtri sono soggette a letture frequentissime (ad ogni syscall) e scritture rare (modifiche alla configurazione da parte dell'amministratore). Questo è lo scenario ideale per l'utilizzo del meccanismo RCU.

- **Lettura**: Le operazioni di lookup nel wrapper sono protette da primitive RCU, che garantiscono l'accesso non bloccante e wait-free. Questo è cruciale per non introdurre latenza nelle system call.
- **Scrittura**: Le operazioni di modifica sono protette da write locks tradizionali. Sebbene più lente, la loro rarità rispetto alle letture rende il costo ammortizzato trascurabile.

Questa strategia assicura la coerenza dei dati senza penalizzare le performance del sistema durante il normale carico di lavoro.

Infine, per garantire la consistenza dei metadati statistici, come i contatori che tengono traccia del numero totale di oggetti monitorati, sono state impiegate variabili atomiche. L'uso di istruzioni atomiche permette di gestire incrementi e decrementi concorrenti in modalità non bloccante, assicurando l'esattezza del conteggio senza incorrere nell'overhead dei meccanismi di locking tradizionali.

III. Gestione della Finestra Temporale

L'intera logica di throttling del modulo è subordinata al concetto di finestra temporale. Il limite di esecuzione imposto dall'amministratore non è assoluto, ma relativo a un intervallo di tempo discreto. In conformità alle specifiche di progetto, la granularità temporale predefinita è fissata a 1 secondo, valore che può essere tuttavia modificato in fase di compilazione agendo sulle apposite macro di configurazione.

a. Implementazione del Timer

Il meccanismo di clock è gestito tramite un *Kernel Timer*. Il timer viene inizializzato e armato durante la fase di caricamento del modulo e configurato per auto-rigenerarsi ad ogni scadenza.

La rigenerazione del timer avviene sommando l'intervallo configurato al valore corrente di `jiffies`. Poiché la granularità del throttling è macroscopica, l'uso di timer standard basati sui `jiffies` offre un compromesso ideale tra precisione e basso overhead, evitando il costo computazionale dei timer ad alta risoluzione.

È cruciale notare che la funzione di callback associata al timer viene eseguita in contesto di *SoftIRQ*. In questo stato, il codice non può cedere la CPU né utilizzare primitive di sincronizzazione bloccanti come i Mutex. Per questo motivo, tutte le operazioni critiche di transizione descritte sono protette rigorosamente tramite RCU (o RWLock con irqsave), garantendo la coerenza dei dati senza violare i vincoli di atomicità del kernel.

La callback è responsabile delle seguenti operazioni:

- **Aggiornamento Statistiche**: Vengono consolidati i dati relativi alla finestra appena terminata invocando le routine di aggiornamento.
- **Reset Atomico**: Il contatore atomico che traccia il numero di invocazioni nella finestra corrente viene resettato a zero. Questa operazione è eseguita atomicamente per garantire che non vi siano interferenze con thread che stanno tentando di acquisire uno slot proprio nell'istante del cambio finestra.
- **Risveglio dei Thread bloccati**: Vengono risvegliati i thread bloccati nella finestra corrente per tentare di acquisire uno slot nella finestra successiva.

b. Strategia di Risveglio

Una volta azzerato il contatore e aperta ufficialmente la nuova finestra, il sistema deve gestire i thread precedentemente bloccati nella `waitqueue`.

Una implementazione ingenua prevederebbe l'utilizzo di `wake_up_all`, risvegliando indistintamente tutti i processi in attesa. Tuttavia, in scenari di alto carico in cui il numero di thread in coda supera il limite di slot disponibili per la nuova finestra, questo approccio causerebbe il cosiddetto *Thundering Herd Problem*: un numero eccessivo di thread verrebbe svegliato, competerebbe per le risorse, ma solo i primi riuscirebbero ad acquisire uno slot, costringendo la maggioranza a tornare immediatamente in stato di sleep. Questo

comportamento genererebbe un numero elevatissimo di context switch inutili, degradando le prestazioni del sistema.

Per mitigare questo fenomeno, il modulo adotta una politica di risveglio condizionato e limitato:

- **Risveglio Selettivo**: Il timer non sveglia l'intera coda, ma un numero di thread pari al limite di throughput configurato (gli slot disponibili). In questo modo, si garantisce che (in assenza di nuovi thread entranti) ogni thread svegliato abbia un'alta probabilità di procedere all'esecuzione.
- **Guardia sulla Wait Condition**: La primitiva di attesa utilizzata valuta una condizione booleana rigorosa prima di restituire il controllo al processo. Anche se un thread viene svegliato dal timer, esso verifica immediatamente se il contatore delle invocazioni correnti è ancora inferiore al limite massimo. Se nel frattempo nuovi thread (non provenienti dalla coda) hanno consumato tutti gli slot disponibili, la condizione risulterà falsa e il thread tornerà a dormire senza uscire dalla funzione di attesa.

Questo meccanismo implica un modello di competizione ibrido: i nuovi thread in arrivo competono con quelli risvegliati dalla coda. Sebbene ciò possa teoricamente permettere a un nuovo thread di "rubare" lo slot a un thread in attesa, in pratica massimizza l'utilizzo della CPU e il throughput complessivo, evitando l'overhead di code FIFO rigorose che richiederebbero meccanismi di locking più complessi.

IV. Raccolta delle Statistiche

In conformità ai requisiti di progetto, il modulo implementa un sottosistema dedicato alla raccolta di metriche prestazionali. L'obiettivo è fornire all'utente una visione chiara dell'impatto del monitoraggio sul sistema. Le metriche collezionate sono:

- **Peak Delay**: Il massimo ritardo introdotto su una singola system call dall'avvio del monitor.
- **Thread Blocking Stats**: Il numero medio e il picco massimo di thread bloccati per finestra temporale.

a. Peak Delay Tracking

La gestione del Peak Delay richiede una struttura dati dedicata in grado di memorizzare non solo il valore temporale del ritardo, ma anche il contesto in cui si è verificato: il numero della syscall, l'EUID dell'utente e il nome dell'eseguibile.

Poiché ogni thread monitorato calcola il proprio ritardo al termine dell'esecuzione, la struttura che ospita il massimo globale è soggetta a forte contesa in scenari ad alto carico. Per mitigare l'impatto sulle prestazioni, sono state implementate diverse strategie di ottimizzazione.

i. Ottimizzazioni Preliminari

Prima di tentare l'acquisizione di qualsiasi primitiva di sincronizzazione, vengono eseguiti due controlli leggeri:

1) **Noise Filtering**: Viene verificato se il ritardo osservato supera una soglia minima configurabile a compile time (default 0). Questo filtro evita che fluttuazioni microscopiche e trascurabili causino inutili tentativi di scrittura.
2) **Lockless Comparison**: Viene effettuato un confronto speculativo senza lock. Se il ritardo corrente del thread è inferiore al picco globale già registrato, l'operazione viene abortita immediatamente. Anche se la lettura del valore globale non è atomica in questa fase (potrebbe essere "stale"), la logica è comunque efficace: se il valore locale è minore di un vecchio picco, sarà sicuramente minore o uguale a un eventuale nuovo picco aggiornato da altri thread. Questo riduce drasticamente l'acquisizione dei lock.

ii. Strategie di Sincronizzazione

Il modulo supporta due strategie di aggiornamento mutuamente esclusive, selezionabili a tempo di compilazione:

- **RCU - Default**:
  Questa strategia è ottimizzata per scenari in cui la lettura delle statistiche è frequente e non deve mai essere bloccata dagli aggiornamenti.

Quando viene rilevato un nuovo picco, il sistema alloca un nuovo buffer, vi copia i dati aggiornati e scambia il puntatore in maniera atomica. L'allocazione avviene con flag GFP_ATOMIC per garantire la non-bloccabilità; in caso di fallimento dell'allocazione, l'aggiornamento viene scartato a favore della stabilità del sistema.
Sebbene garantisca letture *wait-free*, questo approccio comporta un overhead di allocazione dinamica per ogni aggiornamento del picco.

- **Spinlock - Low Memory Mode**:
  Pensata per sistemi embedded o con risorse limitate, questa modalità aggiorna la struttura dati in-place proteggendola con uno spinlock.
  Elimina la necessità di allocazioni dinamiche ricorrenti, riducendo drasticamente l'overhead sulla gestione della memoria e prevenendo la frammentazione.
  Introducendo però una sezione critica "forte", a differenza dell'approccio lockless per i lettori, qui l'aggiornamento è mutualmente esclusivo rispetto alle interrogazioni. Ciò riduce leggermente il parallelismo e, in scenari estremi, può causare latenza se lo scrittore deve attendere il completamento di operazioni di lettura concorrenti.
  Questa modalità può essere attivata manualmente o automaticamente se definita la macro LOW_MEMORY.

b. Time Window Statistics

La raccolta dati sui thread bloccati segue una logica basata su finestre temporali discrete. Il modulo mantiene sia il conteggio in tempo reale della finestra attiva, sia lo storico aggregato (totale thread bloccati e picco massimo registrato in una singola finestra).

i. Sincronizzazione dei Dati Aggregati

Per la gestione dei valori storici e non atomici, il modulo adotta le medesime strategie di sincronizzazione descritte per il Peak Delay (RCU o Spinlock).

Al termine di ogni finestra temporale, l'operazione di aggiornamento non si limita al semplice confronto con il picco storico. La sezione critica protegge infatti la modifica congiunta e atomica dell'intero set statistico necessario al calcolo delle medie e dei record:

- **Picco Massimo**: Viene verificato se il numero di thread bloccati nella finestra appena conclusa supera il record storico e, in caso affermativo, viene aggiornato.
- **Somma Cumulativa**: Il numero di thread bloccati viene sommato al totale globale.
- **Contatore Finestre**: Viene incrementato il numero totale di finestre osservate.

L'aggiornamento simultaneo di somma e conteggio sotto la protezione di RCU (o Spinlock) è indispensabile per garantire la consistenza matematica del dato: se questi valori venissero aggiornati separatamente senza lock, una lettura concorrente potrebbe prelevare una somma aggiornata e un conteggio vecchio (o viceversa), portando a un calcolo errato della media. Questo approccio garantisce che le statistiche globali rimangano coerenti anche in presenza di race condition dovute al reset delle finestre.

È però doveroso specificare che, nonostante la presenza di robusti meccanismi di sincronizzazione per garantire la coerenza dei dati, l'architettura del sistema è progettata per serializzare alla fonte l'evento di scrittura. La logica di chiusura della finestra temporale e il conseguente aggiornamento delle statistiche storiche sono demandati a un singolo flusso di esecuzione (il thread responsabile della gestione del timer). Di conseguenza, la contesa tra scrittori è architetturalmente azzerata, poiché non è possibile che più thread tentino simultaneamente di committare i dati relativi alla medesima finestra temporale.

ii. Gestione della Finestra Corrente

Al contrario, il contatore dei thread bloccati nella finestra corrente è gestito tramite una variabile atomica. Essendo l'unica metrica soggetta ad aggiornamenti ad altissima frequenza, l'uso di operazioni atomiche garantisce incrementi rapidi e lock-free, evitando il collo di bottiglia che si creerebbe acquisendo un lock per ogni singolo evento di throttling.

iii. Logica di Conteggio

Un aspetto cruciale riguarda la logica di conteggio: un thread viene conteggiato come *bloccato* solo al primo tentativo fallito all'interno di una specifica richiesta. Se il thread viene risvegliato, tenta nuovamente di acquisire uno slot e fallisce ancora (ritornando in `waitqueue`), questo secondo evento non incrementa il contatore. Questo approccio evita di gonfiare le statistiche con retries multipli dello stesso thread all'interno delle varie finestre, fornendo un dato fedele al numero reale di richieste uniche sottoposte a throttling.

iv. Calcolo della Media

Poiché il kernel Linux non supporta nativamente l'aritmetica in virgola mobile, al fine di evitare l'overhead legato al salvataggio del contesto e preservare le performance, il calcolo della media dei thread bloccati richiesto dalle specifiche è stato implementato utilizzando una tecnica di aritmetica a virgola fissa.

In questo approccio, viene fornito un "fattore di scala" (ad esempio 100 o 1000) che il kernel utilizza per moltiplicare il totale cumulativo dei thread bloccati prima di eseguire la divisione per il numero di finestre temporali. Il valore restituito è un numero intero che rappresenta la media scalata; spetta quindi all'utilizzatore dividere tale risultato per il fattore di scala originale per ricostruire la parte decimale corretta. Questa soluzione consente di soddisfare i requisiti di precisione mantenendo l'efficienza delle operazioni intere all'interno del kernel.

V. Syscall Wrapper

Il cuore operativo del modulo è rappresentato dalla funzione `syscall_wrapper`. Questa routine agisce come un proxy trasparente per tutte le chiamate di sistema intercettate: si interpone tra lo spazio utente e la funzione kernel originale, orchestrando la logica di filtraggio, il throttling e la raccolta delle metriche.

Essendo eseguita nel contesto di ogni syscall monitorata, l'efficienza di questo componente è critica. Il design è stato ottimizzato per minimizzare l'overhead, adottando percorsi di esecuzione rapidi per le chiamate che non richiedono throttling.

a. Reference Counting

All'ingresso della funzione, ancor prima di risolvere la syscall target, il wrapper incrementa atomicamente il contatore globale `active_threads`.

Questa operazione di *Reference Counting* è essenziale per tracciare il numero di thread attualmente "in volo" all'interno del codice del modulo; questo contatore funge da barriera di sicurezza: garantisce che le strutture dati interne non vengano deallocate finché vi sono thread attivi che le stanno utilizzando.

Al termine dell'esecuzione, indipendentemente dall'esito (successo, blocco o errore), il contatore viene decrementato atomicamente, notificando eventualmente se il conteggio scende a zero.

b. Identificazione e Filtraggio

Il wrapper recupera l'indice della system call originale dai registri della CPU e risolve l'indirizzo della funzione originale tramite la tabella interna.

Successivamente, viene eseguita una pipeline di filtraggio a cascata per decidere se applicare il throttling:

- **Global Status**: Verifica rapida se il monitoraggio è attivo.
- **Syscall Filter**: Controlla se l'indice della syscall è marcato per il monitoraggio.
- **Target Filter**: Verifica se l'utente o il programma correnti corrispondono alle regole configurate.

Se uno qualsiasi di questi controlli fallisce, il wrapper salta immediatamente alla fase di esecuzione, rendendo l'impatto prestazionale trascurabile per il traffico non monitorato.

c. Logica di Throttling

Se la richiesta supera i filtri, il thread entra nella logica di rate limiting. Viene avviato un timer ad alta risoluzione per misurare la latenza introdotta.

Il core del throttling è implementato come un ciclo `while(1)` basato su operazioni atomiche, senza l'uso di lock pesanti:

1) **Acquisizione Slot**: Il thread tenta di acquisire uno slot incrementando il contatore delle invocazioni correnti.
2) **Verifica Limite**:
   a) **Successo**: Se il valore ritornato è inferiore o uguale al limite, il thread ha acquisito il diritto di esecuzione. Esce dal ciclo e procede.
   b) **Fallimento**: Se il limite è superato, il thread entra in fase di attesa.
3) **Fase di Sleep**: Prima di dormire, viene aggiornata la statistica dei thread bloccati. Successivamente, il thread invoca wait_event_interruptible sulla syscall_wqueue.
   La scelta della variante interruptible è essenziale: permette al thread di essere svegliato non solo dal timer del modulo, ma anche da segnali di sistema. Se un thread bloccato riceve un segnale, il wrapper rileva l'interruzione, abortisce l'attesa e ritorna immediatamente, prevenendo situazioni di processi "zombie" non terminabili.
4) **Risveglio e Retry**: Quando il thread viene svegliato dal timer, non procede automaticamente all'esecuzione, ma ricomincia il ciclo dal punto 1. Questo impone che il thread debba competere nuovamente per acquisire uno slot, garantendo il rispetto rigoroso del limite anche in presenza di un massiccio numero di thread risvegliati.

d. Calcolo delle Statistiche

Una volta acquisito il diritto di procedere (uscendo dal ciclo di throttling), il wrapper arresta il timer di misurazione.

Il delta temporale calcolato rappresenta esclusivamente il ritardo imposto dal modulo (il tempo trascorso in coda di attesa). Questo valore viene utilizzato immediatamente per aggiornare le statistiche del *Peak Delay*.

È fondamentale sottolineare che questa operazione avviene prima dell'esecuzione della syscall reale. Questo garantisce che la metrica rifletta accuratamente l'overhead del throttling, disaccoppiandolo completamente dal tempo di esecuzione intrinseco della system call (che dipende dall'IO, dal disco, etc.).

e. Esecuzione della Syscall

Terminata la fase di gestione amministrativa, il wrapper invoca la funzione originale tramite il puntatore risolto in precedenza. Il valore di ritorno della syscall reale viene catturato e restituito intatto al chiamante nello spazio utente, rendendo l'intera operazione di intercettazione trasparente al flusso logico dell'applicazione.

VI. Gestione e Configurazione

L'architettura del sistema prevede una netta separazione tra il core logico di monitoraggio, che opera nel contesto delle interruzioni o dei processi monitorati, e l'interfaccia di gestione rivolta all'amministratore di sistema. Per colmare il divario tra lo spazio utente e lo spazio kernel, è stato progettato e implementato un driver di dispositivo a caratteri.

Questa scelta architetturale, preferita rispetto ad alternative come i filesystem virtuali o socket netlink, nasce dalla necessità di disporre di un canale di comunicazione robusto, sincrono e versatile, capace di gestire con pari efficienza sia il trasferimento di complesse strutture dati di configurazione sia la generazione di reportistica testuale per il monitoraggio umano.

a. Inizializzazione e Integrazione nel Filesystem

Al momento del caricamento del modulo, il kernel alloca dinamicamente un Major Number disponibile e registra il driver del dispositivo a caratteri.

Un aspetto qualificante del design è l'automatismo nella creazione del nodo di accesso. Invece di delegare all'utente l'onere di creare manualmente il file speciale tramite comandi di sistema, il modulo interagisce direttamente con il modello a oggetti del kernel Linux creando una classe dedicata e istanziando il dispositivo logico. Questa operazione innesca la creazione automatica del nodo nel filesystem virtuale, rendendo

l'interfaccia di controllo immediatamente accessibile alle applicazioni utente senza passaggi intermedi di configurazione.

b. Interfaccia di Controllo IOCTL

La complessità dei parametri di configurazione richiesti dal monitor ha guidato la decisione di non utilizzare le classiche operazioni di scrittura su file per impartire comandi, ma di adottare l'interfaccia ioctl. L'utilizzo di semplici scritture avrebbe costretto il kernel a farsi carico di gravosi compiti di parsing testuale, introducendo fragilità e potenziali vulnerabilità di sicurezza. Al contrario, l'approccio tramite ioctl permette di definire un protocollo di comunicazione rigoroso basato su strutture dati fortemente tipizzate.

Attraverso questo canale, lo spazio utente può inviare comandi atomici per manipolare le liste di controllo degli accessi. Il driver funge da validatore e smistatore: riceve i dati grezzi, ne verifica la coerenza strutturale e semantica, e infine invoca le primitive interne per aggiungere o rimuovere le regole di filtraggio per le system call, gli identificativi utente e i programmi.

Particolare attenzione progettuale è stata dedicata alla gestione dei percorsi degli eseguibili. Consapevoli che il riferimento a un file tramite percorso testuale può divenire invalido se il file viene rinominato o cancellato, il driver espone una doppia modalità operativa. L'amministratore può configurare il monitoraggio fornendo il percorso canonico, che il driver risolverà internamente, oppure, per scenari che richiedono maggiore resilienza, può fornire direttamente la coppia univoca composta da numero di inode e identificativo del device. Questa flessibilità assicura che le regole di sicurezza persistano correttamente anche a fronte di mutazioni nel filesystem.

c. Reportistica e Statistiche

Per quanto concerne la visualizzazione dello stato del sistema, il driver implementa l'operazione di lettura standard secondo un paradigma basato su istantanee. Quando un applicativo utente tenta di leggere dal dispositivo, il driver non si limita a esporre direttamente la memoria kernel, operazione che risulterebbe insicura e incoerente in un ambiente fortemente concorrente.

Il sistema invece alloca dinamicamente un buffer temporaneo in memoria kernel e procede a "congelare" lo stato attuale del monitor. Vengono interrogate atomicamente le strutture dati statistiche, recuperati i contatori di throttling, i valori di picco di latenza e le liste di configurazione attive. Questi dati vengono successivamente formattati e serializzati in un report testuale strutturato all'interno del buffer temporaneo. Solo al termine di questa fase di elaborazione il contenuto viene copiato nello spazio di indirizzamento dell'utente.

Questa strategia di disaccoppiamento garantisce che l'utente riceva sempre una fotografia coerente e non corrotta dello stato del sistema in un preciso istante temporale, evitando le race condition che potrebbero emergere leggendo dati vivi mentre vengono aggiornati dal traffico delle system call.

d. Sicurezza e Controllo degli Accessi

La natura critica delle operazioni gestite dal modulo impone un rigoroso modello di sicurezza. Sebbene il nodo del dispositivo viene configurato con permessi che ne consentono la lettura a utenti non privilegiati per scopi di monitoraggio, le operazioni che alterano lo stato del sistema sono protette da controlli aggiuntivi.

All'interno del driver, ogni comando di configurazione inviato tramite ioctl è subordinato a una verifica preventiva delle credenziali del processo chiamante. Il codice verifica esplicitamente che l'utente possieda i privilegi di amministrazione, respingendo con un errore di permesso qualsiasi tentativo di modifica proveniente da utenti non autorizzati.

Questo meccanismo di difesa in profondità assicura che la politica di throttling del sistema non possa essere compromessa o disattivata da attori malevoli o processi non autorizzati.

VII. Ciclo di Vita del Modulo

La stabilità di un modulo kernel non dipende solo dalla correttezza delle sue funzionalità a regime, ma soprattutto dalla robustezza delle procedure di inizializzazione e, ancor più criticamente, di rimozione. Un errore durante queste fasi transitorie può portare a memory leak, risorse zombie o kernel panic fatali.

Per questo motivo, la gestione del ciclo di vita del modulo *sct* è stata progettata seguendo rigorosamente un ordine di dipendenze gerarchico e implementando meccanismi di sincronizzazione avanzati per lo shutdown sicuro.

a. Inizializzazione Gerarchica

All'atto del caricamento, la funzione sct_init orchestra l'avvio dei vari sottosistemi. L'ordine delle chiamate non è arbitrario, ma riflette le dipendenze funzionali tra i componenti:

1) **Statistiche e Filtri**: Vengono per prime allocate e inizializzate le strutture dati passive (tabelle hash, contatori, liste RCU). Questo assicura che, quando i componenti attivi partiranno, troveranno un ambiente dati pronto e coerente.
2) **Monitor Core**: Viene preparata l'infrastruttura di sincronizzazione del monitor (waitqueue, variabili atomiche globali).
3) **Timer**: Il timer viene armato e avviato. Da questo momento il sistema ha una nozione del tempo e delle finestre, anche se ancora non vi è traffico.
4) **Hooks**: Solo ora vengono installati gli hook sulle syscall. Il traffico può iniziare a fluire attraverso il wrapper. Essendo le strutture dati già pronte, il sistema è immediatamente operativo, ma non ancora configurato.
5) **Device Driver**: Come ultimo passo, viene registrato il dispositivo /dev/sct-monitor. Questo impedisce allo spazio utente di inviare comandi di configurazione a un modulo non ancora completamente inizializzato.

In caso di fallimento di uno qualsiasi di questi step, una logica di error handling basata su goto provvede a deallocare le risorse in ordine inverso, garantendo un'uscita pulita senza leak.

b. Riconfigurazioni Dinamiche

La modifica a runtime dei parametri operativi, sia che si tratti dell'attivazione/disattivazione globale o della ridefinizione del limite massimo di throughput, condivide una filosofia architetturale volta alla massimizzazione delle prestazioni.

Ogni volta che il monitoraggio viene disattivato, il modulo non si limita ad aggiornare un flag logico di stato, ma procede alla disinstallazione fisica degli hook dalle system call intercettate. Questa scelta radicale garantisce che, quando il monitor è spento, il sistema operi a velocità nativa con overhead zero, eliminando completamente il costo computazionale del wrapper e dei salti indiretti per il traffico non gestito.

Questa logica si applica in modo trasparente anche al cambio del limite massimo, gestito come una sequenza automatica di *Stop-Reset-Start*.

La disattivazione temporanea implicita nella fase di Stop (così come lo spegnimento manuale) comporta la rimozione degli hook e causa l'immediato risveglio di tutti i thread attualmente bloccati nella coda di attesa. Questi vengono lasciati defluire verso l'esecuzione senza restrizioni, svuotando il sistema da stati pendenti. Durante questa transizione, o finché il monitor resta spento, i nuovi thread in ingresso bypassano completamente il modulo.

Specificamente nel caso del cambio limite, contestualmente alla fase di *Reset*, tutte le statistiche accumulate vengono azzerate per evitare di inquinare i dati della nuova configurazione con metriche riferite alla vecchia soglia. Solo al termine delle operazioni, il monitor viene riattivato, reinstallando gli hook e imponendo le nuove regole al traffico.

c. Strategia di Shutdown Sicuro

La fase di rimozione del modulo rappresenta lo scenario più critico per la stabilità del kernel. Il rischio principale è che il codice del modulo venga rimosso dalla memoria mentre uno o più thread si trovano ancora all'interno del wrapper o, peggio, mentre la CPU sta per saltare all'indirizzo di una funzione appena deallocata.

Per mitigare questo rischio, è stato implementato un protocollo di shutdown in tre fasi, denominato *Stop-Drain-Teardown*:

1) **Stop**: Viene impedito l'ingresso al wrapper. Il dispositivo viene rimosso e gli hook vengono disinstallati.
2) **Drain**: Si attende lo svuotamento del sistema. La funzione cleanup_monitor imposta il flag globale unloading e utilizza una waitqueue dedicata per attendere che il contatore active_threads scenda a zero. Questo assicura che tutti i thread che erano già entrati nel wrapper completino la loro esecuzione e ne escano.

3) **Teardown**: Solo quando il sistema è garantito essere "fermo", si procede alla cancellazione del timer e alla liberazione della memoria delle strutture dati.

Tuttavia, il livello di sicurezza garantito durante l'unload dipende intrinsecamente dalla metodologia di hooking selezionata in fase di compilazione:

- **Ftrace Hooking**: Se si utilizza il sottosistema Ftrace, la sicurezza dell'unload è garantita architetturalmente dal kernel stesso.
  Ftrace utilizza meccanismi interni (basati su trampolini e RCU) che impediscono la rimozione fisica del codice fintanto che una callback è in esecuzione su una qualsiasi CPU. Questo, combinato con il reference counting del modulo, offre la massima garanzia che il sistema non cercherà mai di eseguire codice in pagine di memoria non più esistenti.
- **Discover Hooking**: Nel caso della manipolazione diretta della System Call Table, non esiste una garanzia assoluta di sicurezza lato kernel.
  Nonostante il meccanismo di *Drain* attenda che i thread escano dal wrapper, esiste una finestra temporale infinitesimale di rischio.
  Il problema risiede nella natura stessa del *reference counting*: nell'implementazione attuale, è il wrapper stesso a doversi autoproteggere incrementando il contatore all'ingresso e decrementandolo all'uscita.
  Tuttavia, affinché la protezione sia matematicamente assoluta (priva di race condition), la gestione del *reference counter* dovrebbe essere delegata al chiamante (il kernel core), il quale dovrebbe acquisire un lock o un riferimento sull'area di memoria del modulo prima ancora di effettuare l'istruzione di salto verso di essa.
  Poiché la tecnica del *Syscall Table Hacking* opera come un'intrusione esterna su un kernel ignaro della presenza del modulo, tale coordinazione preventiva è impossibile. Implementarla richiederebbe un patching profondo e strutturale del codice sorgente del kernel stesso (integrando il modulo nel core in modo che non possa mai essere rimosso o che il kernel ne gestisca il ciclo di vita), negando il concetto stesso di modulo caricabile a run-time. Di conseguenza, se il modulo viene scaricato nell'istante esatto in cui la CPU si trova nelle istruzioni di prologo o epilogo del wrapper (fuori dalla zona protetta dal contatore interno), il sistema tenterà di eseguire codice in una pagina di memoria deallocata, causando un Kernel Panic.
  Questo rischio, assente in Ftrace grazie alla sua integrazione nativa con il sottosistema RCU del kernel, rappresenta il principale compromesso di stabilità di questa modalità.

d. Fast Unload

Uno scenario critico durante la fase di *Drain* è la presenza di un elevato numero di thread bloccati nella `waitqueue` di throttling. Nel comportamento standard di shutdown, il modulo sveglia questi thread e concede loro di procedere all'esecuzione della syscall originale prima di uscire. Sebbene sicuro, questo approccio può rallentare significativamente il comando `rmmod`, costringendo l'amministratore ad attendere il completamento effettivo di tutte le chiamate pendenti dei processi sbloccati.

Per ovviare a questo problema di latenza, è stata implementata la funzionalità opzionale *Fast Unload*.

Quando attivata, questa modalità modifica radicalmente la logica di uscita dal wrapper in fase di shutdown. Invece di sbloccare i thread e permettere loro di eseguire la chiamata di sistema, il wrapper abortisce l'operazione. Rilevando il segnale di unload, il codice interrompe immediatamente l'attesa e ritorna allo spazio utente il codice di errore standard `-EINTR`, saltando completamente l'invocazione della funzione kernel originale.

Questa strategia *Fail-Fast* garantisce uno svuotamento istantaneo della coda e una rimozione del modulo più rapida, ma introduce un importante compromesso semantico.

Poiché la syscall richiesta dall'applicazione non viene effettivamente eseguita, questo comportamento altera il contratto standard tra kernel e user-space. Sebbene `-EINTR` sia un errore previsto dallo standard *POSIX* per le chiamate bloccanti, forzarlo su system call che l'applicazione potrebbe non aspettarsi di veder interrotte (o

su operazioni che non prevedono meccanismi di retry) potrebbe causare malfunzionamenti o terminazioni impreviste nei processi monitorati.

Per garantire la massima stabilità e il rispetto della semantica attesa, questa funzionalità è disabilitata di default e deve essere attivata esplicitamente dall'amministratore solo se la priorità è la velocità di manutenzione rispetto alla trasparenza operativa.

VIII. Link Esterni e Codice Sorgente

Il codice sorgente completo del progetto è ospitato sulla piattaforma GitHub. Il repository contiene:

- Il codice sorgente del Linux Kernel Module.
- Il codice dello strumento di configurazione per l'utente.
- Gli script di test per la validazione delle funzionalità.
- Il `Makefile` per la compilazione automatizzata e l'installazione.

Repository: https://github.com/F-masci/syscall-throttling