Alma Mater Studiorum-Università di Bologna
Campus di Cesena
Scuola di Scienze

Corso di laurea in Ingegneria e Scienze Informatiche

Analisi delle prestazioni del *Linux Kernel Runtime Guardian*

Relazione finale in

Programmazione di Reti

Presentata da:
Simone Magnani

Relatore:
Gabriele D'Angelo

Sessione II
Anno Accademico 2017/2018

«L'hardware è facile da proteggere: lo chiudi in una stanza, lo leghi ad una scrivania, o ne compri uno di ricambio. L'informazione pone molto più di un problema. Può esistere in più di un posto; può essere trasportata attraverso il pianeta in pochi secondi; e può essere rubata senza che tu ne sia a conoscenza.»

Bruce Schneier, Protect Your Macintosh, 1994

Abstract

Gli argomenti d'interesse di questo elaborato sono l'integrità e la sicurezza di un sistema operativo (principalmente Linux). In particolare, viene analizzato il recente software *Linux Kernel Runtime Guardian*, concentrando l'attenzione sulle sue funzionalità ed i controlli effettuati al fine di mantenere il proprio ambiente sicuro.

Prima dell'analisi del sistema, vengono introdotti gli argomenti inerenti per comprendere lo scenario in discussione, quali la sua struttura e gli strumenti in possesso dall'utente per l'interazione. Successivamente vengono presentati sia il software preso in analisi, servendosi della documentazione fornita dall'autore, sia quello progettato ed utilizzato per la misurazione delle performance *SysBench*, interamente sviluppato per calcolare la differenza in termini di tempi di esecuzione di determinate funzionalità in seguito al caricamento di LKRG. Infine, vengono proposti e commentati i risultati sperimentali ottenuti effettuando il test in vari sistemi differenti, al fine di valutarne positivamente o meno l'uso.

Il progetto è OpenSource e si può ottenere clonando il repository nel sito `https://bitbucket.org/SimoMagno/sysbench/src/master/`.

Prefazione

La tecnologia è un mondo molto vasto e in continua evoluzione, diventato non solo determinante per lo sviluppo delle società, ma soprattutto invasivo nella quotidianità delle persone. La maggior parte delle azioni svolte nell'arco di una giornata includono l'uso di mezzi tra cui i cellulari, i computer o altri dispositivi con i quali si riesce a reperire o caricare informazioni in rete in breve tempo. Molte persone usano in maniera superficiale tali tecnologie e non sono consapevoli dei numerosi pericoli presenti in rete; non si è abituati ad immaginare che un'informazione digitale che viaggia nel misterioso cyber spazio abbia un ruolo determinante nella nostra vita a pari valore di qualsiasi 'contratto' cartaceo o verbale. Talvolta, si sente una frase del tipo: "Ma cosa devo proteggere? E da chi? Io non ho nulla da nascondere".

In un mondo in cui ogni cosa oramai è condivisa in rete da più dispositivi è necessario informarsi sulla sicurezza, ed incentivare lo sviluppo di sistemi di prevenzione affidabili in grado di anticipare una minaccia o, in certi casi, addirittura riparare il danno causato. Bisogna però tenere in considerazione che tali sistemi possano essere non solo costosi economicamente, ma anche in termini di risorse che occupano all'interno del dispositivo. È dunque essenziale durante lo sviluppo di un software tenere in considerazione il grado di soddisfazione del futuro cliente, il quale è influenzato da numerosi fattori come la facilità d'uso e velocità di risposta.

È in questo scenario che nasce *SysBench*, un programma per effettuare un benchmark del sistema in seguito all'utilizzo del modulo di sicurezza *Linux Kernel Runtime Guardian*. Essendo una new entry nel mercato, LKRG non ha ancora ricevuto molte recensioni o analisi delle prestazioni che permettano ad un utente di valutarne seriamente l'uso. Pertanto, viene proposto SysBench come software OpenSource per fornire dati concreti circa le prestazioni di questo modulo.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

La sicurezza informatica è una tra le varie tematiche più discusse oggigiorno, non solo perchè ha acquisito maggiore importanza l'informazione, ma anche per l'aumento dell'interconnessione dei vari dispositivi. È infatti possibile ottenere o inviare una risorsa da/ad una persona situato dall'altra parte del mondo, condividere in tempo reale i nostri pensieri, esperienze e tant'altro. La cosa più interessante da osservare però è come i *device* o i servizi che utilizziamo quotidianamente proteggano le nostre informazioni, impedendo o meno ad altre persone esterne di accedervi. Nessuno sarebbe contento di sapere che un'altra persona, tramite vari attacchi possibili in rete, è entrata in possesso di dati privati che concernono la propria vita, come la password del proprio conto in banca o la lista dei nostri impegni. Fortunatamente, per la maggior parte degli sviluppatori è diventato di vitale importanza valorizzare tutto il lato di sicurezza di un sistema prima di fornire un servizio, al fine di garantire all'utente riservatezza ed integrità dei propri dati.

È in questo scenario che nasce *Linux Kernel Runtime Guardian*, un software innovativo con l'obiettivo di mantenere integre e sicure determinate aree del proprio sistema. Prima di parlare della sua struttura e funzionamento, è necessario introdurre gli argomenti teorici inerenti, ovvero le principali componenti di un sistema operativo (nel nostro caso Linux) e le funzionalità che un utente può utilizzare per l'interazione con esso.

1.1 Introduzione al kernel

Molto spesso in informatica i termini *kernel* e *sistema operativo* vengono utilizzati come sinonimi, perchè effettivamente il kernel è il cuore di un sistema operativo, ed ha come obiettivi principali l'interazione con le componenti hardware (hard disk, lettore DVD, etc.) e la fornitura di un ambiente per eseguire le applicazioni utente installate nel sistema. È inoltre uno tra i primi programmi caricati durante la fase di accensione, senza il quale non sarebbe possibile l'utilizzo delle componenti fisiche.

Il primo kernel Linux è stato sviluppato da Linus Torvald nel 1991, pensato per un'architettura specifica. Negli anni la community di sviluppatori si è ampliata, rendendolo portabile e potente, capace di competere con i più grandi marchi del mercato (Windows, MacOs, etc.). Le sue principali caratteristiche sono:

- kernel monolitico: è un unico, grande e complesso programma composto da differenti componenti logiche;
- caricamento dinamico dei moduli: nonostante sia un programma compilato e già in esecuzione nel sistema, offre l'opportunità di caricare/eliminare dinamicamente i moduli (noti come *device driver*), le componenti che lo formano;
- kernel threading: sono thread (un modo per dividere il programma in più componenti simultanee) del kernel associati ad un programma utente o a qualche funzionalità che Linux utilizza in maniera limitata ed efficiente per eseguire alcuni controlli periodici;
- supporto per applicazioni multi-thread: supporta come la maggior parte dei sistemi operativi le applicazioni multi-thread;
- preemptive: può arbitrariamente interrompere l'esecuzione di un task per poi riprenderla in seguito;
- supporto per sistemi multi-processore: supporta *SMP (symmetric multi-processing)*, grazie al quale il sistema utilizza più processori, ognuno dei quali gestisce i propri task;
- Virtual File System: implementando questa tecnologia, è più semplice rispetto agli altri sistemi operativi importare un *file system* (componente logica per il controllo di come e dove le informazioni sono salvate) esterno.

L'attributo sul quale vale la pena fare una riflessione ai fini di questo elaborato è il caricamento dinamico dei moduli. Nonostante sia una caratteristica tipica dei sistemi basati su *microkernel*, ovvero sistemi nei quali il kernel è suddiviso in vari servizi nello spazio utente (piuttosto che essere un unico software in esecuzione nello spazio kernel), Linux offre la possibilità di estendere o ridurre le funzionalità del kernel attraverso semplici strumenti.

È sufficiente utilizzare il comando *modprobe* o *insmode* con i privilegi di root (richiesti appunto per operazioni di rilevante importanza) per poter caricare un modulo compilato, al contrario *rmmod* per rimuoverlo. Mentre insmode carica nel kernel il modulo d'interesse senza eseguire controlli di dipendenze tra moduli, modprobe carica anche tutti i moduli richiesti necessari per il corretto funzionamento. Di seguito, ecco un esempio d'uso di alcuni di questi comandi:

```
prova@prova:~$ sudo insmod testModule.ko
[sudo] password for prova:
prova@prova:~$ lsmod | grep testModule
Module
testModule      16384   0
prova@prova:~$ sudo rmmod testModule.ko
prova@prova:~$ lsmod | grep testModule
prova@prova:~$
```

È possibile osservare come una volta caricato con successo, il modulo appaia nella lista di quelli attualmente in esecuzione nel kernel (comando *lsmod*). Nelle

prossime sezioni verrà spiegato come un utente si può interfacciare con un modulo, leggendo l'output che esso produce o modificando determinati parametri.

1.1.1 Architettura di un modulo

Il linguaggio di programmazione utilizzato per creare un modulo è il C, con qualche differenza rispetto ai classici programmi che si è abituati a scrivere. Infatti, mentre tutti i programmi utente scritti in C richiedono che ci sia una funzione *main*, nel sorgente del modulo bisogna specificare due funzioni: una di inizializzazione e una di uscita. A scelta dello sviluppatore ma altamente consigliato per creare una buona documentazione, possono essere indicate informazioni quali la licenza, l'autore, la descrizione e una versione del modulo. Tali attributi possono essere consultati una volta compilato il modulo con il comando *modinfo module_name*. Una volta compilato tramite un apposito *makefile*, ovvero un file contenente una serie di direttive per la compilazione, si ottiene la versione *.ko* del sorgente in formato ELF (Executable and Linkable Format), la quale può essere caricata con i metodi illustrati.

Osserviamo un esempio di un modulo per avere un'idea più precisa:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void)
{
        printk(KERN_INFO "Hello␣world!\n");
        return 0;
}

static void __exit hello_exit(void)
{
        printk(KERN_INFO "Goodbye␣World!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Simone␣Magnani");
MODULE_DESCRIPTION("Hello␣world␣module!");
MODULE_VERSION("0.1");
```

Tralasciando le informazioni autoesplicative finali, in tutti i moduli sono richiamate le macro *module_init* e *module_exit* alle quali vengono passate come parametri le corrispettive funzioni di inizializzazione e terminazione. Fino alla versione 2.4 del kernel, era sufficiente dichiarare tale funzioni con i nomi *init_module* e *cleanup_module*, ma per comodità di progettazione si è scelto di introdurre le macro. Il modulo una volta caricato produce come output "Hello World", mentre "Goodbye World" è il messaggio stampato durante la fase di rimozione.

È possibile trovarsi in una situazione in cui un modulo 'A' debba utilizzare delle funzioni che, per scelta progettuale o stilistica, sono implementate nel modulo 'B'. Tutto quello che serve fare è creare un *header file* nel quale è dichiarato il prototipo della funzione, includerlo nel modulo 'A' tramite la direttiva *include* e aggiungere alla fine dell'implementazione della funzione nel modulo 'B' la riga *EXPORT_SYMBOL(function_name)*. Infine, è strettamente necessario che il

modulo 'B' che è una sorta di 'fornitore' della specifica sia caricato prima di 'A', altrimenti la risoluzione dei simboli fallirebbe. Per quello che ci concerne, queste sono le poche informazioni indispensabili che servono per comprendere come funziona il kernel, nel caso volessimo consultare il sorgente del software *LKRG* o scrivere un proprio modulo di utility.

1.1.2 Comunicazione utente-kernel

In questa sezione vengono trattati i vari metodi di comunicazione tra lo spazio utente e kernel, ponendo maggiore importanza nell'utilizzo del *Virtual File System (VFS)* e delle *sysctl*, i due metodi più comunemente utilizzati. Quest'ultimo in particolare è il principio di base su cui si basa l'interazione tra l'utente e il modulo LKRG, con il quale è possibile modificare determinati parametri del kernel.

Vi sono in linux diversi modi per interagire con il kernel, tra cui:

1. Virtual File System (Procfs, Sysfs, Configfs, Debugfs, Character Devices)
2. Socket based mechanism (Udp sockets, Netlink sockets)
3. Ioctl (input-output control)
4. System call
5. Kernel signaling
6. Upcall
7. Mmap

Linux implementa il concetto di virtual file system, grazie al quale l'utente può accedere a diversi tipi di file system in maniera uniforme; inoltre a seconda dell'utilizzo è possibile montarli (renderli accessibili) liberamente senza dover rispettare alcun vincolo. Diversi possiedono una propria interfaccia grazie alla quale è più semplice interagire, tra cui ricordiamo:

- /proc
- /sys
- /dev
- /sys/kernel/config (a volte anche solo /config)
- /sys/kernel/debug

Inizialmente progettato per esportare tutte le informazioni riguardanti i processi, come lo stato corrente e i file descriptor aperti, */proc* è ampliamente utilizzato per fornire dati circa:

- il sistema in esecuzione (CPU, memoria, interrupt, etc.);

- i device 'ide', 'scsi' e 'ttyś';
- la rete, come la tabella *ARP* (Address resolution Protocol), la lista delle socket utilizzate o qualche statistica.

In aggiunta, al suo interno vi è un'importante cartella denominata */proc/-sys*, la quale permette di configurare numerosi parametri del sistema, ognuno rappresentato da un singolo file. Tutte le sotto cartelle e i file interni non sono implementati secondo l'interfaccia *procfs*, bensì rispondono ad un meccanismo del kernel appositamente costruito chiamato *sysctl*.

A differenza della filosofia dei file system, secondo la quale il kernel può modificare un file ma l'utente rimane inconscio di questa modifica fino alla prossima apertura, tramite l'utilizzo delle *socket* il kernel può inviare notifiche ad un'applicazione utente in ascolto in qualsiasi momento. Vi sono 3 diverse famiglie da poter utilizzare:

- AF_INET: progettate per la comunicazione in rete, è possibile utilizzare socket *UDP (User Datagram Protocol)*, nonostante vi possa essere più sovraccarico di messaggi.
- AF_PACKET: permette all'utente di definire gli headers dei pacchetti.
- AF_NETLINK: specialmente progettate per la comunicazione tra utente e kernel.

Tra le varie chiamate di sistema che analizzeremo nella prossima sezione, esiste la funzione *ioctl()* grazie alla quale è possibile manipolare i parametri di determinati device. È implementata come un'unica funzione, la quale effettua la multiplazione dei comandi al fine che vengano richiesti i device opportuni. Il *multiplexing* si basa sul *file descriptor*, un puntatore astratto a quella specifica risorsa o canale, e il numero del comando da eseguire su quel preciso device.

L'invio di segnali da parte del kernel è un approcio leggermente diverso dai classici, dato che solo il kernel può inviare segnali allo spazio utente e non vice versa. Inoltre il quantitativo di dati trasportati dai segnali è molto limitato. Esistono due tipi diversi di API nello *user space*: le 'normali', le quali non trasportano dati, e le 'realtime' che trasportano fino ad un massimo di 32 bit. Affinchè il segnale sia ricevuto, nello spazio utente è necessario registrare un'apposita funzione con il compito di essere notificata; tale funzione sarà richiamata ogni volta che il kernel invierà un segnale di quel tipo.

Grazie all'uso delle *Upcall*, un modulo del kernel può invocare l'esecuzione di un programma utente nell'apposito spazio, configurando gli appositi parametri e variabili d'ambiente. Possiamo definire tale procedimento 'ad hoc', in quanto un qualsiasi utente solitamente non programma un apposito modulo per ogni programma che vuole eseguire, ma può risultare molto utile come tecnica nel caso dovesse utilizzare un programma specifico.

Infine, l'unico modo per trasferire grandi quantitativi di dati tra l'utente e il kernel senza effettuarne una copia, è il *memory mapping* ('mmap', mappatura della memoria). Sostanzialmente, si definisce un'area della memoria accessibile sia al kernel che all'utente, nella quale ogni attore può scrivere o leggere i dati

contenuti. La differenza con le tecniche mostrate fino ad ora, risiede nella notificazione dei cambiamenti: con mmap, quando un attore scrive dei dati nella porzione condivisa, l'altro non è a conoscenza di tali cambiamenti fino a quando non ne effettua la lettura. Nel caso non vi fossero delle politiche di lettura/scrittura, se i due attori scrivessero nel buffer senza prima 'consumare' l'informazione comunicata dall'altro, vi è una perdita di dati.

1.1.3 Esempi di comunicazione

Vengono proposti i due esempi di comunicazione più classici ampiamente utilizzati nella programmazione del kernel. Il primo metodo sfrutta il famigerato file system /proc, creando una entry del nostro modulo non appena esso viene caricato, assegnando le funzioni di lettura e scrittura che dichiariamo; in tale modo è possibile scrivere all'interno del file e leggerne il contenuto secondo le nostre politiche, implementate nelle apposite funzioni *write_proc()* e *read_proc()*. Questo esempio è stato testato nella versione 4.15.0.33-generic del kernel Linux.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#define MSG_SIZE (512)

static char *msg;
static int len,tmp;

ssize_t write_proc(struct file *filp, const char *buf, size_t count, loff_t *
    offp)
{
    unsigned long actual_len = count < MSG_SIZE - 1 ? count : MSG_SIZE - 1;
    copy_from_user(msg, buf, actual_len);
    len = count;
    tmp = len;
    return len;
}

ssize_t read_proc(struct file *filp, char *buf, size_t count, loff_t *offp)
{
    if(count > tmp)
        count = tmp;
    tmp = tmp - count;
    copy_to_user(buf, msg, count);
    if(count == 0)
        tmp = len;
    return count;
}

struct file_operations proc_fops =
{
write:
    write_proc,
read:
    read_proc
};

static int __init proc_init(void)
{
    if ((msg = kmalloc(MSG_SIZE, GFP_KERNEL)) == NULL)
        return -ENOMEM;
    proc_create("myProc", 0, NULL, &proc_fops);
    return 0;
}
```

```
static void __exit proc_exit(void)
{
    remove_proc_entry("myProc", NULL);
    kfree(msg);
}

MODULE_LICENSE("GPL");
module_init(proc_init);
module_exit(proc_exit);
```

Si è dichiarato un buffer denominato *msg* il quale ha il compito di tenere in memoria il contenuto della nostra entry *myEntry*. Nella struttura *file_ operations*, sono state agganciate le nostre rispettive funzioni di write e read, le quali non fanno altro che copiare il messaggio utente in msg nel primo caso, viceversa nel secondo. È molto importante eliminare la entry e liberare la memoria allocata per il nostro buffer quando il modulo viene rimosso, onde evitare errori o sprechi inutili di memoria. Una volta caricato nel kernel, per interagire con il nuovo file /proc/myEntry è sufficiente utilizzare dei semplici comandi *cat* e *echo* come di seguito:

```
prova@prova:~$ sudo insmod proc_entry.ko
[sudo] password for prova:
prova@prova:~$ lsmod | grep proc_entry
Module
proc_entry       16384   0
prova@prova:~$ echo "Ciao␣Mondo" > /proc/myEntry
prova@prova:~$ cat /proc/myEntry
Ciao Mondo
prova@prova:~$
```

Il secondo esempio proposto sfrutta la funzionalità *sysctl* del kernel, con la quale è possibile modificare determinati parametri a runtime sfruttando la directory /sys all'interno del file system /proc.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sysctl.h>

int my_var = 0;

static struct ctl_table_header *my_header;

static struct ctl_table my_value[] =
{
    {
        .procname       = "my_value",
        .data           = &my_var,
        .maxlen         = sizeof(int),
        .mode           = 0600,
        .proc_handler   = proc_dointvec_minmax,
    },
    {}
};

static struct ctl_table my_directory[] =
{
    {
        .procname    = "example",
        .mode        = 0600,
        .child       = my_value,
    },
    { }
};
```

```
static int __init myInit(void)
{
    /* register the above sysctl */
    my_header = register_sysctl_table(my_directory);
    if (!my_header)
        return -EFAULT;
    return 0;
}

static void __exit myExit(void)
{
    unregister_sysctl_table(my_header);
}

MODULE_LICENSE("GPL");
module_init(myInit);
module_exit(myExit);
```

Il modulo crea una propria directory (all'interno di /proc/sys/) chiamata 'example' con all'interno l'attributo *my_value*, al cui cambiamento viene risettato il valore della variabile *my_var* (il nostro "parametro" del kernel). Configurata la directory, viene registrata all'interno della tabella delle sysctl, e viene ritornato l'header a questa struttura, il quale serve non solo per verificare se l'operazione è andata a buon fine, ma anche per rimuovere l'entry dalla tabella delle sysctl durante la rimozione del modulo. Di seguito viene mostrato un'esempio d'uso del comando *sysctl* con il nostro parametro registrato:

```
prova@prova:~$ sudo insmod sysctl.ko
[sudo] password for prova:
prova@prova:~$ lsmod | grep sysctl
Module
sysctl  16384   0
prova@prova:~$ sysctl -a | grep my\_value
example.my_value = 0
prova@prova:~$ sysctl example.my\_value=100
example.my_value = 100
prova@prova:~$ sysctl -a | grep my\_value
example.my_value = 100
```

Un ultimo dettaglio su questo strumento: l'entry aggiunta al file system /proc viene rimossa al successivo riavvio della macchina, a meno che non vengano aggiunte queste preferenze al file */etc/sysctl.conf*.

1.2 Architettura di un processore

L'idea di porre dei livelli gerarchici di protezione processore e della memoria (cosiddetti *ring*), è stata per la prima volta introdotta nel computer *GE 645* nel 1965. Questa innovazione è stata molto apprezzata e rivisitata dalla maggior parte dei concorrenti nel mondo informatico, ed è persino utilizzata tutt'ora, seppure con qualche modifica.

L'idea consiste nell'inserire dei controlli riguardanti l'accesso al processore, per proteggere il sistema in base alla tipologia di operazione che deve svolgere. Tali controlli possono essere implementati in 3 modi:

- hardware(HW): i livelli di privilegi sono implementati in microcodice HW per il controllo degli accessi;
- software(SW): il meccanisco di controllo è interamente a livello SW;
- hardware + software (ibridi): componenti di entrambe le nature collaborano nella gestione dei ring.

Generalmente i sistemi con i controlli ibridi sono ottenuti combinando componenti progettate appositamente per tale funzionamento. Supponiamo di avere un computer con il processore strutturato per la cooperazione con un sistema operativo che, diversamente, implementa i controlli solamente a livello software. Cosa succede? Sicuramente potremmo affermare di non avere una gestione dei livelli sia software che hardware, in quanto nel sistema non è integrata questa funzionalità.

Il numero di ring può variare, ma generalmente quelli più interessanti da sapere che sono presenti in tutte le implementazioni sono due: livello 0 (kernel mode) e livello utente (user mode). Mentre il livello 0 è utilizzato dal sistema operativo e rappresenta la massima affidabilità, il livello utente è il livello meno privilegiato, dal quale il sistema si aspetta che le operazioni possano essere malevole.

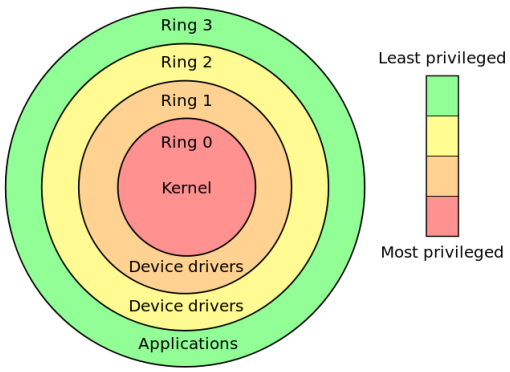

Figura 1.1: Ring nei processori della famiglia x86.
Fonte: https://en.wikipedia.org

Si può notare in Figura 1.1 che tra i due livelli appena discussi ve ne sono altri due utilizzati dai *device drivers*, ossia da quei programmi che controllano e gestiscono determinati dispositivi fisici o virtuali collegati al calcolatore. Un esempio molto comune è l'installazione della propria stampante a casa: quando viene richiesto di scaricare/aggiornare i driver, si tratta di queste risorse che si interfacciano con il dispositivo fisico.

1.2.1 User/Kernel spaces

La memoria di un calcolatore è suddivisa in due importanti aree: *user space* (spazio utente) e *kernel space* (spazio kernel).

Con il termine user space si fa riferimento a tutto il codice eseguito al di fuori del kernel del sistema operativo. Ogni processo eseguito in questo spazio ha una propria memoria virtuale e non può comunicare, a meno che non venga

esplicitato attraverso varie tecniche di programmazione, con gli altri processi. È importante sottolineare che un processo nello spazio utente non ha completo accesso alle risorse di sistema; l'utilizzo di tali risorse è garantito dalle varie *API (Application Programming Interface)* messe a disposizione dal sistema.

Il kernel space ha la proprietà di eseguire il codice a livello 0, tipicamente definita esecuzione in kernel mode. In tale modalità, i processi godono del completo accesso all'hardware e alle risorse, in quanto sono considerati affidabili.

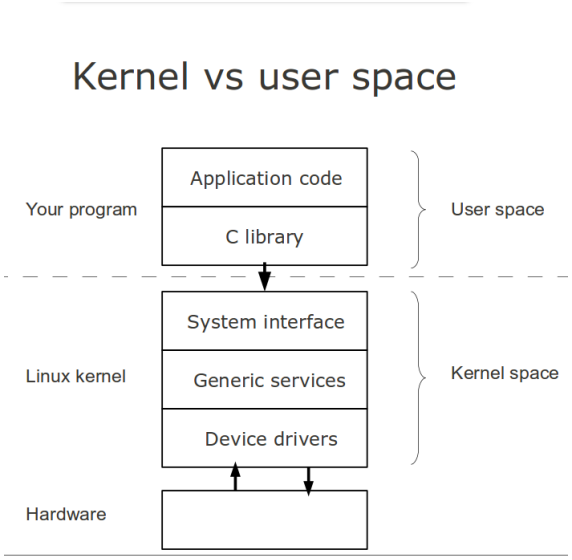

Figura 1.2: Kernel space e User space.
Fonte: https://3.bp.blogspot.com

La Figura 1.2 rappresenta i due diversi livelli di spazio e come si interfacciano con l'hardware. Supponiamo di dover lanciare un programma C che esegue una semplice operazione di scrittura su disco (ad esempio salvataggio di un file) per poi terminare. Come discusso poco fa, l'esecuzione del nostro programma avviene nello user space, ovvero lo spazio dove l'accesso diretto all'hardware non è permesso onde evitare danni alle risorse. L'eseguibile dovrà pertanto effettuare una richiesta (detta anche segnalazione) al sistema grazie alla quale avviene la scrittura su disco. Come è possibile che sia avvenuta l'operazione nonostante il mio programma è eseguito nello spazio utente in modalità utente? Quando il programma ha inviato la segnalazione al sistema, la *CPU (Control Processor Unit)* ha effettuato un'operazione di scambio molto importante tra il ring 0 ed il ring utente grazie alla quale in nostro programma riesce a terminare con successo: il *mode switch*.

1.2.2 CPU mode switch

Talvola il termine *mode switch* viene confuso con *context switch*, sebbene siano due cose concettualmente diverse. Mentre con il secondo si indica il passaggio effettuato dallo scheduler della CPU da un processo o thread ad un altro, il mode switch è il cambio di modalità di esecuzione delle istruzioni all'interno dello stesso processo. Quando il programma utente effettua una segnalazione di sistema per richiedere l'utilizzo di determinato hardware, all'interno della CPU

avviene un cambio di modalità, passando dall'esecuzione in modalità utente alla modalità kernel. Avendo ora i privilegi necessari, è possibile soddisfare la richiesta e, una volta terminata, verrà effettuato un ulteriore passaggio per ritornare alla modalità utente.

Il mode switch e il context switch a seconda di come sono implementati possono avere un costo di esecuzione in termine di tempo molto variabile. Per questo si cerca di progettare il sistema operativo in maniera da effettuare il minor numero di passaggi che, nonostante avvengano in tempi pari all'ordine del nanometro, sono impercettibili all'essere umano ma non trascurabili per il processore. Uno dei vantaggi da sempre riconosciuto a Linux è proprio quello di aver un costo estremamente basso per effettuare tali operazioni.

1.3 Le system call

Fino ad ora si è parlato di interfacce per segnalare al sistema la richiesta di un'istruzione privilegiata senza spiegare altro. Queste funzioni sono conosciute con il nome di *system call (chiamate di sistema)* e sono presenti in ogni sistema operativo, sebbene in maniera differente. Infatti tali funzioni non sono universali per problemi di incompatibilità sia hardware che software; ogni sistema operativo ne possiede delle proprie, le quali hanno comunque come unico obiettivo l'interazione del programma con il sistema e i device che ne fanno parte.

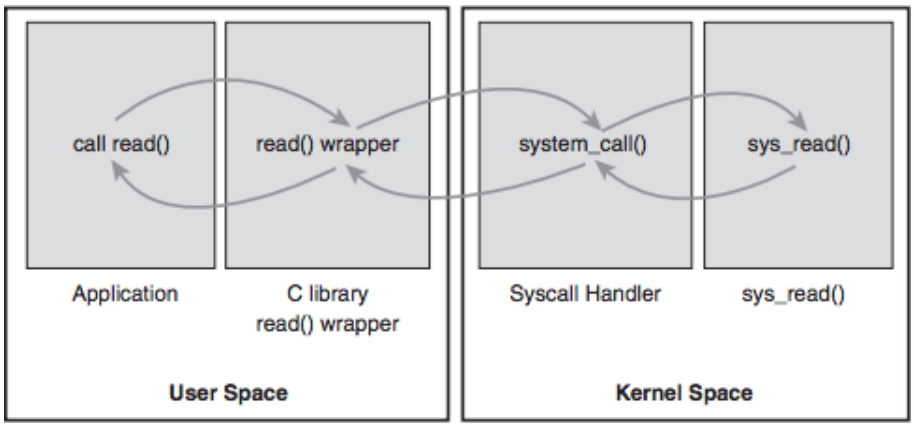

Figura 1.3: Esempio di system call read().
Fonte: https://www.quora.com

La Figura 1.3 mostra un esempio di chiamata della system call *read()*: tra la chiamata a funzione nella nostra applicazione e la vera chiamata di sistema, si noti come è presente un *call wrapper*. Un wrapper consiste in un *layer* di codice con il compito di tradurre in un'interfaccia compatibile il codice di libreria; è sostanzialmente un API grazie alla quale non vi è bisogno di reimplementare del codice, ma si utilizza l'interfaccia che tale prodotto mette a disposizione. La chiamata *read()* è in realtà un'invocazione del wrapper presente nella libreria standard del linguaggio C (*glibc*), il quale ha il compito a sua volta di richiamare la vera e propria system call. Una volta invocata ed avvenuto il *mode switch* in quanto deve essere eseguito del codice kernel, un *handler* (intercettatore) inoltra la chiamata ad un livello inferiore, oltre al quale avviene l'operazione

vera e propria sul device (fisico o virtuale) d'interesse. Alla fine dell'operazione, la CPU ritorna in modalità utente ed il controllo viene restituito al programma.

Una lista completa di tutte le system call con documentazione può essere trovata nel sito `http://man7.org/linux/man-pages/man2/`.

1.3.1 Esempi di strumenti utili

In questa sezione vengono mostrati brevemente degli strumenti da riga di comando molto utili per la fase di debugging del nostro programma, nel caso fossimo interessati a comprenderne il funzionamento a più basso livello. Supponiamo di avere il nostro programma C come segue:

```
#include <fcntl.h>

int main(void)
{
        int fd = open("foo.txt", O\_RDONLY);
        if(fd > 0) close(fd);
        return 0;
}
```

Nella funzione main viene dichiarato un intero all'interno del quale verrà salvato il file descriptor di *foo.txt* nel caso il file esista, altrimenti -1. Se l'apertura del file è avvenuta con successo, si prosegue con la semplice chiusura. Le funzioni *open()* e *close()* sono delle chiamate di sistema, le quali comunicano al kernel di eseguire tali operazioni privilegiate nell'hardware. Grazie al comando *strace* è possibile visualizzare tutte le system call effettuate dal nosto programma. Lanciando il comando *strace ./myprogram* dove *myprogram* è l'eseguibile ottenuto in seguito alla compilazione, si ottiene un output simile:

```
prova@prova:~$ strace ./myprogram

execve("./myprogram", ["./myprogram"], [/* 70 vars */]) = 0
brk(0)                                  = 0x22c3000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    x7f8feb036000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=79132, ...}) = 0
mmap(NULL, 79132, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8feb022000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P␣\2\0\0\0\0\0"..., 832)
     = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1840928, ...}) = 0
mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
    x7f8feaa51000
mprotect(0x7f8feac0b000, 2097152, PROT_NONE) = 0
mmap(0x7f8feae0b000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    MAP_DENYWRITE, 3, 0x1ba000) = 0x7f8feae0b000
mmap(0x7f8feae11000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
    MAP_ANONYMOUS, -1, 0) = 0x7f8feae11000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    x7f8feb021000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    x7f8feb01f000
arch_prctl(ARCH_SET_FS, 0x7f8feb01f740) = 0
mprotect(0x7f8feae0b000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x7f8feb038000, 4096, PROT_READ) = 0
munmap(0x7f8feb022000, 79132)           = 0
```

```
open("foo.txt", O_RDONLY) = 3
close(3) = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Le due righe evidenziate corrispondono esattamente alle due system call richiamate nel programma C, quindi possiamo affermare che quando viene effettuata l'apertura di un file, una scrittura, una lettura o operazioni simili avviene effettivamente la chiamata di una system call con tutto il procedimento di mode switch presentato nella precedente sezione. L'esempio proposto è piuttosto banale e l'uso del comando strace in una situazione del genere è pressochè inutile, ma si immagini in un software molto più complesso quanto possa essere utile analizzare il numero di system call effettuate, al fine di stimare e/o ridurre il tempo di esecuzione in kernel mode.

Un altro comando proposto è *objdump* (esistono comandi molto più potenti e interattivi, come *radare* o *gdb*, ma meno semplici da utilizzare), il quale permette di disassemblare l'eseguibile ed ottenere il codice in linguaggio Assembly corrispondente. Scrivendo in un terminale *objdump -d myprogram*, si comunica al tool di effettuare il disassemblaggio dell'eseguibile *myprogram*. Eliminando un po' di output non interessante e concentrandoci sulla funzione *main* si ottiene:

```
prova@prova:~$ objdump -d myprogram
...
000000000040057d <main>:
  40057d:       55                      push   %rbp
  40057e:       48 89 e5                mov    %rsp,%rbp
  400581:       48 83 ec 10             sub    $0x10,%rsp
  400585:       be 00 00 00 00          mov    $0x0,%esi
  40058a:       bf 44 06 40 00          mov    $0x400644,%edi
  40058f:       b8 00 00 00 00          mov    $0x0,%eax
  400594:       e8 e7 fe ff ff             callq 400480 <open@plt>
  400599:       89 45 fc                mov    %eax,-0x4(%rbp)
  40059c:       83 7d fc 00             cmpl   $0x0,-0x4(%rbp)
  4005a0:       7e 0f                   jle    4005b1 <main+0x34>
  4005a2:       8b 45 fc                mov    -0x4(%rbp),%eax
  4005a5:       89 c7                   mov    %eax,%edi
  4005a7:       b8 00 00 00 00          mov    $0x0,%eax
  4005ac:       e8 9f fe ff ff             callq 400450 <close@plt>
  4005b1:       b8 00 00 00 00          mov    $0x0,%eax
  4005b6:       c9                      leaveq
  4005b7:       c3                      retq
  4005b8:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  4005bf:       00
```

Come nell'output precedente, sono riportate in evidenza le due chiamate di sistema *open()* e *close()* con le relative operazione intermedie di caricamento parametri nei registri della CPU e pulizia di registri dove salvare l'output. Per chi volesse comprendere il significato di *@plt*, è possibile trovato una soddisfacente spiegazione nel seguente sito `https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html`. Le system call e il codice assembly variano a seconda dell'architettura e del sistema operativo, quindi è molto probabile che lanciando i comandi proposti non si ottengano gli stessi risultati.

Capitolo 2

Il progetto *Linux Runtime Guardian Kernel*

Figura 2.1: Logo del software Linux Runtime Guardian Kernel
Fonte: `https://openwall.info`

LKRG è un software gratuito Open Source sviluppato da Adam 'pi3' Zabrocki, un dipendente della Microsoft amante della sicurezza informatica. Si può ottenere in 3 forme differenti: la versione a pagamento, quella 'light' oppure quella sperimentale, la quale differisce per le funzionalità introdotte, nonostante la possibile presenza di bug.

Come sottolinea nella pagina di documentazione l'autore, LKRG è ancora lontano dall'essere una soluzione perfetta per la sicurezza del kernel (può essere bypassato), ma un corretto uso soprattutto se installato in un sistema stabile e non già compromesso potrebbe migliorarne l'integrità e la sicurezza. Infatti gli obiettivi di questo modulo sono:

- prevenire le modifiche non supportate al kernel, inserendo delle regole da rispettare, grazie alle quali qualsiasi patch prima di entrare in funzionamento deve essere validata;
- rilevare con successo un'exploit del kernel, bloccando il processo malevolo ed annullando le modifiche apportate.

Per garantire queste *feature*, LKRG sfrutta un proprio database interno contenente i valori *hash* calcolati in base alle regioni ed i parametri del kernel più importanti da controllare.

2.1 Regioni controllate ed eventi scatenanti

Le regioni del sistema controllate sono:

- i moduli caricati e tutti i loro attributi (puntatori alle strutture dati, etc.);
- l'unità di input-output (IOMMU);
- tutta la sezione del kernel di sola lettura (.rodata);
- le eccezioni del kernel;
- tutta la sezione .text del kernel (dove vi è la tabella delle system call, le funzioni, le procedure, etc.)
- dati critici della CPU e dei singoli core.

A differenza delle altre regioni, l'ultima che concerne la CPU è la parte più critica del progetto. Bisogna tener conto di vari scenari d'utilizzo, supponendo una possibile aggiunta o rimozione a runtime di più core, i quali devono necessariamente essere controllati singolarmente; infatti i controlli d'integrità vengono effettuati su ogni componente, ricalcolando gli hash ed aggiornanando il database, al fine di avere dei dati precisi del sistema. Queste problematiche si possono presentare sia in un'installazione fisica del sistema operativo, sia in una sua virtualizzazione: in quest'ultimo scenario risulta molto semplice cambiare i parametri hardware da virtualizzare, dunque affinchè LKRG sia versatile deve assolutamente prendere in considerazione tale regione del sistema nei suoi minimi particolari.

I controlli d'integrità delle varie aree sono scatenati da eventi ritenuti più o meno a rischio dal sistema, secondo delle percentuali assegnate. La routine di controllo è eseguita:

- dagli interrupt del timer interno al kernel, il quale genera e inserisce nella *working queue* i cosiddetti *work item*;
- su richiesta tramite un comando dedicato ricevuto tramite il canale di comunicazione (sysctl);
- ogni volta che è rilevata l'attività di un modulo;
- ogni volta che viene rilevato un nuovo core attivo, sia fisico che virtuale;
- in base a vari eventi di sistema riportati nella tabella sottostante.

Evento	Probabilità
CPU idle	0.005
CPU frequency	10
CPU power management	10
Network device	1
Network event	5
Network device IPv4 changes	50
Network device IPv6 changes	50
Task structure handing off	0.01
Task going out	0.01
Task calling *do_munmap()*	0.005
USB changes	50
Global AC events	50

Tabella 2.1: Eventi di sistema e probabilità d'attivazione

Nonostante la tabella possa subire variazioni in seguito ad aggiornamenti del software, si può osservare come per ora LKRG abbia fissato delle percentuali relativamente basse per determinate categorie di eventi, mentre per altri è stata data maggiore importanza assegnando persino un 50% di possibilità di scatenare un controllo.

2.2 Modello delle minacce

Per definire un modello di minacce che LKRG deve prendere in considerazione, si sono supposti i seguenti 3 scenari:

1. attacco al kernel dalla boot-chain (avvio del sistema);
2. attacco al kernel tramite le sue vulnerabilità note;
3. attacco al kernel persistente, come l'utilizzo di una backdoor.

Nel primo caso ci si trova dinanzi ad uno scenario fuori dall'obiettivo di LKRG, molto meno noto di altri attacchi famosi e più complicato da applicare. Infatti l'attaccante ha come obiettivo quello di compromettere il sistema durante le prime fasi di start-up, quando ancora il kernel (e quindi anche il modulo) non è stato inizializzato del tutto.

Il secondo caso è in parte coperto dal software e viene trattato nella prossima sezione; consiste in una scrittura (malevola) nella memoria del kernel grazie alla quale un attaccante è in grado di sovrascrivere i processi da eseguire (e quindi alterare il flow d'esecuzione) oppure acquisire privilegi per esecuzione di comandi che un normale utente non potrebbe lanciare.

Gli attacchi persistenti sono l'argomento più sensibile, in quanto è impossibile proteggere interamente il kernel da tutte le possibili modifiche, ma esse possono essere limitate se si definisce il concetto di 'modifica non supportata'. Linux non nasce come un sistema in grado di proteggersi dalle 'wild modifications', ed il kernel è un blocco di codice monolitico in grado di apportare

cambiamenti autonomamente. Questo dettaglio va in contrasto con il primo degli obiettivi di LKRG, pertanto al fine di garantire un buon livello di protezione il modulo deve:

- supportare solo le modifiche legittime del kernel (*_JUMP_LABEL);
- bloccare la funzionalità di caricamento ed rimozione dei moduli (senza i privilegi di root), creando un canale sicuro di comunicazione tra utente e kernel per questo tipo di funzionalità;
- imporre l'esecuzione dei controlli d'integrità in determinate situazioni.

Le *_JUMP_LABEL (in assembly language 'goto label') sono un meccanismo a basso livello di programmazione molto utile, grazie al quale è possibile saltare da una parte all'altra del codice. Un esempio d'uso sono i tracepoint o le stringhe di debug, le quali a seconda del relativo parametro vengono visualizzate o meno nel corso dell'esecuzione del kernel. Se avviene una modifica alla sezione .text del kernel dovuta ad una di queste label, LKRG esegue i seguenti controlli:

- se l'istruzione 'NOP' ('no operation' in assembly language) viene modificata in 'jmp', essa viene codificata per controllare che il puntatore al target dell'istruzione sia interno alla funzione stessa, ovvero che condivida lo stesso spazio dei simboli (nel caso fosse così, viene considerata legittima);
- nel caso fosse 'jmp' ad essere modificata, l'unico cambiamento permesso è un'instruzione di 'NOP'.

Se il kernel è compilato senza l'opzione che abilita le *_JUMP_LABEL, ovvero CONFIG_JUMP_LABEL=n, non vi è bisogno di effettuare tali controlli, in quanto essendo disabilitate non gli permettono di automodificarsi, rendendolo molto più statico, predittivo e più semplice da calcolarne gli hash per il database di LKRG.

2.2.1 Branch sperimentale

Creando un nuovo canale di comunicazione tra l'utente ed il kernel LKRG crea nuovi potenziali vettori di attacco in modalità utente, che si possono riassumere in attacchi:

1. ai processi utente;
2. ai file su disco;
3. ai processi tramite accesso diretto agli indirizzi di memoria;
4. ai file tramite accesso diretto al disco;
5. ai processi tramite le librerie condivise (definiti attacchi 'intermedi').

Al fine di evitare attacchi del primo tipo, LKRG definisce un set di 'protected process' salvato in memoria come 'red-black tree' (albero binario autobilanciato), garantendo che nessun processo in user mode riesca a cambiare lo stato dei processi in esecuzione. In aggiunta, dato che nessuno in modalità utente (incluso 'root') è in grado di vedere questa tipologia di processi, solo uno 'protected' è in grado di stabilire una connessione con terze parti.

Simile al caso precedente, gli attacci ai file su disco vengono prevenuti creando un subset di file definiti anch'essi 'protected'. Il loro contenuto è pertanto inalterabile per tutte le esecuzioni in user mode, mentre la lettura è permessa solamente se l'utente proprietario del file ha assegnato i privilegi necessari all'utente che richiede tale operazione.

Per quanto riguarda gli scenari di attacco tramite indirizzi definiti 'raw', LKRG forza tutti i processi esclusi quelli 'protected' a cedere la loro capacità CAP_SYS_RAWIO, ovvero il permesso grazie al quale possono eseguire operazioni di input/output direttamente con gli indirizzi di memoria e non tramite interfacce. Questa tipologia d'attacco è comunque possibile, in quanto non vi è alcun modo di bloccare gli accessi 'raw' al disco, ed anche senza il permesso CAP_SYS_RAWIO un attaccante con i giusti privilegi sarebbe in grado di aprire i file situati in */dev/sda[0-x]* ed inserire del proprio codice all'interno.

L'ultimo vettore d'attacco può essere contrastato in due maniere differenti: la prima consiste nel compilare staticamente tutto il codice compreso quelli di libreria, mentre un modo proposto da LKRG è quello di definire ogni file di libreria 'protected', con l'obiettivo di impedire l'accesso ad altri processi tramite possibili bug in tale codice.

Infine, un'ulteriore risorsa per il controllo d'integrità sono i file di log che LKRG genera ogni volta che rileva un'inconsistenza nel kernel. Per evitare che un attaccante ripulisca i file di log ed alteri lo stato del kernel, viene salvato un altro set di file definiti 'append', con i quali è possibile ricostruire lo stato del sistema. Mentre rimane possibile appendere informazioni in questi file, il modulo vieta la sovrascrittura e la modifica del loro contenuto, per mantenere informazioni consistenti.

Una breve nota da fare: per interagire con LKRG bisogna essere amministratori di sistema (esempio 'root') e ciò implica l'introduzione di un possibile vettore d'attacco, consistente nella tecnica nota di 'privileges excalation', grazie alla quale solitamente tramite una falla in un servizio si riesce ad ottenere i privilegi di root.

2.3 Rilevazione degli attacchi

Questa sezione ricopre il secondo obiettivo prefissato di LKRG, ovvero la rilevazione di un attacco e la successiva chiusura del processo. Per garantire integrità al sistema, vengono monitorati diversi parametri importanti assegnati ai singoli task (processi o thread), che non possono subire variazione durante l'esecuzione del programma. Tali parametri sono:

- puntatore alla struttura del task;

- pid (process identifier);
- nome del processo;
- puntatore alla struttura 'cred';
- puntatore alla struttura 'real_cred';
- UID (user identifier);
- GID (group identifier);
- EUID (effective user identifier);
- EGID(effective group identifier);
- SUID (set user identifier);
- SGID (set group identifier);
- FSUID (file system user identifier);
- FSGID (file system group identifier);
- SECCOMP (secure computing with filters);
- SELinux (security-enhanced Linux), variabili selinux_enable e selinux_enforcing.

Non è necessario che l'utente conosca tutti questi attributi; è sufficiente essere consapevoli che molti di essi possono essere modificati tramite varie system call, e quindi da un programma utente. Quando viene invocata una chiamata di sistema il cui scopo è quello di alterare questi parametri, LKRG intercetta l'esecuzione della funzione compiendo i propri controlli degli attributi non solo del processo chiamante, ma di tutti quelli presenti nel sistema (un attaccante esperto potrebbe utilizzare una falla in un servizio X per arrivare ad alterare il servizio Y). Le system call che scatenano tali controlli sono:

- setfsgid;
- setfsuid;
- setregid;
- setreuid;
- setgid;
- setuid;
- setresgid;
- setresuid;
- setgroups;

- exit;
- fork;
- execve;
- init_module (o finit_module, caricamento modulo nel kernel);
- delete_module (rimozione modulo dal kernel);
- may_open (eseguita quando l'utente vuole aprire una risorsa nel sistema);

Le funzioni *setX* sono invocate per modificare il corrispondente attributo *X* contenuto nel nome della funzione. *init_module* e *delete_module* servono per caricare o rimuovere moduli del kernel come mostrato nel Capitolo I, e richiedono i privilegi di root affinchè la loro chiamata termini con successo.

Con la *fork()* è possibile creare un processo definito 'child' (figlio) dal processo 'padre' in esecuzione, il quale avrà una copia esatta dello spazio degli indirizzi del padre (ed il programma è il medesimo), nonostante essi siano ovviamente in due locazioni diverse di memoria. I due processi continueranno la loro esecuzione dall'istruzione successiva alla chiamata di sistema.

Fortemente utilizzata è la system call *exit()*, con la quale avviene la terminazione istantanea del processo chiamante, la chiusura di tutti i file descriptor aperti appartenenti ad esso e la chiusura di tutti i suoi 'child process'.

Ultima ma non per importanza, *execve()* esegue il programma passato come primo parametro, sovrascrivendo tutto lo spazio dedicato al processo chiamante, reinizializzando le memorie ad esso associate (stack, heap); è necessario che il file da eseguire puntato dal primo argomento passato alla funzione sia un eseguibile o uno script in cui sia indicato l'interprete da utilizzare, altrimenti l'esecuzione fallisce.

Questo elenco di system call monitorate è aggiornato alla versione più recente del software (v.0.4) ma non è definitivo, in quanto potrà subire future modifiche in seguito ad aggiornamenti.

Capitolo 3

Architettura di *SysBench*

SysBench è lo strumento che ho sviluppato per effettuare l'analisi delle prestazioni di alcune funzioni del proprio sistema Linux, in particolare dopo il caricamento di LKRG. È interamente scritto in ANSI C e la sua architettura facilita future possibili aggiunte. Al fine di ottenere dei risultati attendibili, tale programma è semplificato al massimo, attenendosi semplicemente alla misurazione del tempo impiegato da ogni singola system call nella maniera più precisa.

3.1 Struttura del progetto

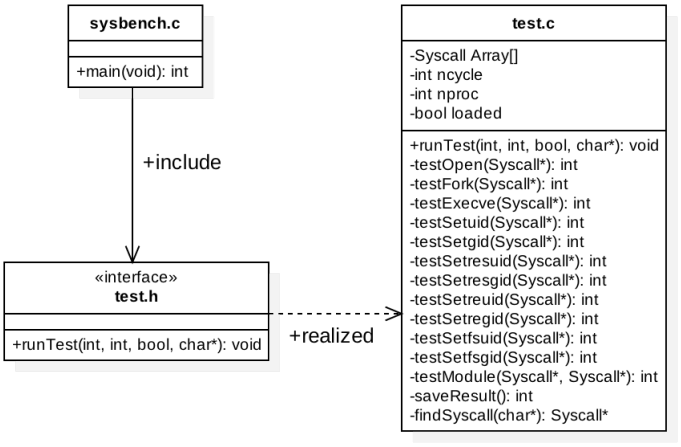

Figura 3.1: Architettura del software SysBench.

La Figura 3.1 mostra la struttura dei file del progetto e la loro interazione. Il *main* è contenuto nel file *sysbench.c*, in cui avviene il setup del programma che comprende la pulizia del prompt dei comandi, al fine di avere la finestra interamente dedicata a SysBench, e la verifica della presenza o meno di LKRG, la quale viene riportata assieme ai risultati nel file di output. Inoltre vengono letti i parametri passati da riga di comando, ovvero il numero di volte *ncycle* che la singola system call verrà testata ed il nome del file *filename* di output in cui vengono salvati i risultati. È importante prestare attenzione ad i parametri inseriti dall'utente, in quanto un malintenzionato potrebbe inserire codice

malevolo per apportare modifiche dannose, dato che vengono richiamate funzionalità potenzialmente pericolose come la *setuid()*. In merito a ciò, si è inserito il controllo di tale parametri al file di verificare la loro integrità.

Il file *test.c* è un grosso ed unico sorgente nel quale vengono definite internamente tutte le funzioni utilizzate nel test; infatti, *runTest()* che è l'unica funzione d'interfaccia è composta da un ciclo all'interno del quale vengono eseguite in serie tutte le valutazioni delle varie system call. Per scelta progettuale, si è deciso di creare la struttura *Syscall* che rappresenta le varie chiamate di sistema da eseguire. È composta da:

- il nome in formato human-readable della funzione;
- il puntatore alla funzione da eseguire, per fare si che ognuna di esse esegua il test opportuno (esempio: alla Syscall che testa la *open()* verrà assegnato il test *testOpen()* durante la creazione);
- il tempo di esecuzione del test misurato in doppia precisione.

Durante la fase di progettazione del sistema si è prestata maggiore attenzione alla scalabilità, ovvero la possibilità di aggiungere/rimuovere funzionalità in maniera semplice ed efficace in base alle proprie esigenze. Grazie alla struttura del programma, è possibile aggiungere un ulteriore test di un'altra system call in maniera facile ed immediata: è sufficiente dichiarare una nuova Syscall all'interno dell'array in fase di inizializzazione, implementare la propria funzione *testX()* ed assegnarla correttamente. Il sistema si preoccupa di soddisfare tutte le Syscall presenti, salvando il risultato di ognuna di esse independentemente dal numero.

Per risparmiare tempo, si è deciso di effettuare contemporaneamente sia il test di caricamento modulo sia quello di rimozione, in quanto non avrebbe senso scorporarli per il semplice motivo che il programma se deve effettuare l'inserimento di un modulo X volte, dovrà effettuare la sua rimozione altrettante volte al fine di non ricevere errore dalla funzione *insert_module()*, la quale fallisce se il target è già caricato nel kernel. Per questo è stata creata una funzione di utility *findSyscall()*, la quale serve per ottenere il puntatore alla Syscall opposta a quella che si sta eseguendo (esempio: si sta testando la *insert_module()*, si deve trovare il puntatore alla *delete_module()* e viceversa), in maniera tale da salvare direttamente il risultato della funzione complementare all'interno della sua struttura.

Le funzionalità *fork()* ed *execve()* lavorano in maniera leggermente differente dalle altre; mentre solitamente la system call viene eseguita ncycle-volte come inserito dall'utente, queste due funzioni devono tenere conto del limite di processi istanziati dall'utente. È infatti possibile osservare tale limite leggendo il file */proc/self/limits* in cui vi è indicato il valore massimo oppure un'apposita struttura in C contenente tutti i limiti del sistema. Una volta letto questo valore, bisogna tenere conto anche dei processi al momento attivi nel sistema e la possibilità di lanciarne altri durante l'esecuzione di SysBench; per questo se il numero inserito dall'utente è troppo grande, fork() ed execve() potranno creare al massimo $LIM/2$, dove LIM è il valore letto in precedenza.

La system call *execve()*, come riportato nella parte finale del capitolo precedente, sovrascrive tutto lo spazio dedicato al processo chiamante per eseguirne un altro e ciò potrebbe essere un problema dato che significherebbe la terminazione di SysBench non appena viene richiamata tale funzione. Come soluzione, si è scelto di far eseguire il comando ad un processo figlio, generato mediante la fork(), ottenendo così come tempo finale d'esecuzione la somma delle due chiamate a funzione. Quest'ultima feature è possibile grazie alla sincronizzazione tra i processo 'padre' ed i 'figli' utilizzando la funzione *wait()*, grazie la quale il chiamante attende che tutti i suoi figli notifichino la loro terminazione, e solo in seguito procede con l'esecuzione del suo codice. Con questa soluzione non solo si riesce a misurare abbastanza accuratamente il tempo d'esecuzione dell'intera chiamata a funzione, ma il programma non genera dei processi definiti 'zombie', ossia processi creati e non terminati che rimangono in memoria in attesa di essere rimossi consumando inutilmente risorse

3.2 Caso d'uso

Essendo SysBench un software d'utility sviluppato per un utilizzo semplice e mirato, l'unico caso d'uso da commentare è il seguente mostrato in Figura 3.2.

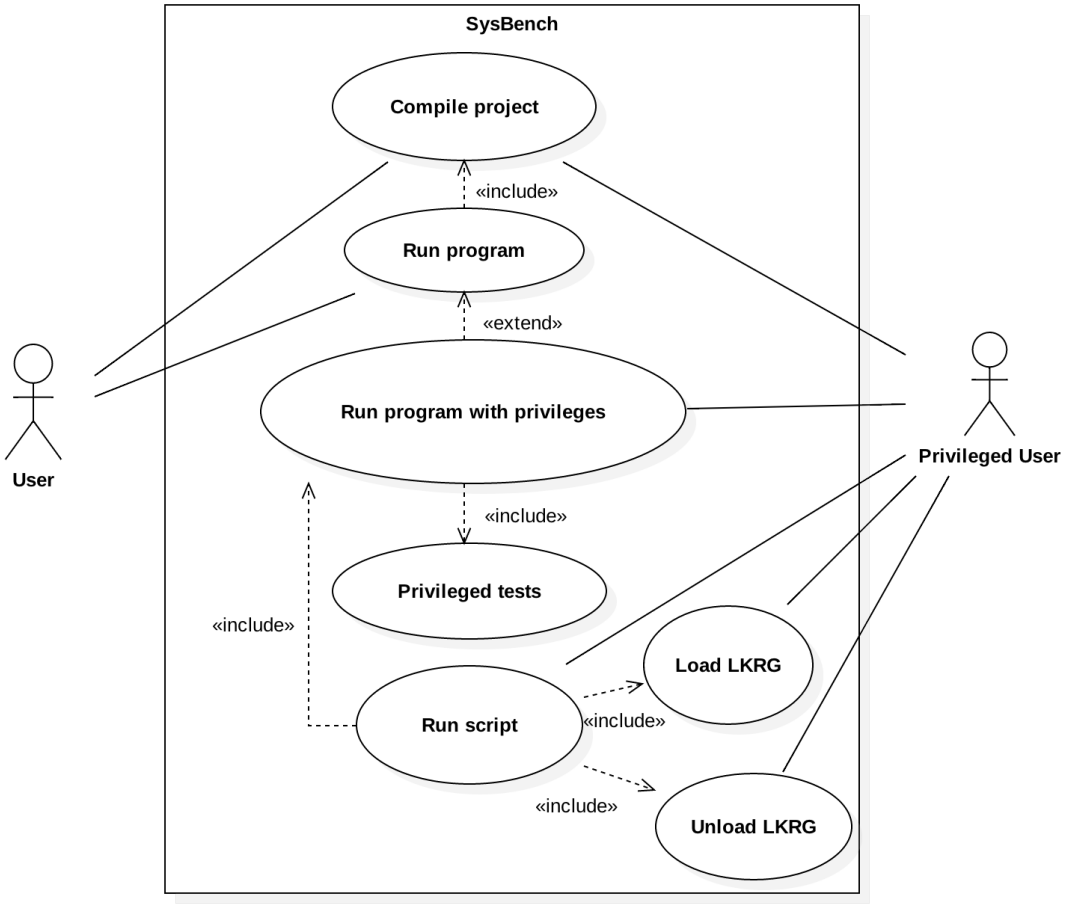

Figura 3.2: Diagramma UML dei casi d'uso di SysBench.

Una volta compilato il progetto, è possibile procedere con due diverse modalità di esecuzione:

1. normal mode (*./sysbench ncycle filename*);
2. sudo mode (*sudo ./sysbench ncycle filename*)

dove ncycle corrisponde al numero di volte che una singola chiamata di sistema viene invocata, mentre filename è il nome del file in cui verrano salvati i risultati all'interno della cartella *output*, creata automaticamente nel caso non fosse già presente.

La differenza risiede nei test delle funzionalità di caricamento e rimozione modulo, ovvero *testMoule()* (il quale richiama le system call *insert_module()* e *delete_module*); non è infatti possibile eseguire con successo questo test se non si hanno i privilegi di root, in quanto per scelta di sicurezza del sistema queste system call non possono modificare i moduli caricati nel kernel se eseguite da un normale utente (non a caso anche i comandi *insmod* e *rmmod* mostrati nel Capitolo I necessitano di tali privilegi).

Ad esclusione di questo test, tutti gli altri sono effettuati con successo in entrambe le modalità e falliscono solamente nel caso vi sia un errore interno al sistema. Nonostante l'utilizzo di sudo permetta di effettuare operazioni ritenute malevole da LKRG quindi non buone per il sistema, in questo progetto tutte le system call testate che prevedono un'alterazione di un parametro del processo si limitano ad impostare come valore dell'attributo quello corrente, in maniera tale da terminare con successo: ad esempio la *setuid()*, utilizzata per cambiare l'identificatore dell'utente che ha lanciato il processo, imposta tale parametro al valore ritrovato in seguito alla system call opposta, ovvero *getuid()*.

Vi è inoltre un'altra modalità d'esecuzione che prevede il caricamento e rimozione automatica di LKRG tra una valutazione di SysBench ed un'altra. Nel progetto è presente infatti uno script che richiede i privilegi di root per essere eseguito (per il motivo appena spiegato), grazie al quale viene eseguito il benchmark un numero di volte a piacere dall'utente. Si è infatti osservato che la differenza tra un'esecuzione singola di SysBench con ncycle=50 e 50 esecuzioni con ncycle=1 presentano risultati completamente differenti introdotti nel prossimo capitolo. Questa differenza è dovuta dalle ottimizzazioni e salvataggio in memoria cache delle funzioni invocate che permettono la loro immediata esecuzione, risparmiando al programma la loro ricerca nella tabella delle system call.
È possibile eseguire lo script da riga di comando lanciando
sudo ./script.sh ntimes ncycle path/to/p_lkrg.ko dove:

- ntimes è il numero di volte che verrà effettuato il benchmark;
- ncycle è il numero di chiamate alle singole system call in ogni benchmark;
- path/to/p_lkrg.ko è il percorso del modulo compilato nel proprio sistema.

Nonostante i risultati ottenuti siano differenti, la possibilità di eseguire il programma con il parametro ncycle maggiore di 1 è stata volutamente lasciata, perchè i valori non solo sono interessanti per futuri sviluppi del progetto, ma anche per un'analisi delle ottimizzazioni di un calcolatore esterna a questo elaborato.

Capitolo 4

Analisi delle performance

In questo capitolo interamente tecnico vengono presentati i risultati ottenuti in seguito alla valutazione di tre sistemi Linux differenti. Essendo tra le versioni più conosciute ed utilizzate al momento, si è deciso di effettuare il benchmark in Ubuntu, Debian e Mint in ambiente virtuale utilizzando come software per la creazione delle macchine VirtualBox. Si definisce il computer fisico *host*, in quanto mette a disposizione il proprio spazio e l'hardware per virtualizzare il sistema definito *guest* creato con VirtualBox.

Al fine di ottenere risultati confrontabili è necessario che durante la fase di testing non solo l'host rimanga nella medesimo stato per ogni guest, ovvero stessi programmi/processi in esecuzione, memoria, cpu disponibile ed altri valori, ma anche i guest devono eseguire il benchmark con le stesse risorse allocate.

Nella tabella sottostante vi sono le specifiche dei sistemi utilizzati:

Tipo	**Modello**	**Sistema Operativo**	**Kernel**	**RAM(GB)**	**CPU**	**Core**
Host	MacbookPro (2015)	High Sierra v10.13.6	Darwin v17.7.0	16	IntelCore i7-4770HQ 2.2GHz	4
Guest	""	Ubuntu v16.04.5	Linux v4.15.0-29-generic	2	""	1
Guest	""	Debian v4.04.5	Linux v4.9.0-7-amd64	2	""	1
Guest	""	Mint v19	Linux v4.15.0-20-generic	2	""	1

Tabella 4.1: Specifiche dei sistemi utilizzati

Nonostante nella pagina di LKRG l'autore proponga gli esempi di rilevazione software solamente in ambiente Ubuntu v16.04, si è deciso di effettuare un'analisi anche nelle altre due distribuzioni, per poter osservare in che modo reagissero al caricamento del modulo e soprattutto se nella stessa maniera, oppure con qualche differenza. Dovendo testare system call comuni a tutte le distribuzioni Linux, non vi è alcun problema di compatibilità nell'utilizzo di SysBench nei tre guest.

In alcune verifiche effettuate è risultata una netta differenza di tempi d'esecuzione delle funzioni con LKRG attivo e non, per cui volendo confrontare dei dati così diversi si è scelto di utilizzare una scala logaritmica per l'asse delle ordinate (y) o delle ascisse (x) a seconda della tipologia di grafico. È importante sottolineare questo dettaglio, perchè la lettura del grafico è leggermente differente: mentre solitamente osservando l'asse y notiamo un aumento caratterizzato da step lineari (esempio: 1, 2, 3, ..., 10), utilizzando la scala logaritmica la distribuzione dei valori è differente, in quanto la distanza tra i punti è calcolata utilizzando il logaritmo secondo la legge $\lambda(OA) = \log(a)$ che rappresenta a distanza tra un punto di coordinata (0, a) e l'origine dell'asse y O è data da .

Prendiamo come esempio la scala logaritmica in base 10 utilizzata in questo elaborato osservando la figura sottostante in cui vi sono riportate alcune potenze:

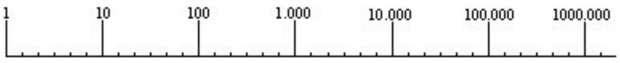

Figura 4.1: Scala logaritmica di base 10.
Fonte: https://en.wikipedia.org

Il valore *1* è calcolabile come la distanza tra il punto d'origine O e sé stesso, ovvero: $\lambda(OO) = 0 => \log(x_0) = 0 => x_0 = 1$. Ugualmente il valore *10* che rappresenta il primo step è dato da $\lambda(OA) = 1 => \log(x_1) = 1 => x_1 = 10$.

Essendo abituati alla scala lineare, verrebbe da pensare che il punto medio tra l'1 ed il 10 di distanza 0.5 dall'origine rappresenti il valore 5; in realtà nella scala logaritmica applicando il calcolo si ottiene $\lambda(OM) = 0.5 => \log(x_m) = 0.5 => x_m = 1/\sqrt{10} \approx 3.16$.

In questa maniera è vero che si perde un po' di precisione del dato, ma è possibile raffigurare valori estremamente differenti, persino con ordini di grandezza doppi o tripli.

Al fine di una buona lettura, si ricorda che in ogni grafico sono riportati i parametri *ncycle* ed *nproc*, i quali corrispondono al numero di singole system call invocate e processi creati (esempio: ncycle=100 significa che la system call X è stata eseguita 100 volte). Inoltre nei grafici sono stati utilizzati i simboli matematici per indicare:

- $\boldsymbol{\sigma}$: la deviazione standard, ovvero l'indice di dispersione dei dati rispetto la media aritmetica;
- $\overline{\boldsymbol{x}}$: per indicare il tempo di esecuzione medio della singola system call, calcolato dividendo i tempi ottenuti per il numero di system call invocate.

Infine, i risultati presentati sono stati ricavati mediante diversi test, in alcuni dei quali certe system call hanno un tempo d'esecuzione pari a 0. Per coerenza si è deciso di non rimuovere tali informazioni nei grafici, nonostante possa sembrare leggermente più complessa la lettura dei valore lungo l'asse delle y. Pertanto quando lungo tale asse è indicato il valore $0.00e^{+0}$, si prenda come riferimento per i valori leggermente superiori l'ordine di grandezza $1.00e^{-7}$, il secondo valore più piccolo misurato.

4.1 Test in Ubuntu

L'analisi di LKRG in Ubuntu ha presentato risultati interessanti, soprattutto confrontando le diverse modalità d'esecuzione del programma. Iniziamo osservando il grafico sottostante, ottenuto eseguendo lo script mediante il quale avvengono 50 esecuzioni consecutive di SysBench con ncycle=1 (e non una singola esecuzione con ncycle=50 che verrà discussa in seguito), salvando i risultati

in altrettanti file di output. In questo modo non vi possono essere ottimizzazioni dovuti alla cache, in quanto essendo una memoria veloce e piccola in cui vengono memorizzati i dati recentemente usati dalla memoria principale (RAM), ogni volta che la system call viene invocata il sistema salva le informazioni relative a tale funzione nella cache, ma non ne farà uso per il semplice motivo che il programma richiama la funzione una singola volta, mentre come vedremo in seguito tale memoria incide sul tempo d'esecuzione nel caso la stessa esecuzione richiami la system call più volte.

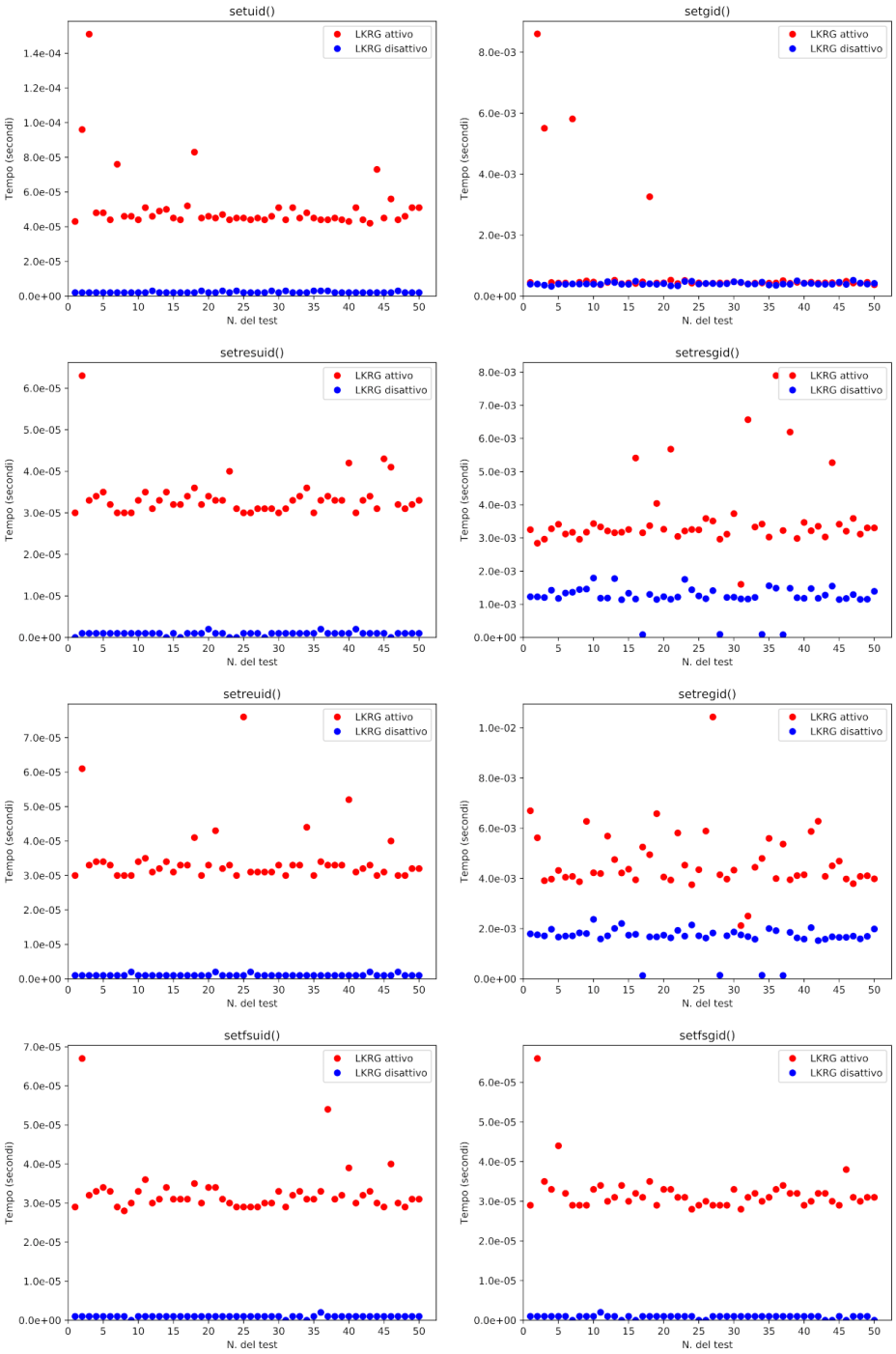

Figura 4.2: Benchmark funzioni *setX* con ncycle=1 (Ubuntu).

Nella Figura 4.2 sono mostrati i risultati dei test delle funzionalità indicate con *setX()* che alterano i parametri del processo, dove X è il parametro stesso (esempio: uid). Si noti come l'unica system call il cui tempo d'esecuzione non è rilevantemente influenzato dalla presenza di LKRG sia la *setgid()*, i cui risultati mostrano come rimanga circa costante ad eccezione di qualche test.

Per tutte le altre funzioni, si osserva che la differenza del tempo d'esecuzione con il modulo caricato e modulo non caricato è elevata, passando da un'ordine di grandezza pari a $10^{-6} \approx 0$ nel grafico a 10^{-5} tranne nella *setregid()* e *setresgid()* in cui il tempo aumenta ma di fattore ≈ 2. Tali tempi per il calcolatore sono pressappoco irrilevanti, nonostante graficamente possano avere un impatto diverso e si potrebbe pensare che sia totalmente sconsigliato l'utilizzo di LKRG. Di seguito è riportata la tabella in cui vengono indicate la media e la deviazione standard per ognuna delle system call nel grafico.

SystemCall	$\overline{x}$ **loaded**	$\overline{x}$ **unloaded**	σ **loaded**	σ **unloaded**
setuid()	5.130e-05	2.200e-06	1.763e-05	4.000e-07
setgid()	8.679e-04	4.079e-04	1.549e-03	4.176e-05
setresuid()	3.366e-05	9.200e-07	5.152e-06	4.400e-07
setresgid()	3.577e-03	1.210e-03	1.052e-03	3.700e-04
setreuid()	3.458e-05	1.100e-06	8.183e-06	3.000e-07
setregid()	4.652e-03	1.646e-03	1.234e-03	4.758e-04
setfsuid()	3.270e-05	9.600e-07	6.306e-06	2.800e-07
setfsgid()	3.212e-05	8.400e-07	5.548e-06	4.176e-07

Tabella 4.2: Dati benchmark funzioni *setX* con ncycle=1 (Ubuntu)

Si osservi l'enorme differenza che vi è non solo tra il tempo medio della singola chiamata, ma anche tra la deviazione standard, ovvero l'indice di dispersione statistico che indica quanto mediamente i dati si discostano dalla media stessa. La differenza tra la deviazione standard con LKRG caricato e non è sempre almeno di un ordine di grandezza, ad eccezione della prima funzione in cui è persino due ordini. L'incremento dei tempi d'esecuzione delle system call suggeriscono l'effettivo intervento del modulo, il quale compiendo i propri controlli d'integrità aggiunge sicuramente ulteriore tempo alla chiamata a funzione.

I grafici in Figura 4.3 e i dati nella Tabella 4.3 riportano i risultati delle restanti system call testate nelle stesse medesime condizioni. Mentre i grafici delle funzionalità *open()*, *fork()* e *delete_module()* presentano una distribuzione dei dati simili a quelle analizzate in precedenza, quella della *insert_module()* è molto differente, in quanto i tempi variano ragionevolmente in entrambe le situazioni di testing. Mediamente il tempo d'esecuzione con LKRG è superiore, ma come si può notare talvolta può impiegare più tempo quando il modulo non è presente, in quanto incidono numerosi fattori nell'esecuzione di questa chiamata legati al caricamento dinamico di un modulo nel kernel; infatti è interessante osservare che la deviazione standard misurata è maggiore quando il modulo non è caricato. È dunque possibile, come nel nostro caso, che nonostante LKRG sia attivo il sistema impieghi meno tempo a caricare un modulo rispetto a quando non lo sia.

SystemCall	$\overline{x}$ loaded	$\overline{x}$ unloaded	σ loaded	σ unloaded
open()	3.398e-05	1.100e-06	5.006e-06	3.000e-07
fork()	3.518e-05	1.260e-06	4.117e-06	4.386e-07
execve()	7.740e-06	6.300e-06	2.234e-06	1.389e-06
insert_module()	9.532e-02	6.940e-02	1.180e-02	1.587e-02
delete_module()	3.226e-05	9.400e-07	4.837e-06	4.200e-07

Tabella 4.3: Dati benchmark restanti funzioni con ncycle=1 (Ubuntu)

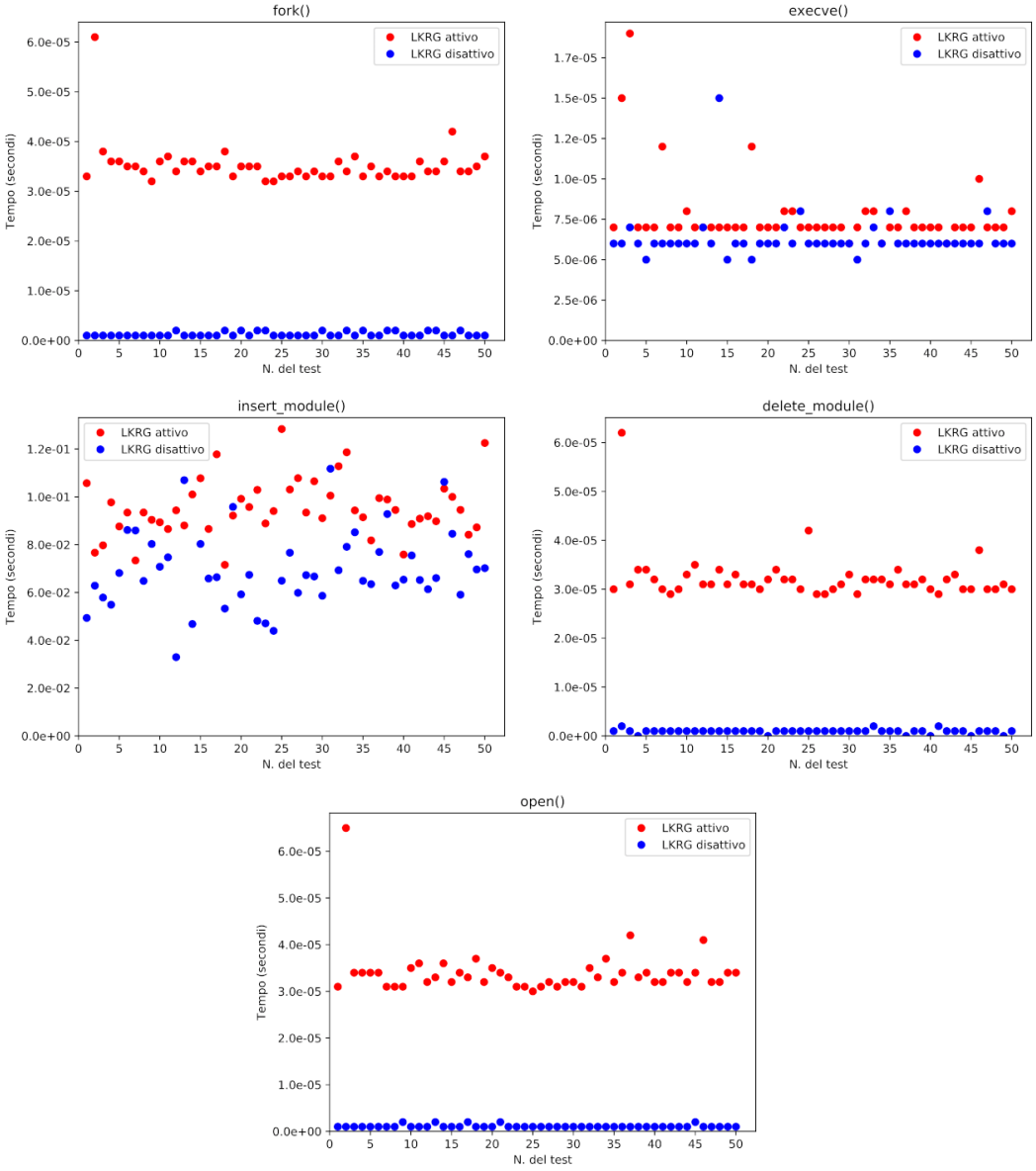

Figura 4.3: Benchmark restanti system call con ncycle=1 (Ubuntu).

Anche la *execve()* ha un range di valori differenti dalle altre system call, nonostante la distribuzione sia molto simile. Sicuramente LKRG aggiunge del tempo effettuando i suoi controlli, ma la parte più consistente di questa funzione è il reset di memoria del processo corrente, effettuando la sovrascrittura delle

vecchie risorse con delle nuove dedicate al comando indicato come parametro. È dunque possibile, come nel nostro caso, che nonstante LKRG sia attivo il sistema impieghi meno tempo ad allocare le risorse al nuovo processo, per vari motivi di scheduling, disponibilità delle risorse o ottimizzazioni. Mediamente si può affermare che il tempo misurato con LKRG è superiore di circa il 22% il tempo misurato senza.

In generale i dati nella Tabella 4.3 sono consistenti con quelli nella Tabella 4.2: il tempo di certe funzionalità come la rimozione del modulo cambia di fattore ≈ 10, mentre in altre la differenza è minore.

Analizziamo ora come cambiano i tempi d'esecuzione lanciando singolarmente il programma SysBench con ncycle=1, 10, 100, 1000 e 10000 per capire perchè i risultati medi ottenuti possano essere così diversi rispetto a più esecuzioni con lo stesso parametro. Più semplicemente, la differenza tra le due esecuzioni si può riassumere nei seguenti comandi:

- *sudo ./script.sh 10000 1 path/to/p_lkrg.jo*, dove 10000 sono le volte che verrà richiamato il programma Sysbench e 1 corrisponde al parametro ncycle;
- *sudo ./sysbench 10000 filename.*

Senza consultare alcun risultato, possiamo aspettarci di ottenere due output differenti, influenzati dalle ottimizzazioni del processore. L'obiettivo dunque è verificare che il valore della singola system call misurato richiamandola 10000 volte all'interno dello stesso programma sia effettivamente inferiore della media dei valori ottenuti mediante 10000 esecuzioni in cui viene eseguita una volta sola.

Iniziamo commentando la Figura 4.4, raffigurante il tempo totale d'esecuzione delle system call nelle 5 esecuzioni con ncycle differente. Per l'asse delle ordinate (y) si è utilizzato una scala logaritmica, al fine di riuscire ad inserire tutti i dati nello stesso grafico, sebbene siano molto discostanti tra loro.

Si può osservare il tempo aggiunto dalla presenza di LKRG semplicemente sottraendo alle barre arancioni le corrispettive blu. In quasi tutte le funzioni ad eccezione della *fork()*, *execve()* e *open()* l'overhead aggiunto dal modulo è significativo ed aumenta sempre di più al variare del numero di chiamate. È da premettere che tali test sono specifici per questa analisi, al fine di comprendere il lavoro del modulo LKRG, ma in una situazione reale è altamente improbabile trovare un programma che effettui così tante volte consecutive la medesima chiamata.

L'unica che potrebbe raffigurare uno scenario reale è la *open()*, in quanto moltissimi programmi lavorano con più file, aprendoli e chiudendoli diverse volte, ma i grafici e le tabelle presenti in questa trattazione suggeriscono che tale system call sia leggermente influenzata da LKRG a differenza di molte altre, tra cui la *insert_module()*. Infatti quest'ultima funzione e la sua opposta *delete_module()* risentono dei controlli d'integrità, aumentando il loro tempo d'esecuzione di circa mezzo ordine di grandezza; se nei primi grafici i loro tempi con LKRG attivo sono inferiori a quelli della *execve()*, negli ultimi in cui ncycle

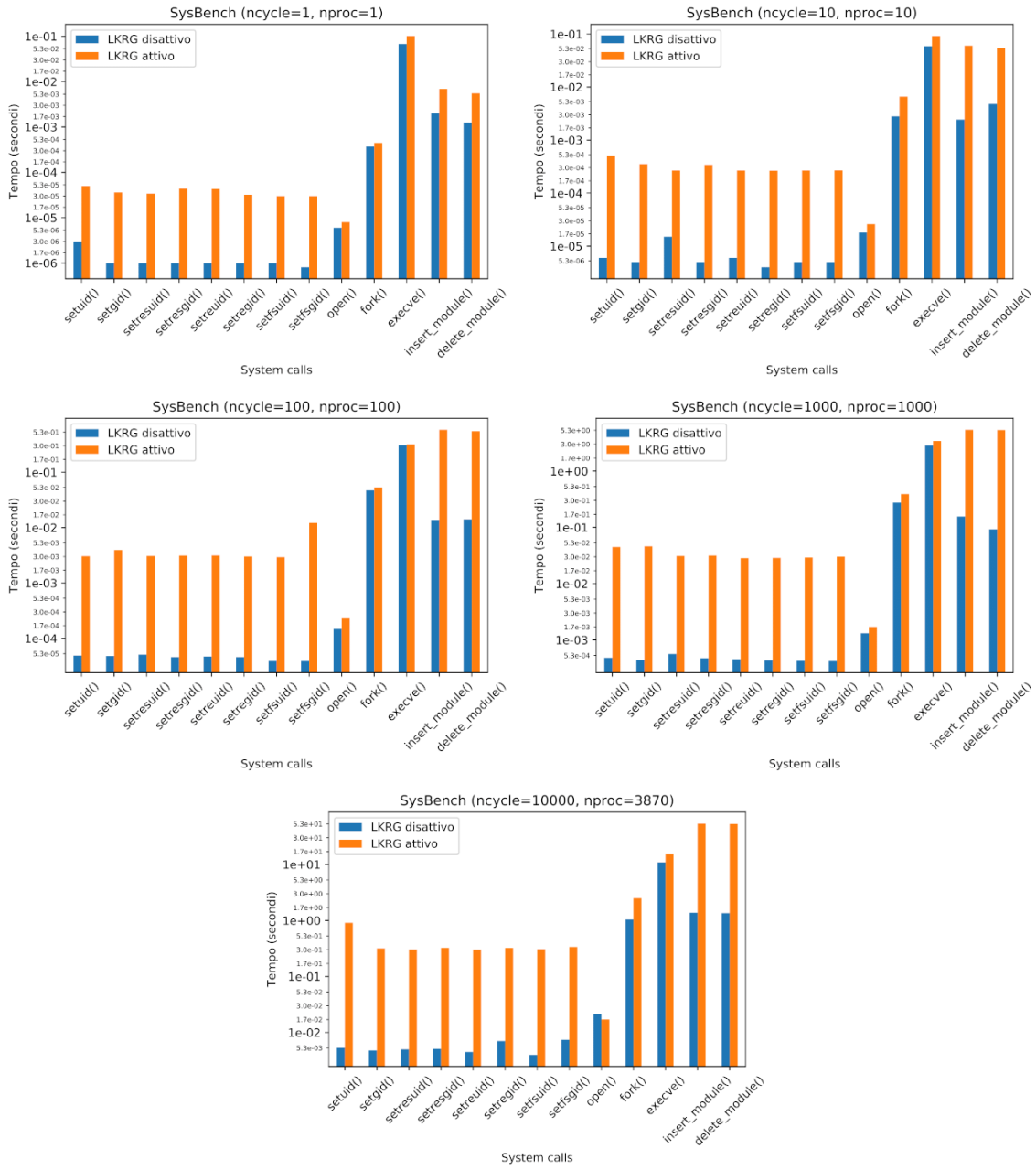

Figura 4.4: Benchmark tempo totale (Ubuntu).

è molto più grande superano qualsiasi tempo d'esecuzione delle altre system call, in quanto si somma ncycle-volte l'overhead aggiunto dal modulo, il quale risulta essere massimo per queste due funzioni. Infine, un'altra differenza facile da notare sono i tempi delle funzioni *setX*; nonostante essi siano molto più piccoli delle altre funzionalità, si può affermare che in percentuale sono quelle maggiormente influenzate, in quanto il rapporto tra i tempi è pari a $1.5x10^1$ per la *setuid()* e persino maggiore per le altre.

Confrontando i dati della Tabella 4.4 con quelli delle tabelle ottenute precedentemente (Tabella 4.3 e Tabella 4.2) si osserva l'enorme differenza del tempo medio d'esecuzione ($\overline{\boldsymbol{x}}$) delle funzioni in entrambi i casi, potendo pensare che

SystemCall	$\overline{x}$ loaded	$\overline{x}$ unloaded	σ loaded	σ unloaded
setuid()	5.349e-05	1.021e-06	2.007e-05	9.903e-07
setgid()	3.772e-05	5.793e-07	4.858e-06	2.112e-07
setresuid()	3.059e-05	8.148e-07	2.326e-06	3.895e-07
setresgid()	3.467e-05	5.885e-07	4.780e-06	2.065e-07
setreuid()	3.197e-05	5.945e-07	5.749e-06	2.102e-07
setregid()	2.994e-05	6.001e-07	2.144e-06	2.269e-07
setfsuid()	2.919e-05	5.426e-07	1.369e-06	2.319e-07
setfsgid()	4.847e-05	5.724e-07	3.667e-05	1.707e-07
open()	3.263e-06	2.547e-06	2.394e-06	1.749e-06
fork()	5.349e-04	3.333e-04	1.067e-04	7.851e-05
execve()	2.428e-02	1.645e-02	3.877e-02	2.558e-02
insert_module()	5.907e-03	5.380e-04	5.558e-04	7.400e-04
delete_module()	5.430e-03	4.213e-04	8.738e-05	4.404e-04

Tabella 4.4: Dati benchmark con ncycle=1, 10, 100, 1000, 10000 (Ubuntu)

vi sia un errore nella misurazione. In realtà la differenza è dovuta all'insieme delle ottimizzazioni apportate dal processore e dalla memoria cache non trattate in questo elaborato. Una buona idea per uno possibile sviluppo è quella di estendere la trattazione analizzando il comportamento del modulo in processori differenti, capendo le modifiche apportate da ognuno analizzando ogni singolo step d'esecuzione del programma. Non essendo il processore e il compilatore argomenti di tesi, mi limito a mostrare qualche grafico interessante per capire come il tempo medio della singola system call cambi a seconda del numero di volte che viene invocata all'interno del medesimo programma.

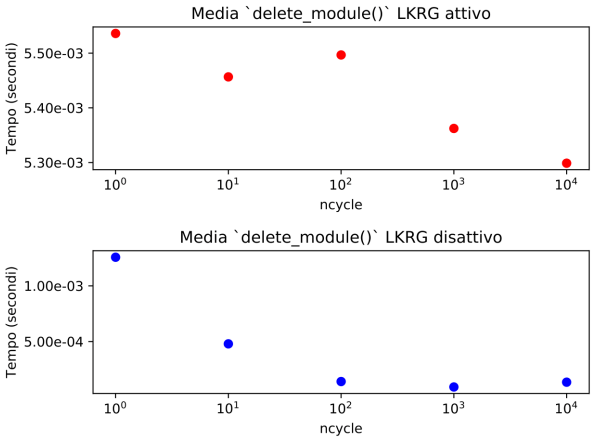

Figura 4.5: Media singole system call nei 5 benchmark parte 1 (Ubuntu).

Ad eccezione dei grafici in Figura 4.8 che presentano valori oscillanti, nei restanti è chiaro l'intervento di un ottimizzatore, grazie al quale in base a quante chiamate vengono effettuate il tempo d'esecuzione medio della singola system call diminuisce in maniera esponenziale. Come scala per l'asse delle x è stata

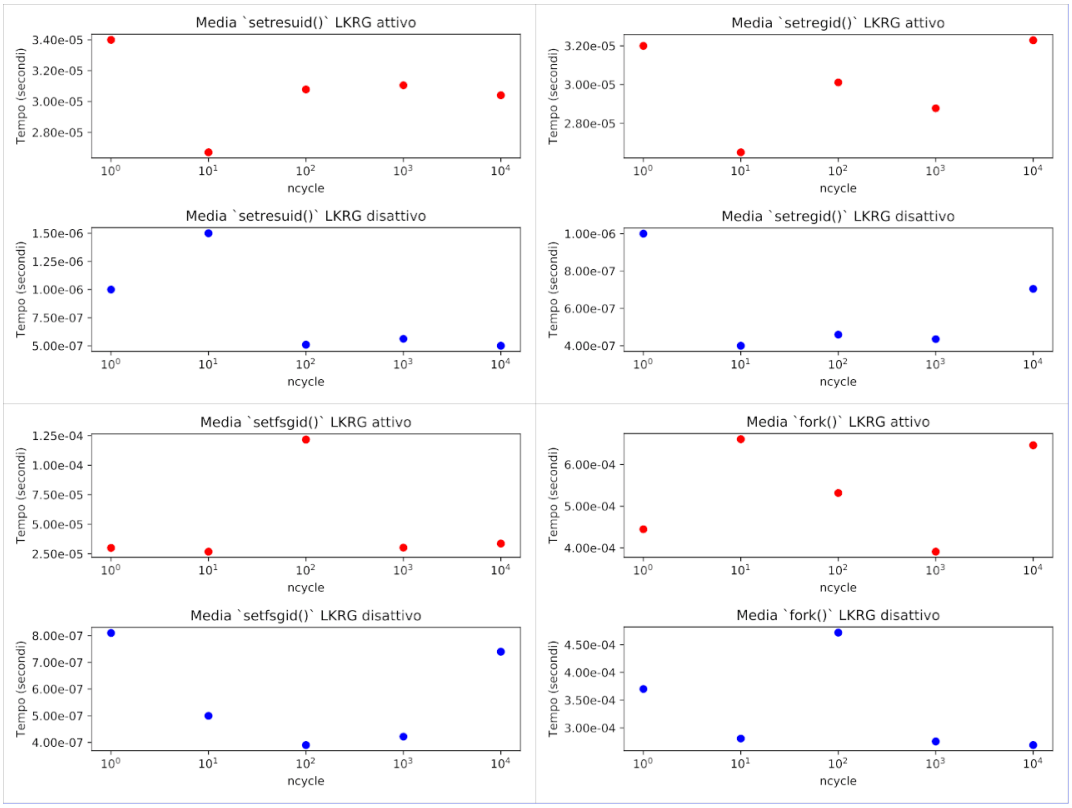

Figura 4.6: Media singole system call nei 5 benchmark parte 2 (Ubuntu).

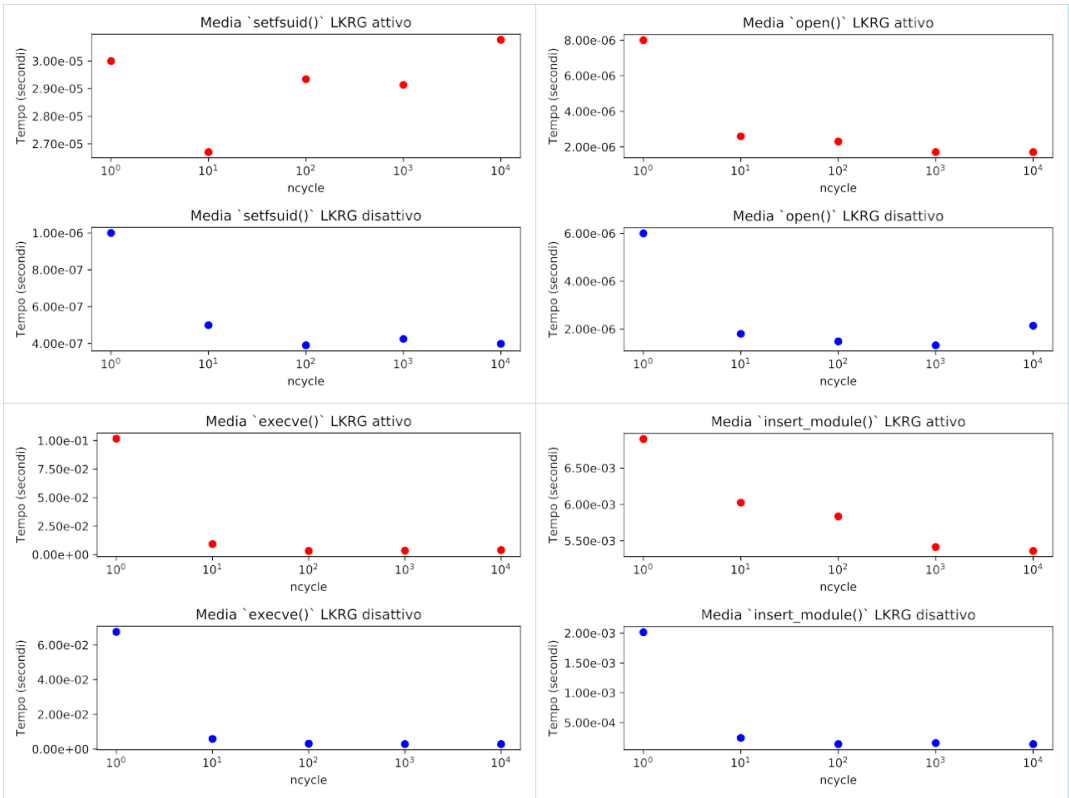

Figura 4.7: Media singole system call nei 5 benchmark parte 3 (Ubuntu).

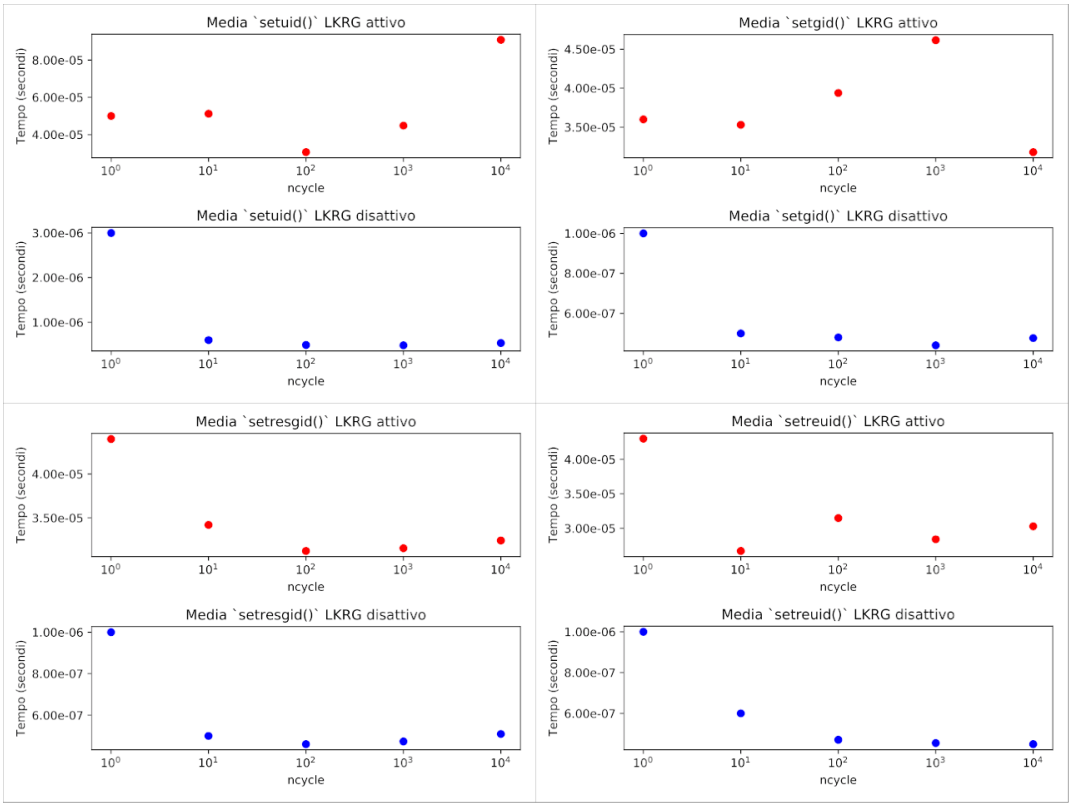

Figura 4.8: Media singole system call nei 5 benchmark parte 4 (Ubuntu).

utilizzata una scala logaritmica per rappresentare il numero tutti i valori assunti da ncycle.

4.2 Test in Debian

In questa sezione sono presentati i risultati della valutazione della seconda distribuzione presa in analisi: Debian. I test in questo ambiente, nonostante presentino varie similitudini con i risultati ottenuti in Ubuntu, suggeriscono un'ottima risposta del sistema al caricamento di LKRG.

Per seguire la stessa logica di presentazione della sezione precedente, iniziamo commentando i risultati ottenuti in seguito a 50 esecuzioni di SysBench con parametro ncycle=1 presentati nei seguenti grafici.

Come si può osservare in Figura 4.9 è chiara la differenza tra i tempi d'esecuzione delle varie system e l'overhead aggiunto da LKRG. Nonostante la dispersione dei grafici sia simile a quelli presentati per Ubuntu, cambia il range dei tempi d'esecuzione: ad esempio per la *setresgid()* osserviamo che i tempi misurati senza il modulo attivo sono di due ordini di grandezza più piccoli rispetto a quelli con LKRG presente, mentre in Ubuntu la differenza è minima, cambia solamente un fattore di proporzionalità pari a 3. La system call che risulta essere meno influenzata anche in ambiente Debian risulta essere la *setgid()*, i cui valori oscillano in un intorno di $1x10^{-4}$ che rispetto ai tempi misurati è relativamente piccolo. Esclusa questa system call e la *setresuid()*, i cui

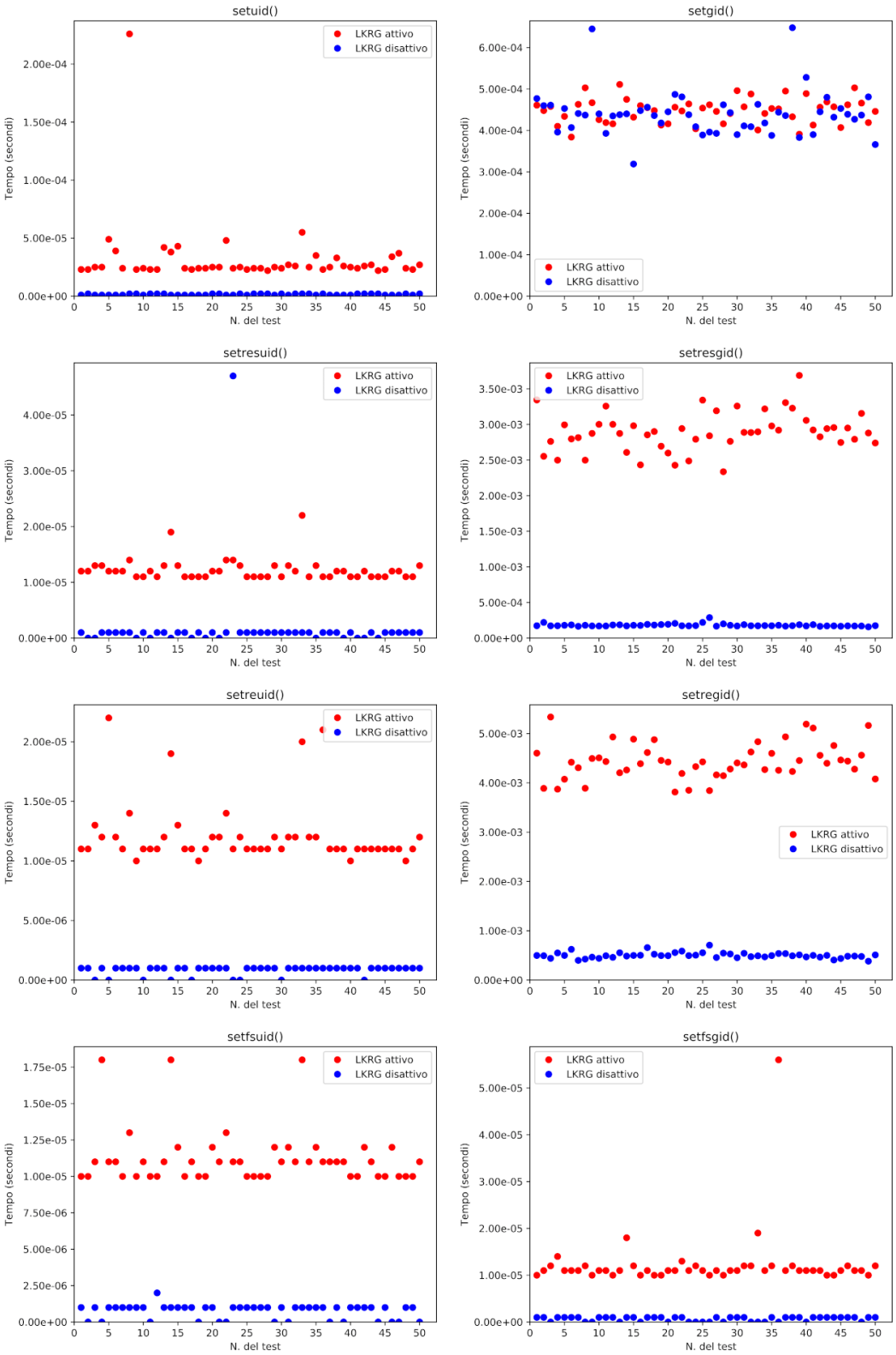

Figura 4.9: Benchmark funzioni *setX* con ncycle=1 (Debian).

test rivelano qualche rallentamento nell'esecuzione della funzione anche senza LKRG, nei grafici di tutte le altre funzionalità si osserva che mentre in presenza

del modulo i tempi campionati sono molto più variabili e dispersi, in sua assenza i valori risultano essere più omogenei ed allineati, sebbene vi sia qualche differenza ma in grandezza minore.

SystemCall	$\overline{x}$ **loaded**	$\overline{x}$ **unloaded**	σ **loaded**	σ **unloaded**
setuid()	3.202e-05	1.460e-06	2.877e-05	4.984e-07
setgid()	4.477e-04	4.402e-04	2.982e-05	5.500e-05
setresuid()	1.220e-05	1.660e-06	1.949e-06	6.492e-06
setresgid()	2.894e-03	1.812e-04	2.685e-04	1.971e-05
setreuid()	1.212e-05	8.200e-07	2.628e-06	3.842e-07
setregid()	4.438e-03	5.018e-04	3.587e-04	5.863e-05
setfsuid()	1.124e-05	7.400e-07	1.893e-06	4.821e-07
setfsgid()	1.228e-05	6.600e-07	6.465e-06	4.737e-07

Tabella 4.5: Dati benchmark funzioni *setX* con ncycle=1 (Debian)

Dalla Tabella 4.5 si evince quanto appena commentato: concentrandosi sulla deviazione standard in entrambe le situazioni, confermiamo appunto che i valori sono più concentrati ed omogenei quando LKRG non è presente nel sistema, mentre una volta caricato i tempi misurati variano in maniera più evidente. Si può affermare dunque che i controlli d'integrità, sebbene non comportino un aumento eccessivo della chiamata, influenzino in maniera variabile l'esecuzione. Per fare un esempio si considerino i dati relativi alla *setuid()*: una deviazione standard di $2.877x10^{-5}$ con LKRG presente è accettabile rispetto alla media $3.202x10^{-5}$ delle chiamate, ma se la si confronta con i valori misurati in assenza del modulo risulta essere non trascurabile, in quanto la deviazione è dell'ordine del 10^{-7}. È necessario prima di giungere a conclusioni considerare questi fattori e queste differenze negli ordini di grandezza dei dati, in modo da comprendere e confrontare con coerenza la variazione dei tempi d'esecuzione delle funzioni.

Sorprendentemente, il grafico della *execve()* in Figura 4.10 rivela che il tempo d'esecuzione è minore quando misurato con il modulo presente, mentre risulta essere leggermente maggiore nel caso opposto, inducendo a concludere che tale system call non sia influenzata da LKRG. La scala dei valori in tale grafico risulta essere meno estesa, suggerendo dunque più omogeneità nei dati rispetto agli altri dove si può osservare che in ogni test delle funzionalità vi è sempre qualche valore che si discosta maggiormente dalla media. Ad esempio si consideri la *fork()*: una misurazione è risultata essere di un'ordine di grandezza e mezzo più grande rispetto a tutte le altre, probabilmente dovuto a qualche ritardo nell'allocazione di risorse per il processo figlio da parte del programma e non a causa di LKRG. Ancora una volta si assiste ad un aumento di due ordini di grandezza del tempo d'esecuzione della *open()* che, come in Ubuntu, varia da $7.200e^{-07}$ in assenza del modulo a $1.424e^{-05}$ una volta caricato.

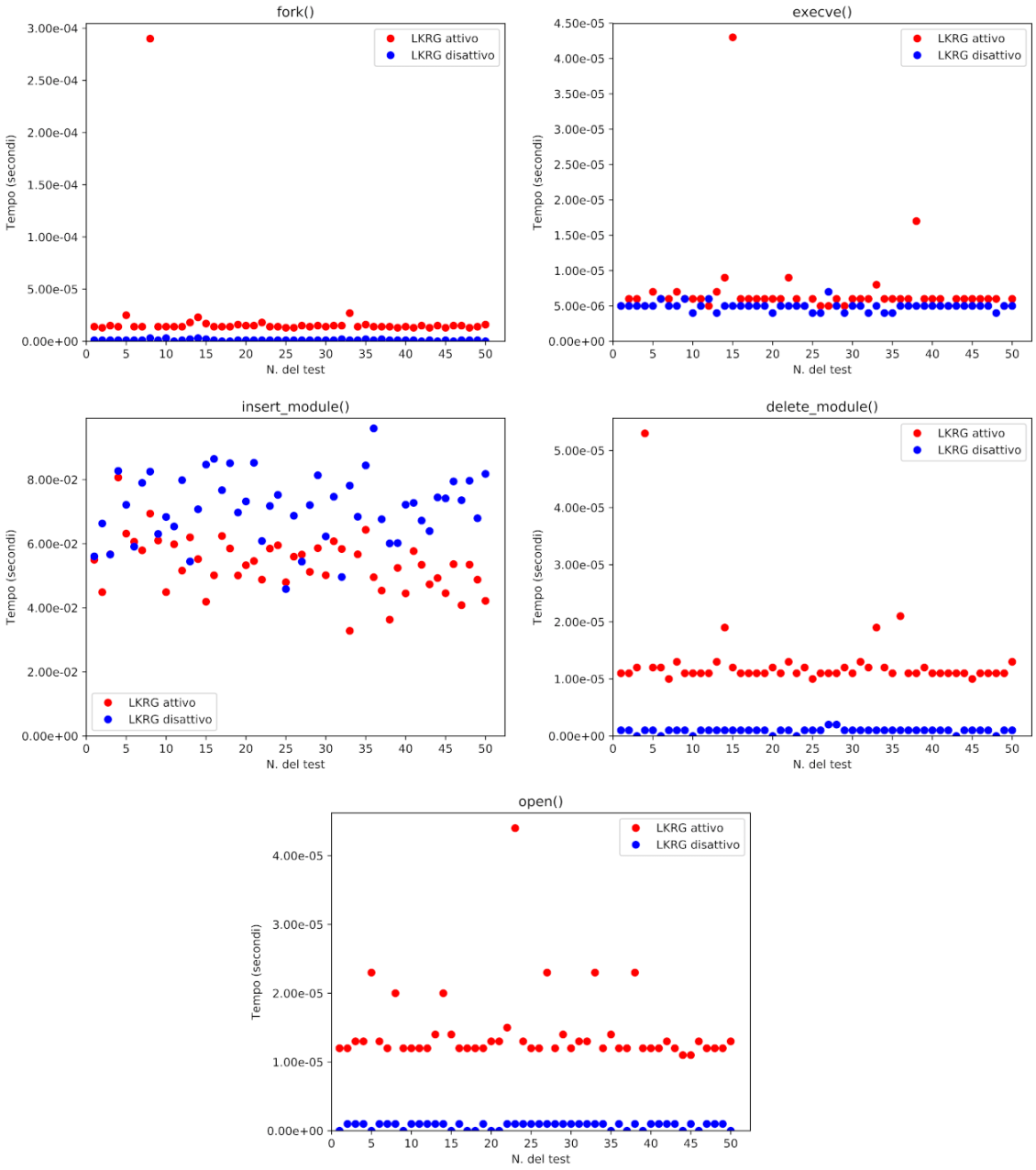

Figura 4.10: Benchmark restanti funzioni con ncycle=1 (Debian).

SystemCall	$\overline{x}$ loaded	$\overline{x}$ unloaded	σ loaded	σ unloaded
open()	1.424e-05	7.200e-07	5.335e-06	4.490e-07
fork()	2.058e-05	1.080e-06	3.859e-05	6.882e-07
execve()	7.000e-06	4.900e-06	5.430e-06	5.745e-07
insert_module()	5.349e-02	7.114e-02	8.344e-03	1.041e-02
delete_module()	1.270e-05	9.000e-07	6.133e-06	4.123e-07

Tabella 4.6: Dati benchmark restanti funzioni con ncycle=1 (Debian)

Nella tabella Tabella 4.6 si osservi come le system call *open()* e *delete_ module()* siano quelle più influenzate dal modulo, in quanto i valori medi si innalzano di fattore 1000, passando da tempi come $7.200e^{-07}$ e $9.000e-07$ a $1.424e-05$ e $1.270e-05$. Coerentemente con quanto detto in prececenza, non solo la media della singola chiamata alla funzione *execve()* risulta essere leggermente maggiore in assenza del modulo, ma anche la deviazione standard (10 volte minore).

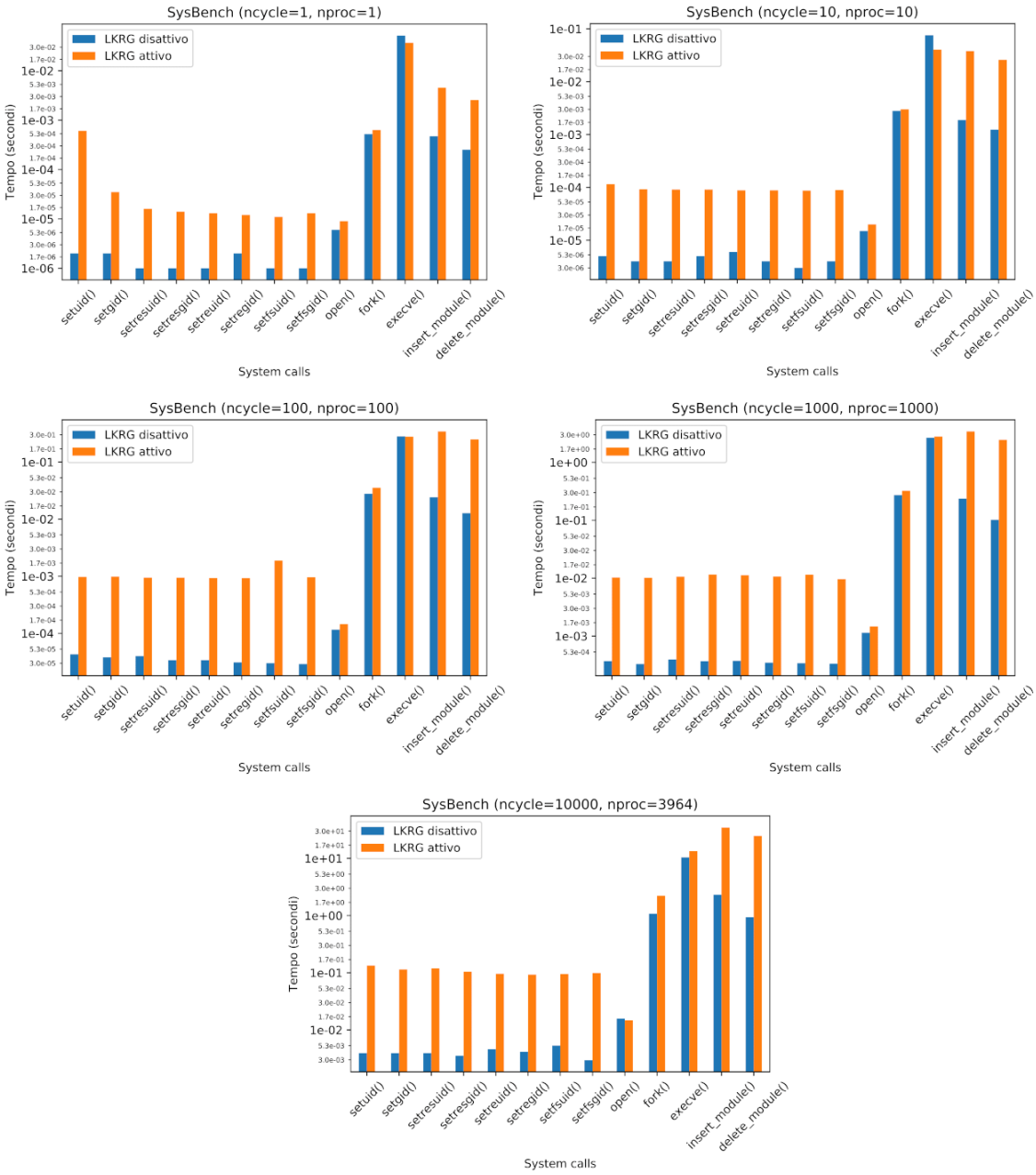

Figura 4.11: Benchmark tempo totale (Debian).

Procediamo ora con la medesima analisi effettuata in Ubuntu, ovvero il tempo d'esecuzione della singola system call nel caso venisse richiamata più volte all'interno del programma. Il test è stato effettuato ugualmente, con

ncycle che assume i valori 1, 10, 100, 1000 e 10000 ed il sistema che si trova nello stesso stato in cui era la macchina Ubuntu. L'obiettivo è sempre quello di osservare il tempo medio d'esecuzione della singola system call per dedurre se anche in questo sistema vengono apportate ottimizzazioni.

In Figura 4.11 si osserva il tempo totale d'esecuzione delle system call nei 5 scenari con ncycle differente. Iniziando a valutare il test con ncycle=1 che rappresenta inoltre i test discussi fino ad ora, si nota come LKRG influenzi maggiormente le funzioni *setX()* e quelle inerenti alle operazioni con i moduli, le quali variano maggiormente rispetto alla *open()*, *fork() e la* execve(). Quest'ultima soprattutto presenta tempi d'esecuzione maggiori in assenza del modulo nei test in cui viene richiamata un numero relativamente piccolo di volte, mentre più cresce ncycle più i due tempi tendono a differire per veramente poche unità sembrando quasi pareggianti. Vi è una lieve differenza anche nelle altre due funzionalità, le quali però mantengono il rapporto $\frac{Tattivo}{Tdisattivo}$ quasi costante. Sicuramente l'utente lanciando il programma è in grado di percepire la differenza d'esecuzione della *insert_module()* e *delete_module()*, in quanto in ordini di grandezza sono quelle con la differenza più alta (persino 30 volte più alta nei test con ncycle=10000), ma quelle che in percentuale al loro tempo d'esecuzione in assenza del modulo hanno subito maggiori rallentamenti sono le *setX()* come già commentato.

SystemCall	$\overline{x}$ **loaded**	$\overline{x}$ **unloaded**	σ **loaded**	σ **unloaded**
setuid()	1.304e-04	7.380e-07	2.383e-04	6.326e-07
setgid()	1.512e-05	7.000e-07	9.966e-06	6.505e-07
setresuid()	1.142e-05	5.171e-07	2.494e-06	2.415e-07
setresgid()	1.089e-05	5.119e-07	1.766e-06	2.507e-07
setreuid()	1.038e-05	5.544e-07	1.544e-06	2.403e-07
setregid()	9.999e-06	6.941e-07	1.179e-06	6.540e-07
setfsuid()	1.188e-05	4.941e-07	3.586e-06	2.671e-07
setfsgid()	1.020e-05	4.634e-07	1.438e-06	2.712e-07
open()	3.077e-06	2.274e-06	2.969e-06	1.871e-06
fork()	4.332e-04	3.241e-04	1.349e-04	9.801e-05
execve()	9.963e-03	1.340e-02	1.345e-02	1.906e-02
insert_module()	3.727e-03	2.734e-04	4.233e-04	1.006e-04
delete_module()	2.510e-03	1.394e-04	5.203e-05	5.722e-05

Tabella 4.7: Dati benchmark con ncycle=1, 10, 100, 1000, 10000 (Debian)

Tralasciando la precisione dei valori, l'analisi da effettuare è la medesima di quella inerente al sistema Ubuntu, ovvero da questa tabella si osservano valori che porterebbero a pensare ad un errore di misurazione. In realtà, osservando i grafici sottostanti si osserva come il tempo d'esecuzione medio delle singole system call diminuisca all'aumentare del numero di volte che vengono testate.

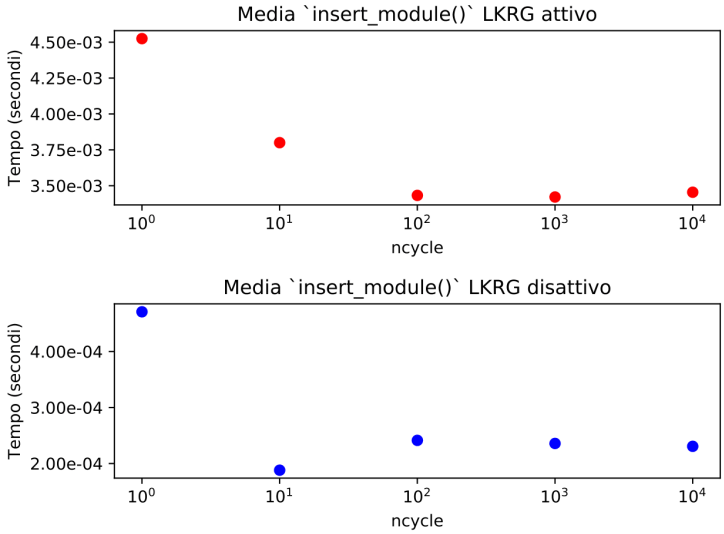

Figura 4.12: Media singole system call nei 5 benchmark parte 1 (Debian).

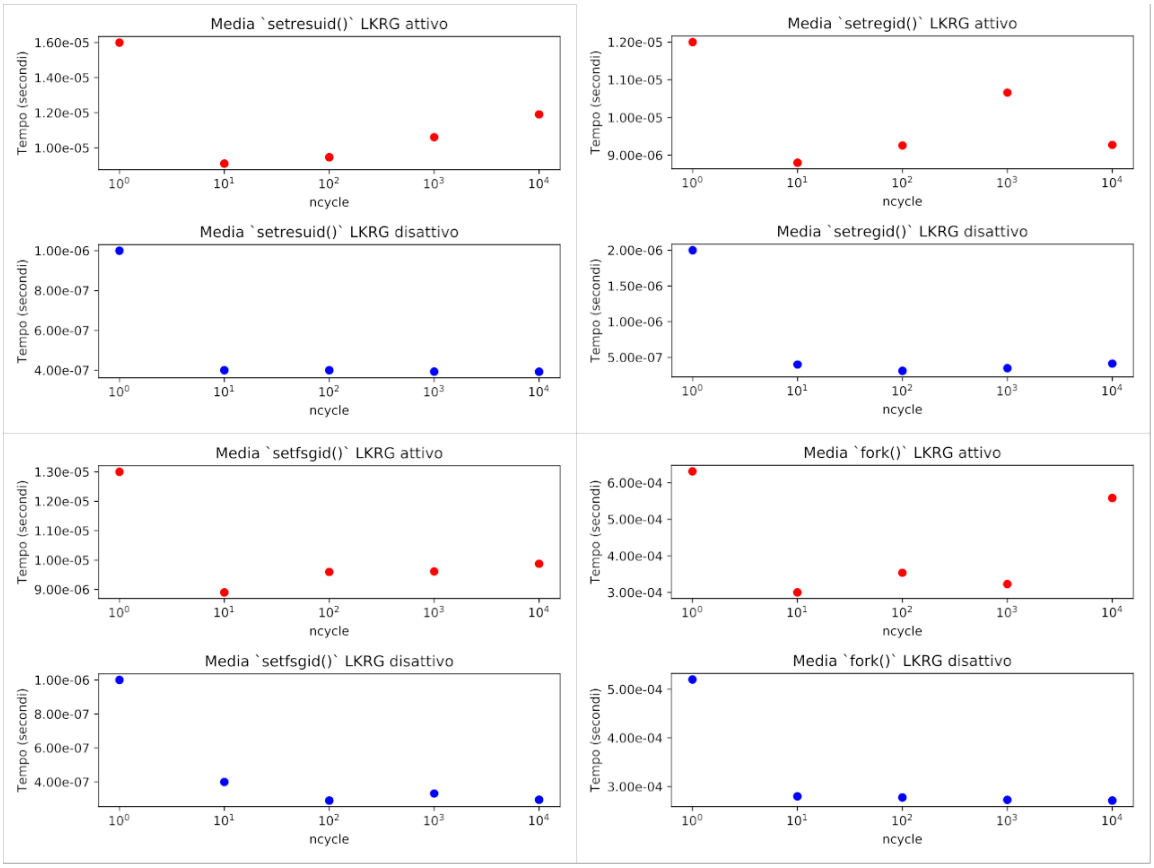

Figura 4.13: Media singole system call nei 5 benchmark parte 2 (Debian).

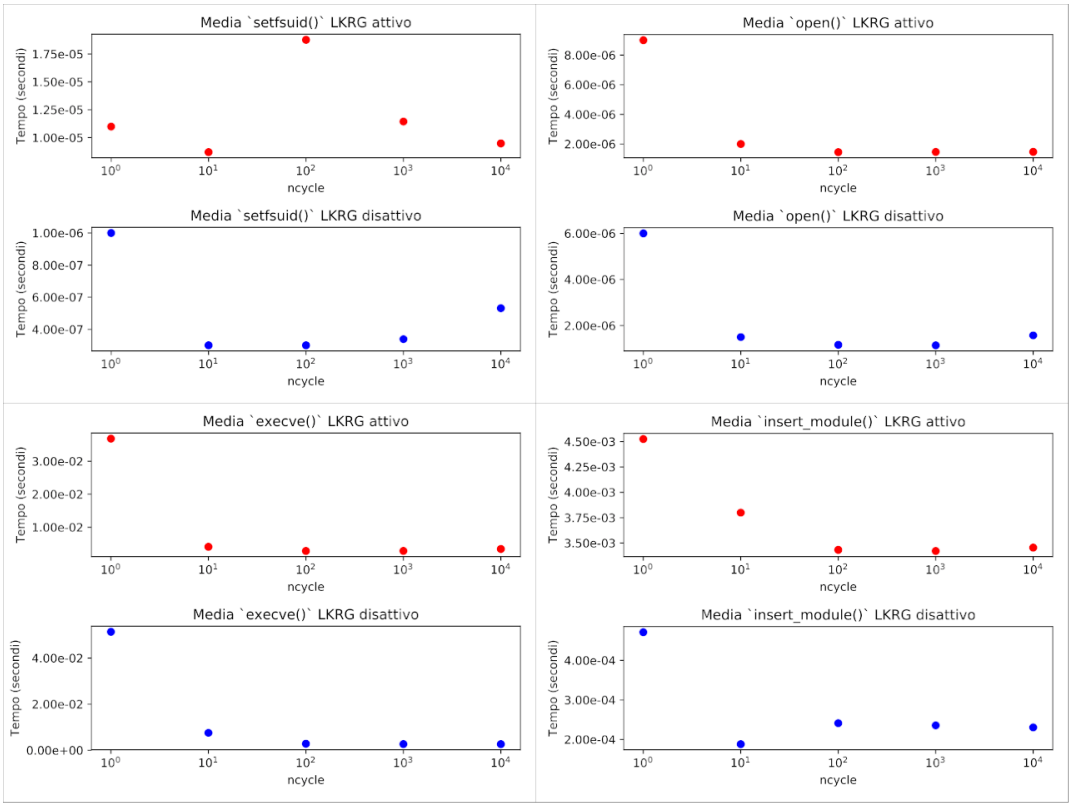

Figura 4.14: Media singole system call nei 5 benchmark parte 3 (Debian).

Nei grafici rappresentati in queste quattro figure è immediato notare l'andamento esponenziale dei tempi delle singole chiamate di sistema quando il modulo non è attivo. A differenza dei tempi misurati in Ubuntu, questi valori sono più stabili e la curva esponenziale che formano è ancora più evidente e precisa. I valori misurati con LKRG caricato nel kernel rimangono invece abbastanza instabili per la maggior parte delle system call, mentre per altre come la *open()* mantengono un andamento esponenziale simile a quelli misurati in assenza del modulo, sebbene siano leggermente più elevati.

4.3 Test in Mint

L'ultimo sistema da testare, nonchè uno tra i più conosciuti ed utilizzati, è Linux Mint. I test hanno rilevato un comportamento differente del sistema, i cui risultati talvolta convergono a quelli ottenuti in Ubuntu, mentre per certe system call sono intermedi tra quelli di Ubuntu e Debian.

In Figura 4.16 sono riportati i grafici ottenuti mediante la solita esecuzione di SysBench con ncycle=1, come per le altre valutazioni.

Il primo commento in merito ai grafici riguarda il comportamento della funzione *setgid()*: il tempo d'esecuzione di tale funzione, escluso qualche test eccezionale, non varia in maniera clamorosa tra la misurazione con LKRG e senza. Inoltre, anche la *setuid()* e molte altre di queste funzioni presentano risultati più analoghi, sebbene leggermente differenti ma di un fattore irrilevante, a quelli ottenuti in Debian rispetto a quelli in Ubuntu, nei quali vi era più di un test

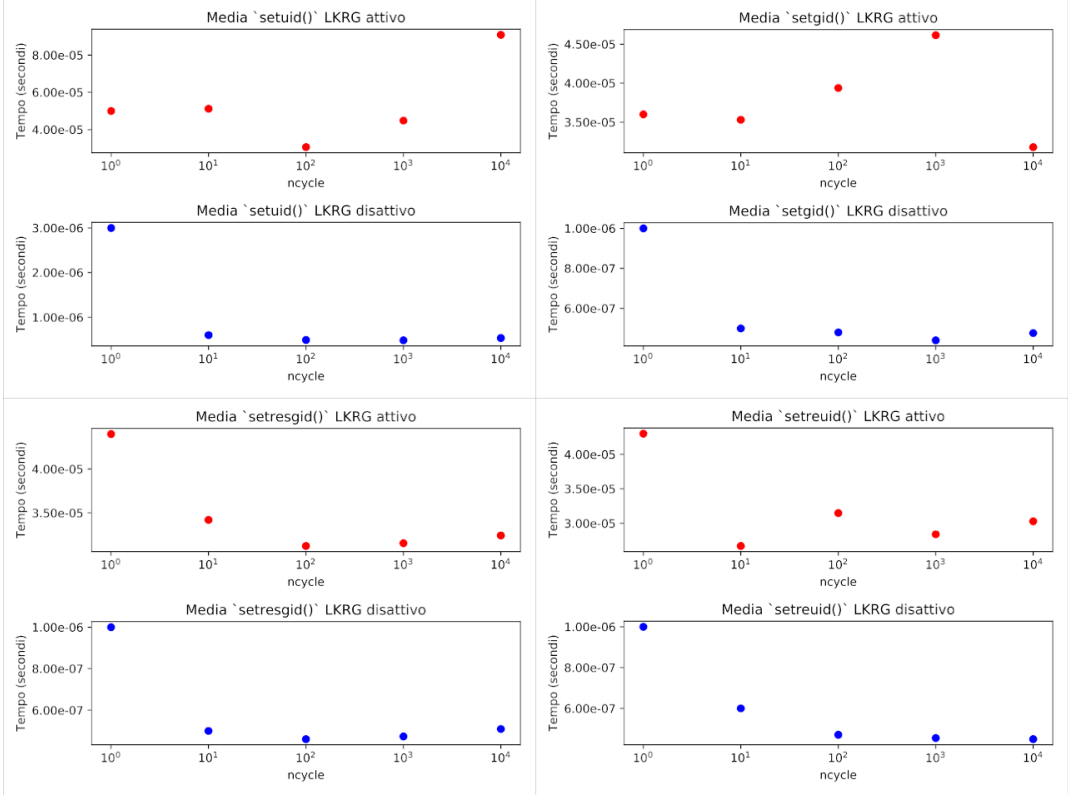

Figura 4.15: Media singole system call nei 5 benchmark parte 4 (Debian).

fuori dalla media. Per altre system call come la *setregid()* e la *setfsuid()* si può affermare l'opposto. La distribuzione dei grafici risulta essere abbastanza chiara come negli altri sistemi, facilitando la lettura e indicando in maniera chiara l'overhead causato da LKRG; si osservi che a seconda della system call il fattore di differenza tra i tempi d'esecuzione è circa 1.5, 3, 10 o 100.

SystemCall	$\overline{x}$ **loaded**	$\overline{x}$ **unloaded**	σ **loaded**	σ **unloaded**
setuid()	3.862e-05	2.060e-06	1.062e-05	4.200e-07
setgid()	4.658e-04	3.863e-04	1.436e-04	5.517e-05
setresuid()	3.212e-05	1.380e-06	4.325e-06	4.854e-07
setresgid()	3.843e-03	1.052e-03	3.305e-04	8.837e-05
setreuid()	2.392e-05	1.120e-06	1.707e-06	1.608e-06
setregid()	4.811e-03	1.473e-03	3.432e-04	1.055e-04
setfsuid()	2.276e-05	6.800e-07	1.871e-06	4.665e-07
setfsgid()	2.278e-05	4.800e-07	3.402e-06	4.996e-07

Tabella 4.8: Dati benchmark funzioni *setX()* con ncycle=1 (Mint)

Nella tabella Tabella 4.8 si noti come la deviazione standard dei valori quando il modulo è attivo sia sempre maggiore rispetto a quella calcolata in sua assenza. Ugualmente, anche la media delle singole chiamate è sempre minore, talvolta persino 100 volte più piccola se prendiamo in considerazione la *setfsuid()* e la *setfsgid()*.

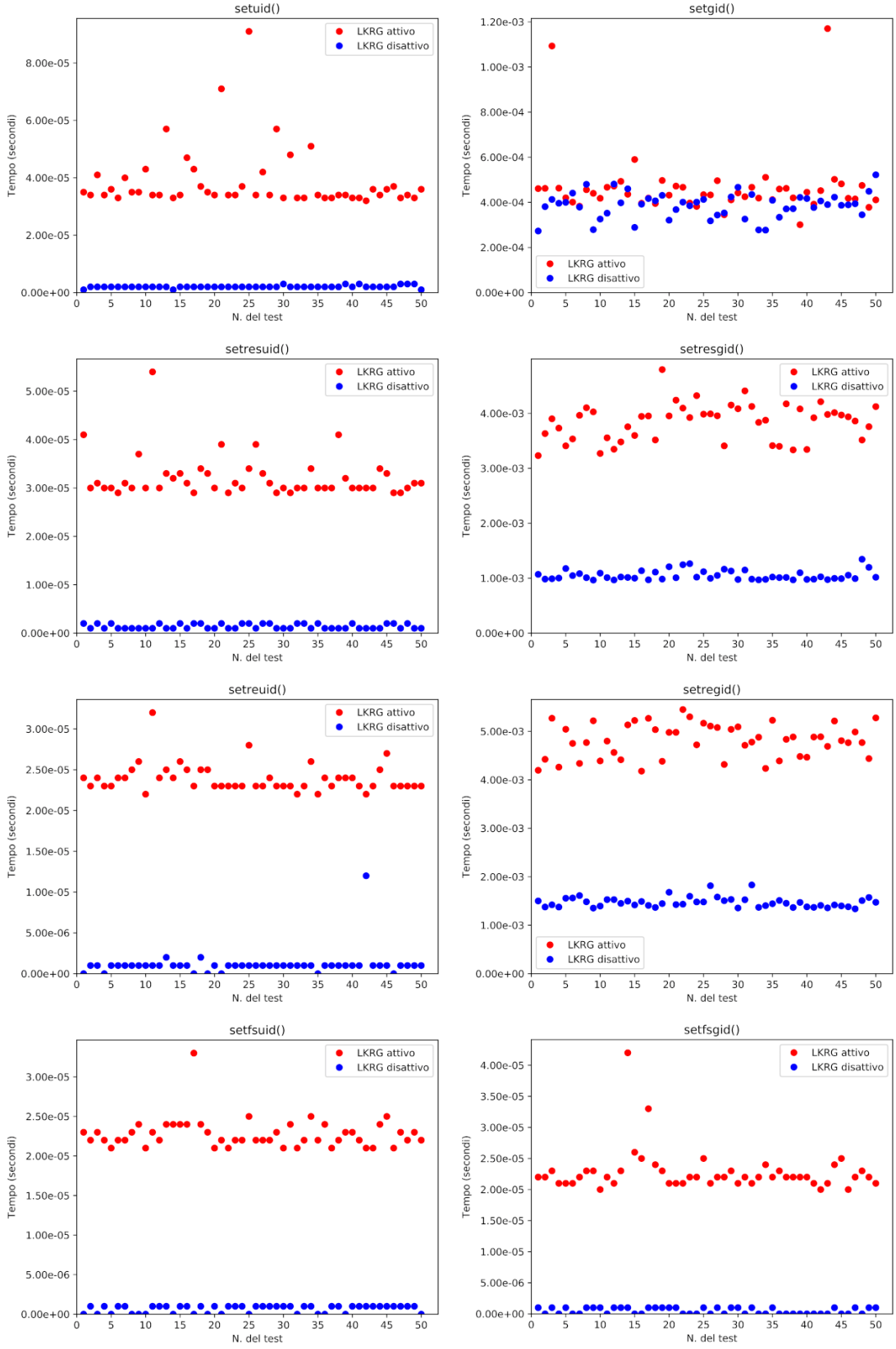

Figura 4.16: Benchmark funzioni *setX* con ncycle=1 (Mint).

Si può dedurre che il sovraccarico di istruzioni aggiunte da LKRG non sia trascurabile se ci atteniamo a considerare i risultati all'interno del loro ordine

di grandezza. Infatti, la maggior parte delle funzionalità nella tabella hanno dei tempi molto differenti.

I risultati delle system call raffigurate in figura Figura 4.17 si avvicinano maggiormente a quelli ottenuti in Ubuntu rispetto a Debian, ad esclusione della *insert_module()*; quest'ultima ha un comportamento interessante anche in Mint, dato che i valori massimi sono registrati quando nel sistema non vi è LKRG. Per quanto riguarda le altre funzionalità, esse hanno un andamento stabile come si può osservare dai grafici: i valori ottenuti in assenza del modulo sono abbastanza omogenei tra loro con poche misurazioni isolate, e lo stesso vale per gli altri.

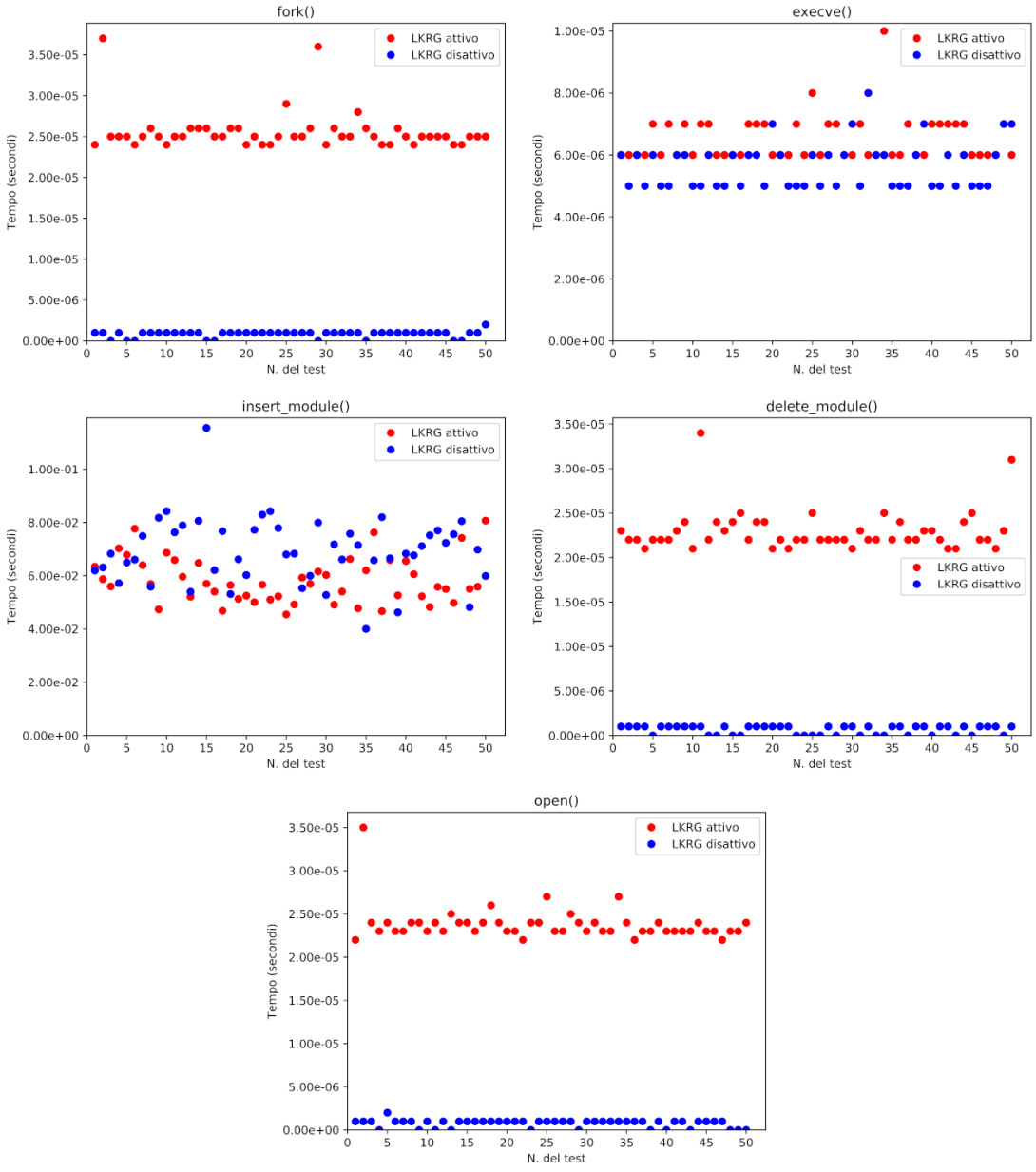

Figura 4.17: Benchmark restanti funzioni con ncycle=1 (Mint).

SystemCall	$\overline{x}$ loaded	$\overline{x}$ unloaded	σ loaded	σ unloaded
open()	2.380e-05	7.800e-07	1.908e-06	4.600e-07
fork()	2.556e-05	8.400e-07	2.434e-06	4.176e-07
execve()	6.500e-06	5.640e-06	7.280e-07	7.419e-07
insert_module()	5.817e-02	6.920e-02	8.481e-03	1.234e-02
delete_module()	2.292e-05	6.400e-07	2.288e-06	4.800e-07

Tabella 4.9: Dati benchmark restanti funzioni con ncycle=1 (Mint)

I dati nella Tabella 4.9 confermano quanto appena detto: il valore medio d'esecuzione della singola *insert_ module()* è effettivamente inferiore quando il modulo è caricato nel kernel, mentre tutti gli altri sono superiori in sua presenza. La funzione *execve()* mantiene circa lo stesso tempo per le due tipologie d'esecuzione, e le deviazioni standard differiscono per un fattore piccolissimo di $2e^{-8}$. I tempi relativi alle altre system call invece variano come nei test relativi agli altri sistemi, passando da valori di ordine 10^{-7} a ordine 10^{-5}, aumentando di conseguenza anche la deviazione standard. Infine nella *delete_ module()* sono stati registrati molti valori pari a 0 senza LKRG, mentre, in presenza del modulo, il tempo minimo registrato è $2x10^{-5}$, un valore comunque molto più elevato della media. Riguardo la *execve()* si può affermare che il tempo d'esecuzione non sia influenzato in maniera evidente, assumendo valori poco discostanti tra loro come nei precedenti test.

Concludiamo l'analisi presentando l'ultimo test di SysBench nei grafici in Figura 4.18 con i relativi dati nella Tabella 4.10, in cui ncycle assume valori esponenziali per dedurre se anche in questo sistema sono apportate alcune ottimizzazioni.

SystemCall	$\overline{x}$ loaded	$\overline{x}$ unloaded	σ loaded	σ unloaded
setuid()	2.797e-05	8.407e-07	8.672e-06	5.848e-07
setgid()	2.303e-05	5.871e-07	2.276e-06	2.066e-07
setresuid()	2.154e-05	6.193e-07	1.564e-06	1.922e-07
setresgid()	2.403e-05	6.368e-07	3.550e-06	1.966e-07
setreuid()	2.211e-05	6.051e-07	1.311e-06	2.040e-07
setregid()	2.108e-05	5.922e-07	9.071e-07	2.245e-07
setfsuid()	2.045e-05	3.754e-07	9.822e-07	1.889e-07
setfsgid()	2.128e-05	5.454e-07	1.969e-06	2.304e-07
open()	2.758e-06	2.278e-06	1.629e-06	1.382e-06
fork()	4.135e-04	3.156e-04	8.257e-05	7.042e-05
execve()	1.420e-02	1.468e-02	2.009e-02	2.152e-02
insert_module()	4.086e-03	4.353e-04	3.770e-04	5.066e-04
delete_module()	3.488e-03	2.788e-04	9.300e-05	3.561e-04

Tabella 4.10: Dati benchmark con ncycle=1, 10, 100, 1000, 10000 (Mint)

Il primo dettaglio da commentare è relativo alla funzione *setfsuid()* nel grafico con ncycle=1, il cui campionamento ha rilevato un tempo d'esecuzione pari

a 0 in assenza di LKRG, diversamente da quanto visto fino ad ora negli altri sistemi. Come è stato possibile osservare nel grafico in Figura 4.16 l'esecuzione di tale funzione è spesso nulla, per questo motivo nel grafico in Figura 4.18 il tempo è pari a 0. Generalmente si può affermare che le funzionalità con il rapporto $\frac{Tattivo}{Tdisattivo}$ massimo sono le *setX()* com'era risultato negli altri sistemi.

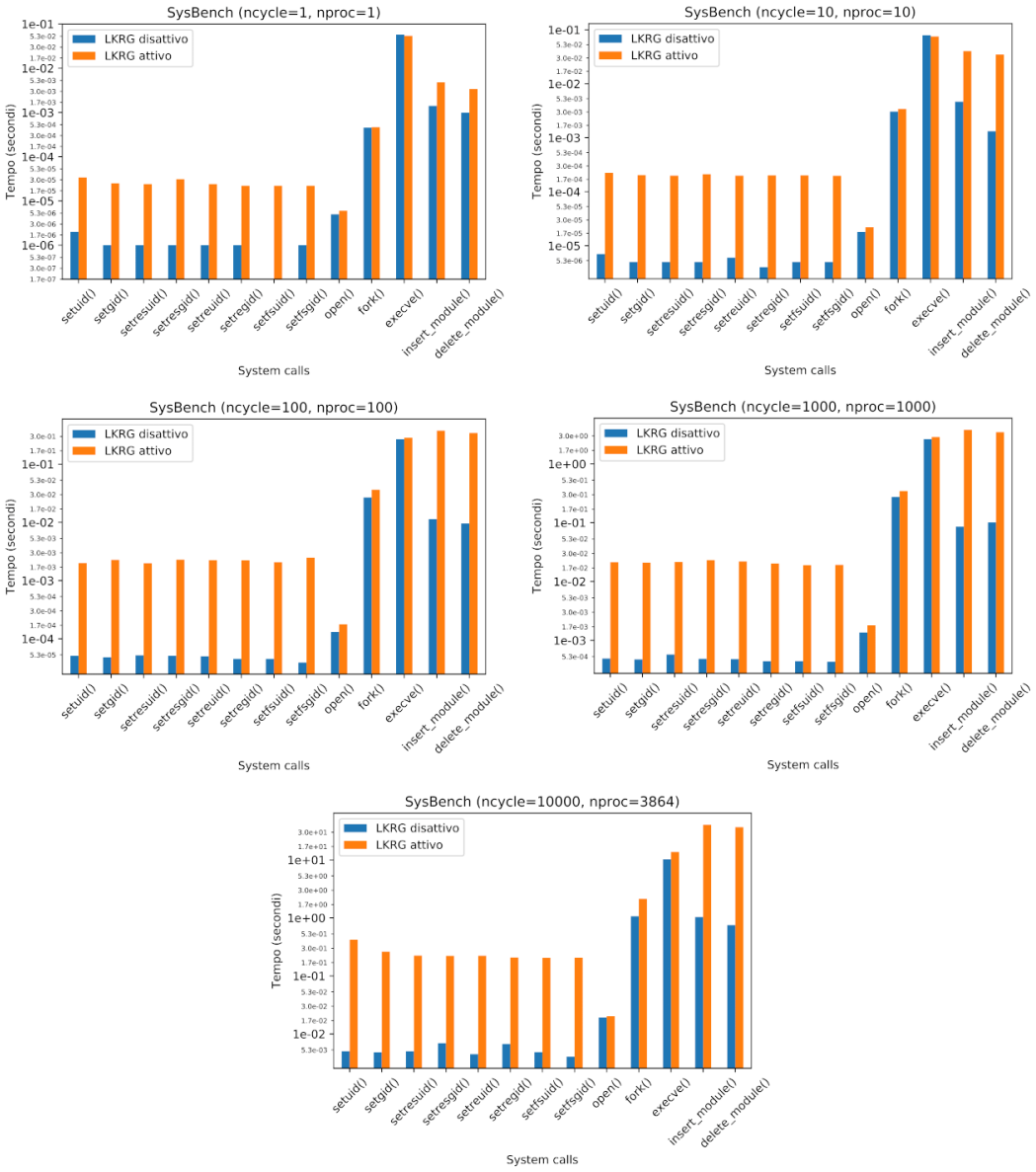

Figura 4.18: Benchmark tempo totale (Mint).

La *open()*, *execve()* e la *fork()* risultano ancora una volta essere le system call meno influenzate, i cui tempi d'esecuzione variano talmente poco che durante il test è quasi impercettibile la differenza (un paio di secondi in più per la execve quando il modulo è attivo e ncycle=10000). I tempi in relazione al loro ordine di grandezza ottenuti in seguito al test della *insert_module()* e *delete_module()* sono i più elevati, per cui la differenza lanciando un programma che esegue tali funzionalità diventa tanto più evidente quanto incrementa il numero di volte che vengono invocate: ad esempio considerando la prima delle due, la quale

presenta una differenza pari a $3x10^{-3}$ tra le due esecuzioni e lanciando il test in cui tale system call viene invocata 10000 volte, il tempo d'esecuzione, non considerando le ottimizzazioni, aumenterebbe di $3x10^{1}$ volte, ovvero 30 secondi.

Nella Tabella 4.10 sono riportati i dati di tutte le system call relativi a quest'ultimo test. Si osservi nuovamente che questi dati si discostano parecchio da quelli contenuti nella Tabella 4.8 e Tabella 4.9, come era già successo nei casi precedenti. Vi è persino una differenza di 10^2 tra il valore medio della singola esecuzione di *insert_module()* presentato in questa tabella e quello in Tabella 4.9. Perciò è corretto assumere che vi sia stato un intervento di ottimizzazione, come si può osservare nei grafici sottostanti.

Anche in questo sistema è risultato che il tempo medio della singola system call ad esclusione della *setfsuid()* diminuisce in base al numero di volte che viene richiamata. Nei grafici successivi è possibile osservare come il tempo d'esecuzione delle chiamate quando LKRG non è caricato risulta essere massimo nel test con ncycle=1, mentre raggiunge il minimo valore registrato nel test con ncycle=10000. In presenza del modulo, il tempo di tali funzionalità è molto variabile, talvolta esponenziale come nel grafico della *execve()* in Figura 4.21, mentre altre volte raggiunge dei picchi di massimo e minimo per poi variare nuovamente (si veda la *fork()* o la *setfsuid()*).

Si ricorda che per questi ultimi grafici è stata utilizzata una scala logaritmica per l'asse delle x, in modo tale da rappresentare i valori esponenziali assunti da ncycle.

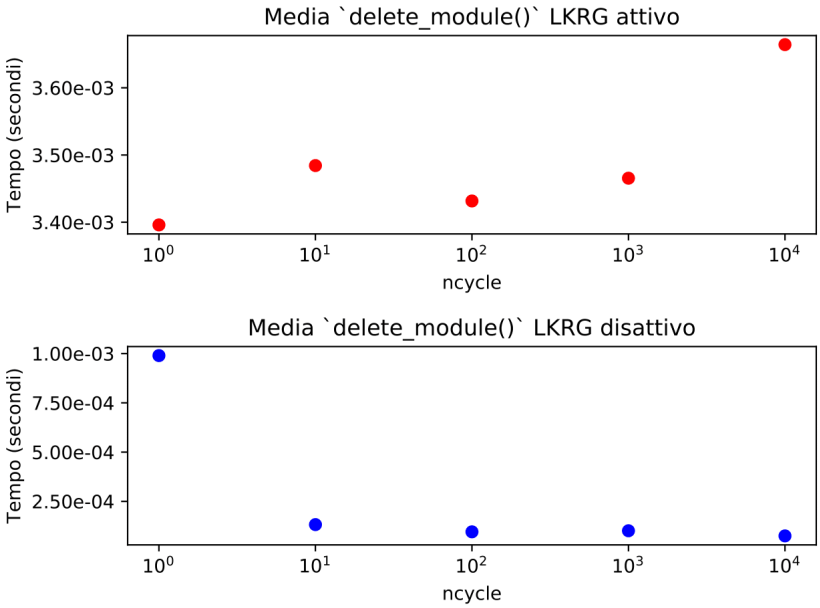

Figura 4.19: Media singole system call nei 5 benchmark parte 1 (Mint).

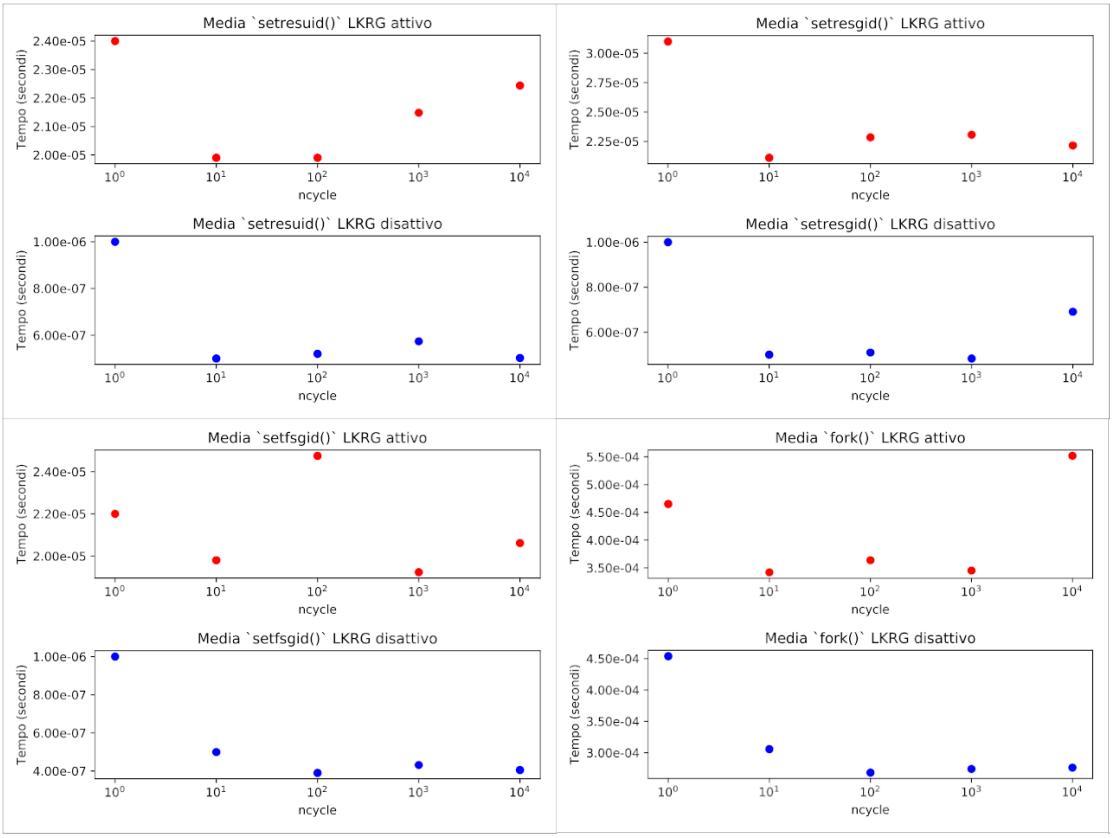

Figura 4.20: Media singole system call nei 5 benchmark parte 2 (Mint).

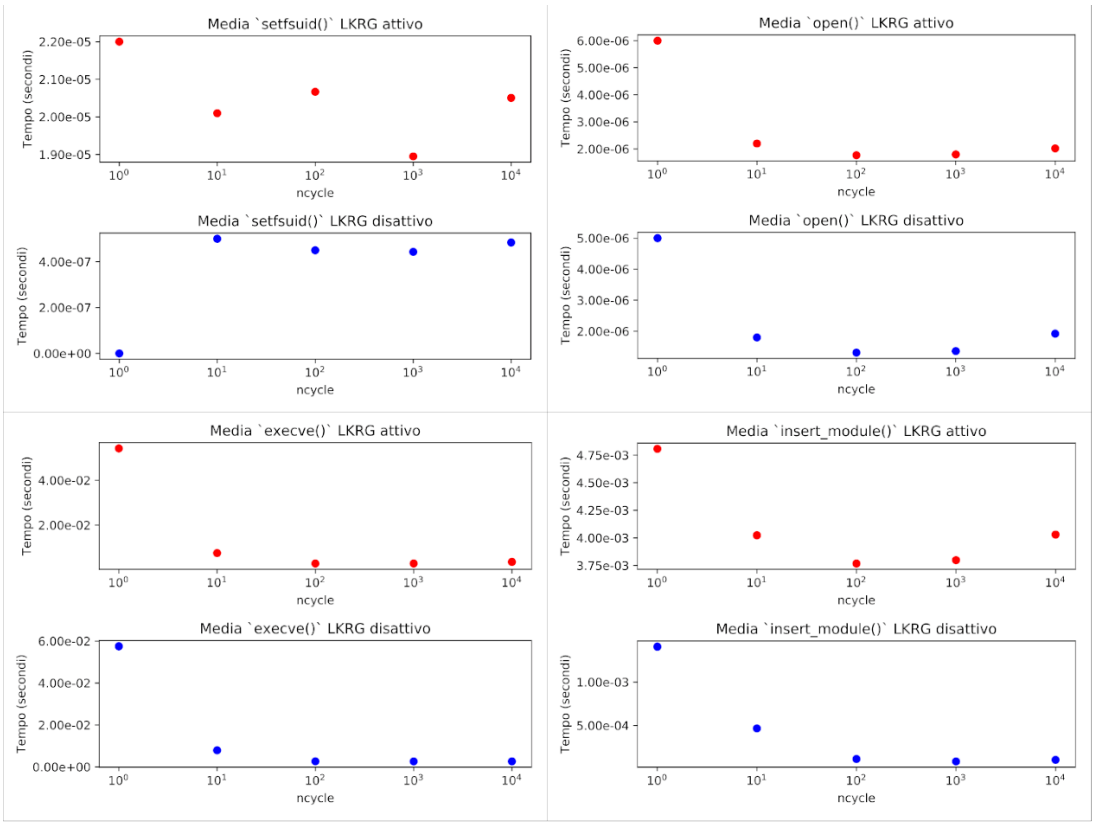

Figura 4.21: Media singole system call nei 5 benchmark parte 3 (Mint).

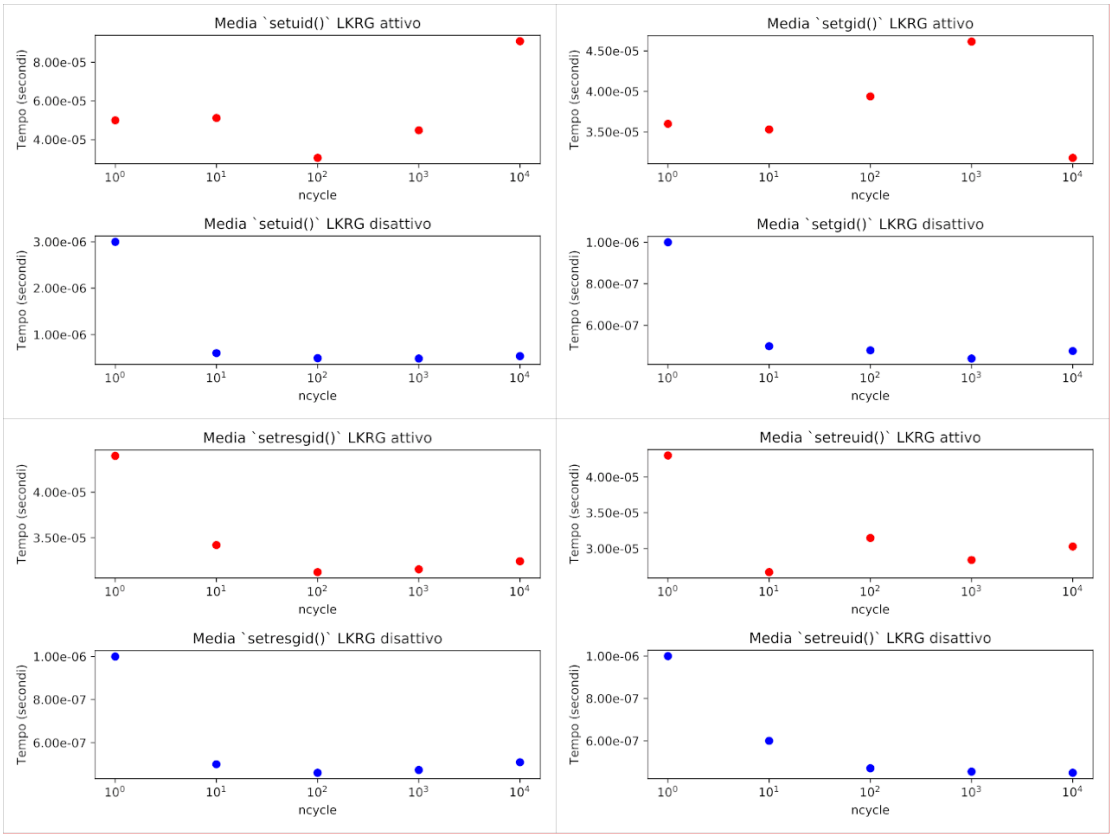

Figura 4.22: Media singole system call nei 5 benchmark parte 4 (Mint).

4.4 I tre sistemi a confronto

Nel corso della trattazione dei test nei vari sistemi sono già state anticipate alcune somiglianze e differenze dei risultati. In questa sezione conclusiva l'attenzione è rivolta all'aumento percentuale del tempo d'esecuzione delle system call in seguito al caricamento di *Linux Kernel Runtime Guardian*, commentando appropriatamente i valori ottenuti. Per ottenere in formule matematiche il tempo misurato con LKRG presente, bisogna attenersi alla legge:
$T_f = T_i + x\%T_i => T_f = T_i * (1 + x\%)$

Ad esempio se una chiamata presenta un aumento pari al 100%, il tempo finale è $T_f = T_i * (1 + 100\%) = T_i * 2 = 2T_i$, ovvero il doppio.

Dalla Tabella 4.11 si evince che le system call minormente influenzate dal modulo in tutti i sistemi sono la *setuid()*, *setregid()*, *execve()* e *insert_module()*, in quanto il tempo d'esecuzione a causa dei controlli d'integrità raddoppia o triplica al massimo. Al contrario, le rimanenti funzioni subiscono un aumento rilevante che varia dal 800% per la *setregid()* (corrispondente a 9 volte il tempo iniziale) ad un massimo di 4746 per la *setfsgid()*, ovvero poco più di $48T_i$.

Questi elevati valori porterebbero a pensare che LKRG rallenti pesantemente il sistema nel quale è installato, portando sfortunatamente alla conclusione sbagliata. Quando si osservano delle percentuali bisogna sempre fare riferimento ai tempi di partenza e al loro ordine di grandezza: ad esempio, un aumento del 100% in una chiamata che impiega 1 secondo ad essere soddisfatta è più

SystemCall	Δ Ubuntu (%)	Δ Debian (%)	Δ Mint (%)
setuid()	2332	2193	1875
setgid()	213	102	121
setresuid()	3659	735	2328
setresgid()	296	1597	365
setreuid()	3144	1478	2136
setregid()	283	884	327
setfsuid()	3406	1519	3347
setfsgid()	3824	1861	4746
open()	3089	1978	3051
fork()	2792	1906	3043
execve()	123	143	115
insert_module()	137	75	84
delete_module()	3432	1411	3581
Media	2056	1222	1932

Tabella 4.11: Aumento percentuale tempo d'esecuzione nei 3 sistemi

rilevante di un aumento pari a 5000% di una system call il cui tempo d'esecuzione è dell'ordine di 10^{-7}. Per questi motivi, dopo aver analizzato tutti i grafici ed i tempi riportati in questo capitolo, è possibile farsi un'idea circa l'effettivo overhead del sistema, riferendosi ad un caso d'uso personale.

Dai grafici e dai dati è risultato che sicuramente in tutti e tre i sistemi LKRG effettua i controlli con tempi differenti, nonostante l'installazione ed l'esecuzione di SysBench fosse avvenuta nelle medesime condizioni. L'ultima riga della tabella riporta l'incremento percentuale medio delle system call in ogni sistema; si osservi come in base a questi valori il sistema al quale LKRG aggiunge maggiore overhead risulti essere Ubuntu, seguito da Mint ed infine Debian.

Dovendo prestare attenzione ad un possibile scenario reale, si pensi ad un server hostato in un sistema Ubuntu, dal quale in base ad alcune richieste dell'utente vengono lanciati i comandi *execve()*, *fork()* e *open()*. Essendo i tre maggiormente utilizzati si vuole valutare se vale la pena o meno inserire LKRG nel server, per evitare che il tempo di risposta del server renda l'esperienza utente meno gradevole. Dall'analisi è risultato che l'aumento in percentuale delle funzioni è rispettivamente pari a 123%, 2792% e 3089%. Per cui assumento come tempi per ogni singola chiamata i valori riportati nella tabella Tabella 4.3, ovvero $7.74x10^{-6}, 3.518x10^{-5}$ e $3.398x10^{-5}$ ed ipotizzando che per ogni utente vengono gestite singolarmente mille istanze delle seguenti operazioni (ncycle=1 ntimes=1000, esecuzione di SysBench tramite lo script), si ha che:

$TempoFinale_{execve} = 7.74x10^{-6} * (1 + 1.23) * 1000 = 0.01726$ secondi
$TempoFinale_{fork} = 3.518x10^{-5} * (1 + 27.92) * 1000 = 1.01741$ secondi
$TempoFinale_{open} = 3.398x10^{-5} * (1 + 30.89) * 1000 = 1.0836222$ secondi

Nonostante questo scenario sia estremo, in quanto non accadrà mai di gestire 1000 di queste system call per un singolo utente, si ha che la richiesta verrà

soddisfatta nei tempi riportati, i quali sono estremamente bassi. Si pensi infatti se l'esperienza utente è veramente influenzata da un ritardo di 1 secondo in una risposta; sono tempi molto piccoli, difficile da percepire se non effettuandone una valutazione con un programma come SysBench.

Conclusioni

In un mondo in cui proteggere le informazioni è diventato oramai un obiettivo di vitale importanza è necessario essere consapevoli dell'esistenza di software in grado di aumentare l'integrità e la sicurezza del proprio sistema. Bisogna però tener conto del trade-off sicurezza-tempo, due importanti fattori talvolta antagonisti, in quanto più si aggiungono layer protettivi al sistema, più aumenta il tempo di risposta. È necessario valutare attentamente le proprie necessità prima di installare qualsiasi materiale, decidendo quali sono e come si possono raggiungere gli obiettivi prestabiliti, valutando se si preferisce garantire un'ottima prontezza di risposta, o la sicurezza (e non è detto che non vi sia un modo per oltrepassare queste ulteriori difese), oppure se raggiungere una soluzione intermedia ricoprendo entrambe le qualità.

Pertanto, in seguito all'analisi sviluppata in questo elaborato, è vivamente consigliato l'uso del modulo *Linux Kernel Runtime Guardian*, il quale non solo è sorprendentemente performante, ma garantisce anche l'integrità dei dati e rileva efficacemente molte minacce al sistema che comprometterebbero la sua sicurezza. Nelle macchine testate, nonostante fossero ambienti virtuali e non fisici, si sono ottenuti ottimi risultati che portano a valutarne positivamente l'uso non solo nella propria installazione locale, ma anche nei vari server utilizzati con obiettivi differenti. Infatti, si è valutato l'utilizzo di LKRG anche in calcolatori con installazioni di Linux meno recenti (ad esempio negli ATM), le quali per problemi di supporto non vengono aggiornate, non curandosi in questo modo di possibili nuovi vettori d'attacco che si sono sviluppati negli ultimi anni. Con un trascurabile calo delle performance, il modulo potrebbe offrire a questi sistemi un maggior livello di sicurezza, favorendone il continuo utilizzo.

Infine, un ulteriore aspetto ritenuto personalmente importante è l'utilizzo di software libero: rispetto ad uno definito "proprietario", la cui licenza non ne permette la modifica, lo studio, la ridistribuzione e la condivisione tenendo segreto il sorgente (l'utente è limitato al semplice utilizzo), un software pubblicato sotto i termini di una licenza di software libero concede lo studio, la modifica e la ridistribuzione del progetto stesso. In questo modo non solo l'utente è a conoscenza della struttura del software potendosi leggere il sorgente, ma può anche apportare dei miglioramenti e condividerli con il resto della community. È grazie a questa filosofia che il mondo Linux è attraente ed efficace, in quanto offre la possibilità di costruire gratuitamente e liberamente il proprio sistema senza avere vincoli nei confronti di nessuno.

In conclusione, LKRG offre un servizio a dir poco eccezionale, facendo sì che anche l'utente meno esperto e senza disponibilità economiche possa difendersi da certi tipi di minacce in maniera semplice ed efficace subendo una diminuzione accettabile delle performance.

Appendice A

Per compilare il progetto SysBench e *Linux Kernel Runtime Guardian* sono necessari vari strumenti ottenibili mediante il proprio gestore di pacchetti. Le distribuzioni presentare sfruttano lo stesso comando per scaricare ed installare nuovi pacchetti nel proprio sistema, ovvero *apt*.

Una volta scaricata ed installata a piacere la propria distribuzione, aprire il terminale e tramite il comando *sudo apt install* (nel nostro caso) installare:

- linux-headers-$(uname -r), necessario al fine della compilazione del kernel e la build dei moduli ($(uname -r) viene sostituito dalla versione attuale del proprio kernel, al fine di ottenere i giusti headers);
- build-essential, comprende il compilatore gcc e make, necessari per la compilazione dei software;
- libelf-dev, per la lettura e scrittura dei file ELF (eseguibili) ad alto livello.

A questo punto il sistema è configurato per la compilazione corretta dei due software.

A.1 Installazione del *Linux Kernel Runtime Guardian*

Per installare LKRG nel proprio sistema è necessario seguire le seguenti istruzioni:

1.Scaricare LKRG al seguente link ufficiale `https://www.openwall.com/lkrg/`.
2.Accedere tramite terminale alla directory in cui si trova: *cd path_to_dir*.
3.Estrarre il contenuto del file .zip: *tar -xzvf lkrg-0.4.tar.zip*.
4.Entrare nella cartella creata: *cd lkrg-0.4*.
5.Compilare il progetto: *make -j8*
6.Entrare nella cartella di output: *cd output*
7.Caricare la versione .ko ottenuta in seguito alla compilazione del modulo: *sudo insmod output/p_lkrg.ko p_init_log_level=3*, dove *p_init_log_level* è il parametro in ingresso tramite il quale si decide il livello di logging delle informazioni in console (e non nel terminale, bensì nell'esecuzione del proprio sistema in modalità console e non desktop, accedendovi premendo CTRL+ALT+F7).
8.Controllare che LKRG sia stato effettivamente caricato: *lsmod | grep p_lkrg*.

Per rimuoverlo dal kernel è sufficiente utilizzare *sudo rmmod p_lkrg* e controllare che sia stato rimosso.

In caso vi fossero problemi con la compilazione del modulo, contattare l'autore tramite la mailing list presente nel sito indicato precedentemente.

A.2 Installazione di *SysBench*

Il procedimento per installare SysBench è il seguente:

1.Accedere al mio repository online tramite browser e copiare l'indirizzo fornito cliccando il tasto *clone*.
2.Clonare il repository nel proprio computer digitando da terminale: *git clone https://SimoMagno@bitbucket.org/SimoMagno/sysbench.git*.
3.Accedere alla cartella clonata: *cd sysbench*.
4.Compilare il progetto: *make*.

A questo punto il progetto è eseguibile e si può lanciare in due maniere presentate:

- singola esecuzione : *[sudo] ./sysbench ncycle filename*;
- multipla esecuzione: *sudo ./script.sh ntimes ncycle path/to/p_lkrg.ko*

Il primo scenario è il normale caso di esecuzione di programma da riga di comando, in cui viene richiamato passando i parametri *ncycle* (intero) e *filename* (stringa); viene eseguito un singolo benchmark in cui ogni system call viene chiamata ncycle-volte ed il risultato è salvato nel file indicato. Da notare che in questo il programma può essere eseguito sia con i privilegi sia senza, in quanto la parte di caricamento/rimozione del modulo LKRG nel kernel è stata volutamente lasciata a carico dell'utente.

Nel secondo caso, il programma viene lanciato ntimes-volte producento altrettanto file di output, ognuna delle quali effettua il test delle system call invocate ncycle-volte. La differenza sostanziale consiste nel tempo d'esecuzione medio e totale delle chiamate a funzione: si è osservato infatti che per valutare il tempo medio d'esecuzione di una system call è più preciso effettuare ntimes-volte il benchmark con parametro ncycle=1, in quanto se la stessa funzione è richiamata più volte all'interno dello stesso programma possono esserci dei salvataggi in cache e miglioramenti apportati dalla glibc, dalla cache o dal processore, i quali alterano i risultati come mostrato nel Capitolo 4. Per questa tipologia d'esecuzione sono necessari i privilegi di root, in quanto il caricamento e rimozione di LKRG è a carico dello script ogni volta che SysBench viene eseguito.

Per qualsiasi tipo di informazione o chiarimento sentitevi liberi di contattarmi.

Ringraziamenti

Grazie a tutti coloro che mi hanno costantemente supportato lungo questo ricco percorso di studio. In particolare il pensiero è rivolto:

ai miei genitori Fabrizio e Rosanna, per avere sostenuto economicamente quest'esperienza;

a mio fratello Francesco, per avermi sopportato in casa e per la sua ispiratrice determinazione e perseveranza nel raggiungere gli obiettivi;

alla mia ragazza Catarina, perchè è stata e continua ad essere una compagna di viaggio impeccabile, disponibile e motivatrice;

ai miei cari amici, per il loro sostegno, la loro presenza e i loro scherzi i quali hanno reso le giornate di studio meno pesanti e il tempo libero ancora più prezioso;

al team di CeSeNA Security, per avermi fatto scoprire il vastissimo mondo della sicurezza informatica ed avermi guidato in questa realtà a me fino ad ora sconosciuta;

al mio relatore prof. Gabriele D'Angelo, per le sue lezioni essenziali e appassionanti, le quali hanno sicuramente influenzato la mia scelta di fare della sicurezza informatica il mio interesse di studio.

Ho incluso brevemente tutte le persone rilevanti con le quali spero di rimanere sempre in contatto, perchè ognuno a modo suo ha avuto (e spero continui ad avere) un ruolo determinante nella mia crescita.

Bibliografia

Corbet, Jonathan, Alessandro Rubini e Greg Kroah-Hartman (2005). *Linux Device Driver 3rd Edition*. O'Reilly Media Inc. 1005 Gravenstein Highway North Sebastopol CA 95472. URL: https://lwn.net/Kernel/LDD3/.

Salzman, Peter Jay, Michael Burian e Ori Pomerantz (2007). *Linux Kernel Module Programming Guide*. Free Book. URL: http://www.tldp.org/LDP/lkmpg/2.6/html/.

Love, Robert (2007). *Linux System Programming*. O'Reilly Media Inc. 1005 Gravenstein Highway North Sebastopol CA 95472. URL: http://igm.univ-mlv.fr/~yahya/progsys/linux.pdf.

Openwall (n.d.). *Openwall Community Wiki*. URL: https://openwall.info/wiki/p_lkrg/Main.

Events, Proidea (n.d.). *CONFidence 2018: Linux Kernel Runtime Guard (LKRG) under the hood*. URL: https://www.youtube.com/watch?v=tOiPM692DOM.

Torvald, Linus (n.d.). *Linus Torvald - GitHub*. URL: https://github.com/torvalds/linux/.

Academy, U.S. Naval (n.d.). *System Programming*. URL: https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/11/lec.html.

Wienand, Ian (n.d.). *PLT and GOT - the key to code sharing and dynamic libraries*. URL: https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html.