Università di Pisa

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

Visibilità nei container Linux tramite eBPF e riconoscimento in tempo reale di anomalie.

Relatore:
Luca Deri

Candidato:
Andrea Bonanno

Anno Accademico 2019/2020

Indice

1 Introduzione **1**

1.1 Motivazione e obiettivo . . . 2

2 Stato dell'arte **4**

2.1 Fonti degli eventi . . . 4

2.1.1 Kprobes . . . 5

2.1.2 Uprobes . . . 9

2.1.3 Tracepoints . . . 10

2.1.4 USDT probes . . . 10

2.2 Meccanismi di raccolta eventi . . . 11

2.2.1 Loadable Kernel Module (LKM) . . . 11

2.2.2 perf_events . . . 12

2.2.3 ftrace . . . 13

2.2.4 eBPF (extended Berkeley Packet Filter) . . . 14

2.3 Front-End . . . 17

2.3.1 ftrace . . . 17

2.3.2 perf . . . 17

2.3.3 bcc . . . 18

2.4 Container Security . . . 18

2.4.1 Policy based . . . 18

2.4.2 Anomaly based . . . 21

2.5 Soluzione Proposta . . . 25

3 Progetto **28**
3.1 Struttura 28
3.2 Prerequisiti 29
3.3 Container in Linux 29
3.3.1 Namespaces 30
3.3.2 Docker 31
3.4 Algoritmo BoSC 32
3.4.1 Definizioni 32
3.4.2 Funzionamento 33
3.5 Il Filter 35
3.5.1 Funzionalità 35
3.5.2 Comunicazione con il Classifier e Maps 35
3.5.3 Identificare Container e Processi 36
3.5.4 Handlers 38
3.5.5 Formato degli eventi 40
3.6 Il Classifier 41
3.6.1 Funzionalità 41
3.6.2 Comunicazione con il Filter 41
3.6.3 Implementazione dell'algoritmo 42

4 Validazione e Risultati **44**
4.1 Test di unità 44
4.1.1 Modulo eBPF + Classifier 45
4.1.2 Modulo BagManager 45
4.2 Test di Sistema 46
4.2.1 Requisiti 46
4.2.2 Usability Test 47
4.2.3 Facility Test 47
4.2.4 Performance Test 55
4.2.5 Storage use Test 56

4.2.6 Security Test 56
4.2.7 Load Test 57

5 Conclusioni **58**
5.1 Lavori futuri 59

A Codice Sorgente **61**

Bibliografia **69**

Sommario

In questa tesi viene analizzato lo stato attuale delle tecnologie e degli approcci per il monitoring dei container in ambiente Linux, valutando quelle che sono le soluzioni più versatili e a basso impatto sul sistema. Lo scopo è esporre le capacità in fatto di versatilità e perfomance dei sistemi di visibilità già esistenti, e di valutare le differenze nei loro tratti salienti. Viene inoltre proposta una soluzione per la rilevazione in tempo reale delle anomalie nei container. Lo strumento realizzato fa uso del tracing framweork eBPF e di una variante di algoritmi già noti basati sull'analisi delle frequenze delle syscall. Il risultato è uno strumento a basso overhead che non richiede conoscenza a priori del container da monitorare. Questa tesi dimostra come si possano ottenere risultati positivi utilizzando eventi semplici come fonti di dati, quali le invocazioni del le syscall, senza necessariamente analizzare le relazioni tra di esse e i loro argomenti.

Capitolo 1

Introduzione

Nel corso dell'ultimo decennio la tendenza all'aumento della complessità nei sistemi informatici ha portato con se nuove sfide nello studio del comportamento del software. Ottenere e migliorare la visibilità del software in sistemi complessi è diventato un importante obiettivo per realizzare sistemi su larga scala. Inoltre, le tecnologie di visibilità devono misurarsi con la necessità di mantenere basso l'*overhead* che il loro uso inevitabilmente aggiunge.

Un approccio classico per ottenere visibilità del software è l'analisi dei cosiddetti *logs*. I log sono un'ottima fonte di informazione precisa sul comportamento del software, e il loro utilizzo difficilmente verrà completamente rimpiazzato. I log però limitano le informazioni disponibili all'utente a quelle che lo sviluppatore dell'applicazione ha deciso di esporre.

Un altro approccio comune è l'uso di metriche relative all'utilizzo di risorse come aiuto nello studio del comportamento del software. Mentre i log espongono informazioni esplicite, le metriche aggregano informazione sul comportamento di un'applicazione in un dato intervallo di tempo.

Il concetto di *observability* affronta il problema della visibilità da una diversa prospettiva. Una possibile definizione di observability è la capacità di effettuare richieste arbitrarie di informazioni ad un sistema e di ricevere risposte esaustive. La maniera più naturale per ottenere l'observability consiste nel raccogliere tutti i dati che un sistema può generare e aggregarli solo quando è necessario formulare una risposta.

I container Linux sono un'astrazione realizzata tramite un'insieme di capacità del Kernel Linux con lo scopo di isolare e gestire processi. Nell'ultimo decennio l'evolversi dei container

Linux ha influenzato l'approccio alla progettazione di sistemi, e l'uso di container in ambienti di produzione è in forte crescita. Tale aumento di popolarità dei container e l'uso delle sottostanti capacità del Kernel Linux ha esposto un nuovo livello di complessità sul quale è possibile applicare observability e con cui è necessario confrontarsi, nonchè nuove sfide in termini di mantenimento della sicurezza.

1.1 Motivazione e obiettivo

La tecnologia dei containers (o più propriamente *OS-level virtualization)* è attualmente una soluzione in rapida crescita per la distruibuzione ed esecuzione di applicazioni in ambienti di produzione[38]. La popolarità dei containers è collegata innanzi tutto a ragioni di performance rispetto ad un approccio basato esclusivamente su macchine virtuali. In generale, i containers offrono una maniera conveniente per consolidare librerie di sistema, file e codice sorgente necessari per il supporto a tempo di esecuzione di un applicativo, permettendo un rapido deployment. Uno dei fattori di crescita della popolarità dei container è la transizione degli ambienti produttivi verso architetture basate su microservizi[74].

Il framework per container Docker continua a dominare lo scenario della virtualizzazione tramite container. Un recente sondaggio afferma che più del 79% delle compagnie utilizzano Docker come principale piattaforma per container. Secondo la DockerCon 17 [54], più di 900,000 applicazioni sono state sviluppate dalla community Docker.

La tecnologia dei container porta con se nuove sfide in termini di sicurezza. Durante il Blackhat 2017 sono stati dati esempi di potenziali usi malevoli tramite l'API Docker in termini di esecuzione di codice arbitrario, host rebinding e attacchi shadow container[32]. Tali attacchi possono causare danni ingenti alle risorse di una compagnia. Nel Settembre 2017, il *National Institute of Standards and Technology* (NITS) ha rilasciato la *Special Publication* (SP)-800-190[59], una guida alla sicurezza dei container che illustra diverse problematiche e soluzioni possibili. La (SP)-800-190 divide i rischi alla sicurezza dei container in cinque categorie: *image risks*, *registry risks*, *orchestrator risks*, *container risks* e *host OS risks*. Proposte di strumenti per il rilevamento delle anomalie in ambiente container come KubAnomaly[67] mettono in rilievo come strumenti e approcci classici alla sicurezza non sono bastevoli per fronteggiare

i rischi relativi a container e orchestrator. Occorre invece sviluppare strumenti specializzati nell'ottenere visibilità a livello di containers e rilevare possibili anomalie.

Molti strumenti attuali per la sicurezza dei containers detti *policy based* necessitano di configurazioni esplicite e della conoscenza a priori del comportamento e delle risorse utilizzate da un container. Altri strumenti invece riescono a modellare il comportamento tipico di un container tramite processi di apprendimento automatico per poi segnalare eventuali comportamenti anomali. Nella maggior parte dei casi però la capacità di apprendimento automatico va a scapito di un carico computazionale oneroso.

L'obiettivo di questa tesi è mostrare come sia possibile realizzare sistemi relativamente semplici e a basso overhead che possono ottenere l'osservabilità a livello di container, modellare il comportamento normale e rilevare le anomalie, senza nessuna conoscenza a priori del container. Un altro punto di interesse è mostrare come il Kernel Linux sia attualmente dotato di funzionalità mature e versatili ai fini dell'osservabilità di container.

Capitolo 2

Stato dell'arte

Un sistema di visibilità in ambiente Linux può essere generalmente diviso in tre parti. In primo luogo gli *event sources*, cioè meccanismi nel kernel-space preesistenti che possono essere sfruttati come fonte degli eventi. In secondo luogo un *tracing framework* eseguito in kernel-space, che ha il compito di raccogliere e contare gli eventi, e nel caso di un framework programmabile anche di effettuare in maniera efficiente aggregazione e creazione di statistiche. Infine un *front-end* nello user-space che fornisce all'utente la possibilità di interfacciarsi con il tracing framework, creare statistiche e riepilogo dei dati e visualizzarli all'utente.

Una volta ottenuta la visibilità desiderata, se lo scopo è l'identificazione di comportamenti anomali, i dati vengono processati da un sistema di rilevamento di anomalie, il quale processa i dati e classifica gli episodi di comportamento anomalo da quelli normali. Lo scopo di questo capitolo è offrire una panoramica sulle principali alternative che costituiscono lo stato dell'arte attuale in ambiente Linux. Verranno inoltre motivate quelle che sono state le scelte effettuate ai fini della realizzazione dello strumento.

2.1 Fonti degli eventi

Come primo passo nell'implementazione del progetto, è stata necessaria l'analisi e valutazione di diverse fonti di raccolta eventi. Il Kernel Linux offre un ecosistema di fonti dalle quali attingere che si è ampliato nel corso del tempo.

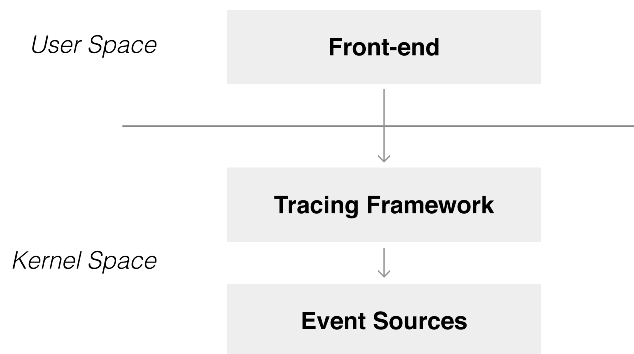

Figura 2.1: Componenti di un Tracing System.

2.1.1 Kprobes

I Kprobe forniscono un meccanismo di *debugging* per il Kernel Linux che può essere utilizzato anche per tenere traccia degli eventi all'interno di un sistema in esecuzione in maniera dinamica e non distruttiva. I Kprobe sono stati sviluppati da IBM come meccanismo sottostante un altro strumento di tracciamento, i Dprobe. I Dprobe aggiungono una serie di funzionalità ai Kprobe, compreso un proprio linguaggio di scripting per la definizione di probe handlers, ma solamente i Kprobe sono stati inclusi nel Kernel standard.

Il funzionamento dei Kprobe dipende da caratteristiche e funzionalità specifiche dell'architettura del processore e utilizza meccanismi diversi in base all'architettura sottostante. Nella descrizione dei Kprobe si farà riferimento all'architettura x86. I Kprobe sono disponibili per le seguenti architetture: ppc64, x86_64, sparc64, ia64 e i386.

Un kernel probe è un insieme di handlers collegati ad certo indirizzo istruzione. Esistono tre tipi di kernel probe attualmente, i Kprobe, i Jprobe e i Kretprobe. Un Kprobe è definito da un *pre-handler* e da un *post-handler* e può essere installato su una qualunque indirizzo delle istruzione del Kernel (salvo alcune eccezioni). Quando un Kprobe è installato su una specifica istruzione e quell'istruzione viene eseguita, il pre-handler è eseguito subito prima dell'istruzione alla quale è collegato. Analogamente, il post-handler è eseguito immediatamente dopo l'istruzione alla quale è collegato. I Jprobe sono usati per accedere agli argomenti di una funzione del Kernel a tempo di esecuzione. I Jprobe sono definiti da un Jprobe handler

con lo stesso prototipo della funzione ai cui argomenti si desidera accedere. Quando la relativa funzione viene eseguita il controllo del flusso d'esecuzione viene trasferito al Jprobe handler definito dall'utente, al fine del quale il controllo ritorna alla funzione originale. I Kretprobe si comportano come i Jprobe, ma l'handler è eseguito all'uscita della funzione sulla quale sono installati.

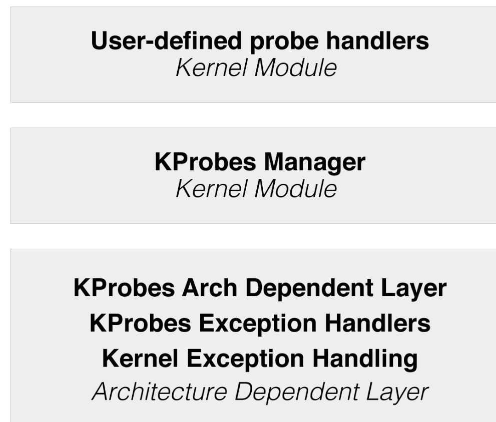

Figura 2.2: Architettura Kprobe.

In Figura 2.2 è illustrata la struttura dell'architettura Kprobe. Nel caso dell'architettura x86, i Kprobe sfruttano i meccanismi di trattamento delle eccezioni modificando i gestori standard delle eccezioni, e ciò crea un livello di implementazione che è dipendente dall'architettura sulla quale sono eseguiti [42]. Esiste anche un livello indipendente dall'architettura sottostante costituito dal Kprobe Manager, che implementa le funzioni di registrazione e de-registrazione dei probe. L'utente fornisce un probe handler all'interno di un modulo del Kernel, il quale registra il probe tramite l'uso del Kprobe Manager. La API relativa ai Kprobes è fornita da *linux/kprobes.h*, il Kprobe Manager (indipendente dall'architettura) è implementato in *kernel/kprobes.c* e la parte dipendente dall'architettura in *arch/*/kernel/kernel.c*.

Funzionamento di un Kprobe

Quando un Kprobe viene registrato, esso salva una copia dell'istruzione sulla quale viene installato e modifica i primi byte(s) dell'istruzione originale inserendo un comando di *breakpoint*, ad esempio *int3* su architettura x86_64 e i368. Non appena il flusso d'esecuzione raggiunge l'in-

dirizzo l'istruzione di breakpoint l'istruzione *int3* è eseguita (e registri della CPU salvati) causando l'esecuzione del gestore relativo alle eccezioni di tipo breakpoint in *arch/i386/kernel/traps.c*. Il gestore notifica il Kprobe tramite il meccanismo *notifier_call_chain*. A questo punto il controllo dell'esecuzione passa al Kprobe e alla funzione *kprobe_handler* che esegue il pre-handler definito dall'utente, passando all'handler l'indirizzo della *kprobe struct* contente informazioni sul Kprobe e i registri della CPU che sono stati salvati. Successivamente il Kprobe esegue la propria copia dell'istruzione sulla quale era stato installato (*single-stepping)*. Sarebbe più semplice eseguire l'istruzione *in place*, ma così facendo il Kprobe dovrebbe rimuovere il codice di breakpoint, creando una finestra temporale nella quale il flusso d'esecuzione relativo ad un'altra CPU potrebbe saltare il breakpoint. Dopo l'esecuzione dell'istruzione copiata, il Kprobe esegue il proprio post-handler (se è stato definito) tramite *post_kprobe_handler*. L'esecuzione poi riprende normalmente dall'istruzione successiva a quella su cui è stato installato il probe. Lo schema in Figura 2.3 riassume le fasi appena discusse.

Figura 2.3: Esecuzione di un Kprobe.

Funzionamento di un Jprobe

Un Jprobe è implementato tramite un particolare Kprobe installato sull'*entry point* di una funzione del Kernel. Esso utilizza un semplice principio di *mirroring* per accedere agli argomento della funzione a runtime, e al termine dell'esecuzione dell'handler registrato restituisce il con-

trollo alla funzione originale senza alterare lo stato dei registri e dello stack. L'handler relativo al Jprobe deve avere la stessa *signature* (argomenti e tipo restituito) della funzione sulla quale si desidera installare il Jprobe, e deve sempre terminare con la funzione *jprobe_return*. Anziché invocare un pre-handler definito dall'utente, un Jprobe definisce il proprio pre-handler chiamato *setjmp_pre_handler* e fa uso di un altro handler chiamato *break_handler*. L'esecuzione di un JProbe può essere divisa in tre fasi.

Nella prima fase l'esecuzione raggiunge il breakpoint e la funzione relativa ai Kprobe *kprobe_handler* esegue il pre-handler, cioè *setjmp_pre_handler* nel caso del Jprobe. *setjmp_pre_handler* copia il contenuto dello stack e dei registri e poi modifica l'instruction pointer corrente con un puntatore all'handler registrato dall'utente, dopo di che restituisce alla funzione chiamante *kprobe_handler* il valore 1, cioè istruisce di terminare anziché proseguire con il single-stepping come nel caso di un normale Kprobe. Al ritorno di *kprobe_handler* viene eseguito l'handler registrato dall'utente azichè la funzione del Kernel originaria, ed esso ha accesso completo allo stato (stack e registri). L'handler invocherà *jprobe_return* come sua ultima istruzione.

Nella seconda fase *jprobe_return* genera un breakpoint che trasferisce il controllo a *kprobe_handler* tramite *do_int3*. *kprobe_handler* verifica che l'indirizzo di breakpoint generato (l'indirizzo dell'istruzione *int3* in *jprobe_handler* non ha un probe registrato. In questo caso assume che il breakpoint sia stato generato dal Jprobe ed invoca il *break_handler* relativo. Il *break_handler* ripristina il contenuto di stack e registri con quelli salvati in prima di trasferire il controllo all'handler dell'utente.

Nella terza fase *kprobe_handler* esegue la preparazione per il single-stepping dell'entry point della funzione sulla quale era stato installato il Jprobe e il resto della sequenza è identico a quella relativa ad un Kprobe. Lo schema in Figura 2.4 riassume le fasi appena discusse.

Funzionamento di un Kretprobe

Un Kretprobe è implementato sfruttando i meccanismi dei Kprobe. Uno speciale Kprobe viene registrato sull'entry point della funzione sulla quale l'utente vuole installare il Kretprobe. Quando la funzione viene eseguita il Kprobe salva una copia dell'indirizzo di ritorno della funzione e sostituisce a quello corrente un indirizzo detto *trampoline*. L'indirizzo *trampoline* punta ad una sezione arbitraria di codice, tipicamente semplicemente ad una istruzione *nop*.

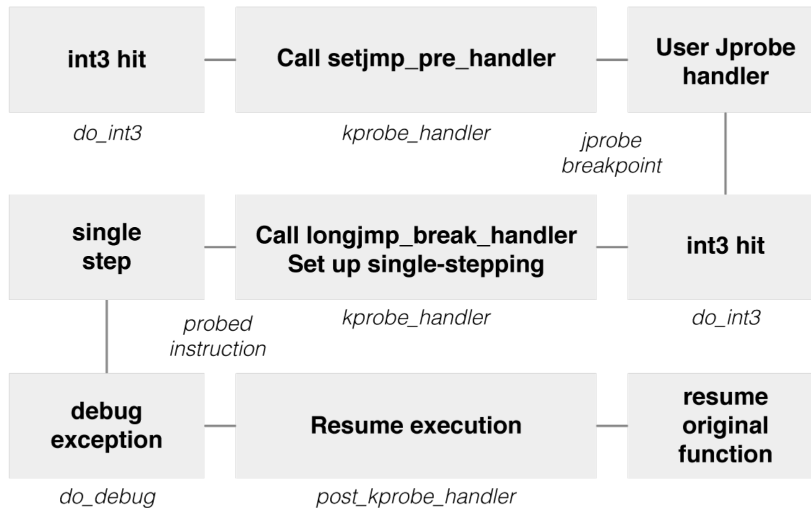

Figura 2.4: Esecuzione di un Jprobe.

All'avvio del sistema il modulo Kprobe mantiene sempre un Kprobe installato sull'istruzione *trampoline*.

Quando la funzione sulla quale è installato il Kretprobe viene eseguita, il controllo passa al Kprobe speciale che installa il *trampoline* e termina la propria esecuzione. Al termine della funzione originale il flusso di esecuzione raggiunge il *trampoline* e viene eseguito il Kprobe relativo. L'handler per questo Kprobe esegue a sua volta l'handler che l'utente aveva associato per il Kretprobe, dopodichè ripristina l'indirizzo di ritorno che era stato modificato con il *trampoiline*.

2.1.2 Uprobes

Gli Uprobe forniscono un meccanismo di tracciamento dinamico non-distruttivo analogo a quello dei Kprobe, ma relegato alle funzioni delle applicazioni nello user-space. Nello specifico i meccanismi forniti sono gli Uprobe e gli Uretprobe, che una volta installati su una funzione permettono l'esecuzione di handlers definiti dall'utente in kernel-space. La API è fornita da *linux/uprobes.h* e gli Uprobe/Uretprobe devono essere installati utilizzando un file eseguibile

ed un offset al suo interno che corrisponde all'indirizzo dell'istruzione da tracciare.

Siccome i probe sono associati ad un file eseguibile, essi hanno effetto su tutti i processi che eseguono codice relativo a quel file. Quando un probe viene registrato, viene salvata una copia dell'istruzione tracciata, ed i primi byte(s) dell'istruzione originale vengono modificati inserendo un'istruzione di breakpoint. Quando il flusso d'esecuzione raggiunge il breakpoint un meccanismo analogo a quello dei Kprobe trasferisce il controllo all'handler appropriato registrato per quel probe.

2.1.3 Tracepoints

I Tracepoints[64] sono dei riferimenti statici stabiliti all'interno del codice sorgente del Kernel i quali, quando abilitati, possono essere utilizzati a tempo di esecuzione per estrarre dati dal Kernel rispetto al punto dove sono stati stabiliti. Inoltre l'utente può registrare una funzione *probe* che verrà invocata al verificarsi del Tracepoint, in maniera simile a quanto visto per i Kprobes. La differenza principale tra Tracepoints e Kprobe è che i primi sono uno strumento statico anziché dinamico, cioè vanno codificati esplicitamente tramite cambiamenti al codice del Kernel. La stessa natura statica dei Tracepoint però porta con se il vantaggio di una *Application Binary Interface* (ABI) stabile. Nello specifico ogni Tracepoint già esistente viene incluso in ogni versione successiva del Kernel Linux. L'uso più diffuso dei Tracepoints è quello del debugging del Kernel e diagnosi di problemi di performance, senza la necessità da parte dell'utente di conoscere in maniera approfondita tutti i dettagli del Kernel.

Un utente può visionare i Tracepoint disponibili stampando una lista degli pseudo-file presenti in */sys/kernel/debug/tracing/events*. Per usare un tracepoint occorre includere l'header *linux/tracepoint.h* e fare uso della macro *TRACE_EVENT*[64].

2.1.4 USDT probes

USDT (Userland statically Defined Tracing) è un meccanismo tramite il quale uno sviluppatore di un'applicazione può inserire dei probes direttamente all'interno del codice della propria applicazione. Ciò permette ad un utente in un secondo momento di effettuare il tracciamento di alcune operazioni di interesse. Gli USDT probes forniscono uno strumento a basso overhead

per ottenere visibilità del codice eseguito in user-space e il loro uso è principalmente per fini di debug. Gli USDT probe sono stati resi popolari da *DTrace* uno strumento sviluppato in origine da parte di Sun Microsystems per il tracciamento dinamico dei sistemi Unix. Dtrace è stato reso disponibile per Linux solo di recente a causa di problemi legati alla licenza. In ogni caso, gli sviluppatori del Kernel Linux hanno preso ispirazione dal lavoro per la creazione di Dtrace e hanno implementato i probe USDT. In maniera analoga a quanto accade per i Tracepoints, gli USDT probe richiedono l'inserimento di istruzioni specifiche nel codice sorgente da parte dello sviluppatore. La API più diffusa per inserire USDT probes nel proprio codice è la API di *SystemTap* e le relative macro *DTRACE_PROBE()*[45].

2.2 Meccanismi di raccolta eventi

2.2.1 Loadable Kernel Module (LKM)

Linux fornisce delle API potenti e complete per con cui le applicazioni possono interagire, ma spesso esse non forniscono un supporto sufficiente quando lo scopo è la visibilità completa del sistema. L'uso di un modulo del Kernel è solitamente necessario per interagire direttamente con le periferiche o effettuare operazioni che richiedono accesso ad informazioni privilegiate nel sistema.

Un *Loadable Kernel Module* (LKM) Linux è un file oggetto compilato che contiene codice mirato ad estendere il Kernel corrente (*base Kernel*). I meccanismi legati agli LKM permettono di aggiungere codice (o rimuoverlo) dal Kernel Linux a tempo di esecuzione. Gli LKM sono solitamente usati per aggiungere supporto per nuove periferiche hardware, filesystems, o per aggiungere nuove syscall. Quando la funzione fornita da un LKM non è più necessaria, esso può essere rimosso dalla memoria principale per far spazio ad altre risorse.

Senza questa capacità modulare il Kernel Linux occuperebbe molto spazio in memoria, dovendo supportare ogni driver nel proprio codice base. Inoltre sarebbe necessario il *rebuild* del Kernel ogni volta che si desidera aggiungere una nuova periferica o aggiornare un driver.

Gli LKM in Linux sono caricati e rimossi dalla memoria tramite il comando *modprobe*. I file oggetto LKM si trovano in */lib/modules* e sono identificati dall'estensione *.ko* ("kernel object").

Il comando *lsmod* fornisce una lista degli LKM caricati al momento.

Le operazioni implementate tramite LKM vengono eseguite in kernel space e hanno diversi vantaggi in termini di efficienza e soprattutto di accesso completo ad ogni risorsa del sistema. Ciò li rende una tecnologia molto utilizzata nella realizzazione di tracing framework, e diversi strumenti di monitoring e sicurezza fanno uso di LKM appositi.

Sviluppare correttamente un modulo è considerata un'attività complessa in quanto alterando il Kernel si rischia potenziale perdita di dati e corruzione dell'intero sistema. Inoltre il codice del kernel non è tutelato da meccanismi di sicurezza di cui godono le applicazioni nello user space. In caso di errore a tempo di esecuzione, l'intero sistema può bloccarsi, oppure nel caso pessimo l'errore può non manifestarsi in maniera evidente, e nel tempo può portare alla sovrascrittura o corruzione di importanti strutture dati e buffer del Kernel.

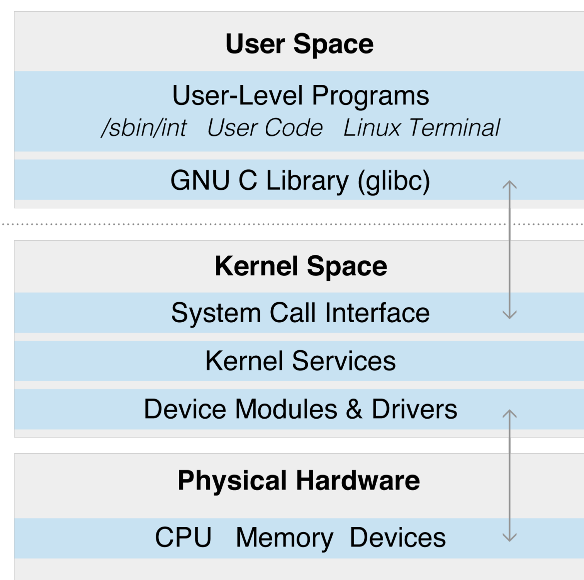

Figura 2.5: Moduli e Kernel space.

2.2.2 perf_events

Il framerok perf_events (conosciuto semplicemente come perf o Performance Counter for Linux, PCL) è uno strumento di tracing e analisi delle performance in Linux, disponibile dal 2009 con la versione del Kernel 2.6.31. Perf è capace di interfacciarsi con dati provenienti dai *performance counters* della Performance Monitoring Unit (PMU)[27], cioè registri hardware della

CPU che conteggiano il verificarsi di alcuni eventi come il numero di istruzioni eseguite, *cache miss* o fallimenti nel *branch prediction.* I performance counters sono una fonte di metriche e utili e perf fornisce un sistema di astrazione dall'hardware, ad esempio filtrando i counter per CPU e per task, e offrendo funzionalità di sampling, *profiling* e aggregazione. Perf inoltre può interfacciarsi con Tracepoints, Kprobes e Uprobes.

La API di perf è costituita dalla chiamata *perf_event_open()*[21]:

```
#include <linux/perf_event.h>
#include <linux/hw_breakpoint.h>

int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid, int cpu, int group_fd,
                    unsigned long flags);
```

La chiamata effettuata con opportuni argomenti restituisce un file descriptor dal quale si possono leggere dati con le chiamate di sistema standard (es. *read(2)*). Il vantaggio di perf rispetto ad altre soluzioni è che si tratta di una soluzione completamente contenuta nel Kernel e l'uso di front-end nello user space serve perlopiù a semplificare l'utilizzo della chiamata perf_event_open(). Sfortunatamente, la documentazione e risorse relative perf non sono considerate avanzate come quelle di altre soluzioni[69].

2.2.3 ftrace

Ftrace (abbreviazione di Function Tracer) è un tracing framework per il Kernel Linux. Originariamente progettato per il monitoring delle chiamate di funzioni del Kernel, le capacità di tracing di ftrace sono state ampliate per ricoprire una vasta gamma di eventi nel Kernel space[41].

Ftrace può utilizzare dati provenienti da diversi eventi definiti staticamente, come eventi relativi allo scheduling, interrupts, memory-mapped I/O e operazioni relative ai file system e alla virtualizzazione. Inoltre è disponibile il monitoring dinamico delle funzioni del kernel con la possibilità di generare dati aggregati riguardanti le chiamate. Ftrace può essere usato anche per misurare alcune latenze come la lunghezza degli interrupts. I meccanismi di tracing dinamico delle funzioni utilizzano un approccio simile a quello dei Kprobes[65].

Il Kernel Linux abilitato all'uso di ftrace può essere compilato abilitando l'opzione di configurazione *CONFIG_FUNCTION_TRACER*. L'interazione a tempo di esecuzione con ftrace avviene tramite i file contenuti nello pseudo-filesystem *tracefs* e la directory */sys/kernel/tracing*[15]. Questo significa che ftrace non richiede nessuna utility specializzata nello user-space per essere adoperato. Esistono però diversi strumenti specializzati che forniscono funzionalità più avanzate come l'analisi e la visualizzazione dei dati.

Ftrace è sviluppato principalmente da Steven Rostedt, ed è stato integrato nel Kernel Linux nella versione 2.6.27, rilasciata nell'Ottobre 2008.

2.2.4 eBPF (extended Berkeley Packet Filter)

BPF

Extended Berkeley Packet Filter (eBPF) è una tecnologia relativamente moderna che permette ad un utente privilegiato in ambiente Linux di inserire codice generico (con alcune restrizioni) all'interno del Kernel. Questo codice verrà eseguito in specifici momenti, solitamente all'avvenire di certi eventi di interesse all'interno del kernel-space. Il primo Berkeley Packet Filter (BPF) risale al 1992 ed era stato progettato per la cattura e il il filtraggio di pacchetti di rete che rispettassero alcuni filtri forniti dall'utente. I filtri erano implementati come programmi da mandare in esecuzione su una macchina virtuale in kernel-space [56].

eBPF

Nel 2014 l'estensione eBPF è stata sviluppata principalmente grazie al lavoro di Alexei Starovoitov e in seguito inclusa nel Kernel dalla versione 3.18 e costantemente arricchita di funzionalità [7]. La versione obsoleta è stata retroattivamente rinominata *classic BPF* (cBPF), e il Kernel Linux esegue solamente eBPF; il codice cBPF è tradotto in maniera trasparente in eBPF prima dell'esecuzione del programma [3]. Attualmente la macchina virtuale eBPF fa uso di un numero maggiore di registri (11 anziché gli originali 2 [19]) a 64 bit, con l'intento di facilitare la traduzione di istruzioni della macchina virtuale con istruzioni native dell'architettura sottostante[17]. Ciò ha reso possibile l'implementazione di un just-in-time compiler (JIT) per le architetture più popolari e un aumento delle prestazioni [44]. A partire dal Kernel

v1.18(2014), la macchina virtuale è stata esposta allo user-space tramite la chiamata di sistema *bpf()* e le API *uapi/linux/bpf.h*.

Sicurezza e il Verifier eBPF

I motivi per i quali si esegue una macchina virtuale all'interno del Kernel rimandano principalmente a motivi di sicurezza e facilità d'uso. Le funzionalità offerte da eBFP potrebbero essere svolte da un normale modulo del Kernel, ma la programmazione diretta del Kernel è delicata in quanto è relativamente facile causare blocchi, corruzione di memoria, introdurre vulnerabilità e in generale aumentare la superficie d'attacco di un sistema. La tecnologia eBPF permette di eseguire codice nel Kernel compilato *Just in Time* con prestazioni comparabili al codice nativo, ma tramite una macchina virtuale. Si tratta quindi di una soluzione per applicare il principio di *sandboxing* all'ambiente nel quale il codice è eseguito.

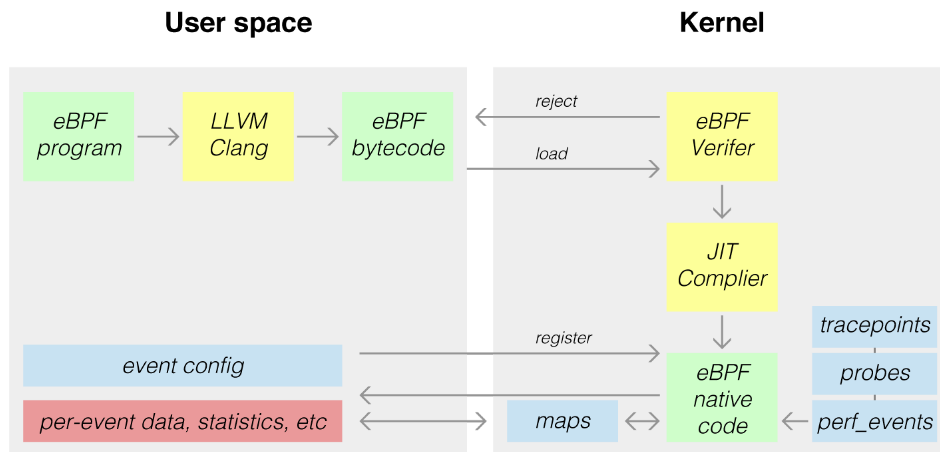

Figura 2.6: Workflow di un programma eBPF.

Un programma particolare chiamato *eBPF Verifier* ha il compito di prendere in input un programma eBPF scritto nell'apposito bytecode ed effettuare una serie di verifiche statiche mirate ad assicurare che l'esecuzione del programma eBPF non richieda un tempo indefinito. A causa dell'Halting Problem [70] questa verifica non è possibile per un ogni programma di lunghezza arbitraria, quindi il Verifier esegue la verifica controllando che il programma non possa superare un limite superiore stabilito di istruzioni, in ogni sua possibile esecuzione (il

limite attuale è 64K istruzioni). Questo tipo di verifica impone una serie di vincoli sulla scrittura dei programmi eBPF, tra cui:

- Il programma non deve essere composto da più di *BPF_MAXINSNS* istruzioni (attualmente 4096).
- Il numero di iterazioni di tutti i loop deve poter essere verificato a tempo di compilazione e non deve superare una soglia stabilita[66].
- Il numero di chiamate annidate di funzioni non deve superare un limite stabilito (attualmente 32).
- Assenza di istruzioni non raggiungibili dal flusso di esecuzione.
- Salti incondizionati malformati o *out-of-bound*.

eBPF Maps

La macchina virtuale eBPF e il suo linguaggio non permettono l'allocazione dinamica di memoria nello heap. Le Maps eBPF sono delle strutture dati particolari gestite dal Kernel e inizializzate tramite la chiamata *ebpf(2)* che permettono di mantenere lo stato attraverso diverse invocazioni degli handler di un programma eBPF, così come la condivisione di dati tra diversi programmi eBPF e verso lo user space.

Le map sono strutture dati di tipo associativo chiave/valore, e diversi tipi di map possono essere inizializzate tramite la chiamata *ebpf(2)*[9]. L'utilizzo delle map nello user space è mediato tramite alcune funzioni helper definite in *linux/bpf.h*:

```
/* Userspace helpers */
int bpf_map_lookup_elem(int fd, void *key, void *value);
int bpf_map_update_elem(int fd, void *key, void *value, __u64 flags);
int bpf_map_delete_elem(int fd, void *key);
int bpf_map_get_next_key(int fd, void *key, void *next_key);
};
```

Ulteriori Restrizioni

Oltre ai vincoli sopracitati, è importante sapere che la dimensione massima di un dato sullo stack di un programma eBPF è di 512 bytes, che sommato all'impossibilità di allocare memoria sullo heap, spingono all'uso di maps per manipolare tutti i dati non banali. In aggiunta, le funzioni che possono essere invocate da un programma eBPF sono un numero limitato e prendono il nome di *eBPF helpers*[8]. Gli helper sono lo strumento principale con il quale un programma eBPF può interagire con il Kernel e recuperare facilmente informazioni, e data l'origine di BPF, sono particolarmente complete le funzioni riguardanti il networking. La suite di helper BPF però manca di molte funzioni che fanno parte di librerie standard di altri linguaggi, per esempio le funzioni delle librerie standard del linguaggio C.

2.3 Front-End

2.3.1 ftrace

Ftrace permette la configurazione degli eventi e delle metriche da monitorare tramite lettura e scrittura da file. Nello specifico le interazioni con ftrace avvengono tramite la modifica dello pseudo-filesystem *sys/kernel/tracing*. Le interazioni basate su file permettono lo sviluppo di diversi strumenti per interagire con ftrace in maniera più comoda per lo sviluppatore. I progetti di maggior rilievo come front-end per ftrace sono trace-cmd[63], Catapult[28] e Kernelshark[16].

2.3.2 perf

Perf è il nome dell'applicazione front-end più diffusa che permette di interfacciarsi con perf_events tramite riga di comando, e offre un ricco insieme di comandi per ottenere e analizzare i dati forniti. Perf è uno strumento generico che implementa una varietà di sotto-comandi. Alcuni tra i comandi più usati sono i seguenti:

- perf stat: ottiene il conteggio di eventi
- perf record: memorizza gli eventi su file per un uso secondario

- perf report: filtra gli eventi per processo, funzione ecc.
- perf annotate: ottiene informazioni sul codice assembly collegato all'evento
- perf top: mostra il conteggio live degli eventi
- perf bench: esegue diversi *microbenchmarks* del Kernel

2.3.3 bcc

Il framework *BPF Compiler Collection* (BCC) è stato creato ed è correntemente mantenuto dall'IOVisor Project, un collaborativo della Linux Foundation che ha come scopo sviluppare nuovi strumenti e tecniche di visibilità rivolte al kernel-space.

Lo scopo di BCC è fornire un insieme di strumenti per rendere più lineare e semplice lo sviluppo di programmi eBPF. Una delle innovazioni introdotte da BCC sono delle interfacce per i linguaggi Python e Lua che semplificano notevolmente tutte le interazioni con eBPF (principalmente l'uso della syscall *bpf()* e le iterazioni con le maps). Inoltre BCC fornisce una libreria che rende più lineare il workflow la scrittura di programmi eBPF, definendo un linguaggio C ristretto per programmi eBPF, fornendo una serie di macro e funzioni helper e un wrapper per LLVM per la compilazione in bytecode eBPF.

Infine il progetto presenta una serie di tool mono-funzione e programmi di esempio che mirano ad esporre le potenzialità della tecnologa eBPF.

2.4 Container Security

2.4.1 Policy based

AppArmor

AppArmor[1] è un modulo del Kernel Linux che implementa funzionalità di sicurezza in maniera policy based. L'uso di AppArmor permette di stabilire un profilo di sicurezza per un dato processo con conseguente limitazione delle sue capacità, come accesso a interfacce di rete, linux capabilities, o permessi di lettura/scrittura di file. AppArmor permette di stabilire e associare dei profili di sicurezza a container anziché singoli processi. AppArmor può

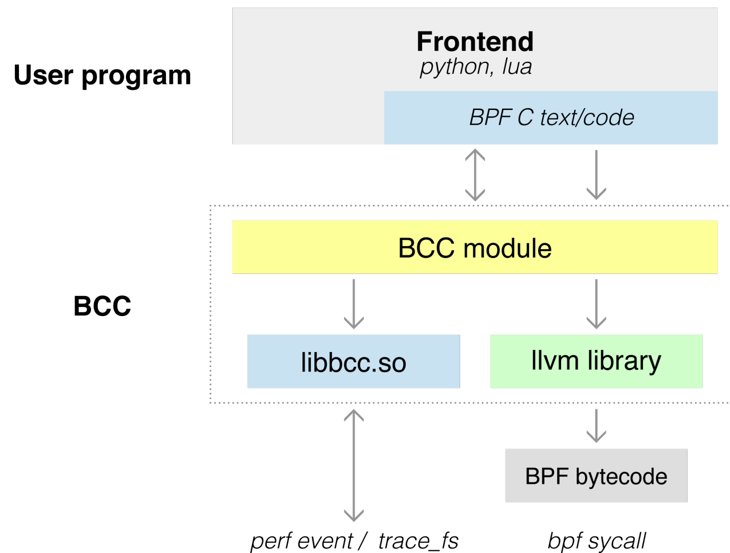

Figura 2.7: Framework BCC.

operare in modalità *learning*, cioè notificando eventuali violazioni del profilo di sicurezza a tempo di esecuzione, solitamente utilizzata per migliorare la definizione del profilo in uso. La modalità *enforcement* invece nega al processo o container l'esecuzione delle azioni non permesse. AppArmor solitamente carica i profili all'avvio dell'host e permette di caricare più profili learning/enforcement contemporaneamente.

AppArmor ha origine con Immunix, una distribuzione commerciale di linux incentrata sulla sicurezza. Successivamente è stata integrata nelle distribuzioni Ubuntu, Novell/SUSE e Mandriva. AppArmor è inclusa nel Kernel standard dalla versione 2.6.36.

L'uso di AppArmor e la definizioni di profili sono considerate di facile apprendimento. I file che definiscono i profili sono file di testo che definiscono regole basate principalmente su path. I path supportano meccanismi di *globbing* e *matching* identici a quelli della shell Bash, e le regole più specifiche sovrascrivono quelle più generiche. Quello che segue è un esempio di profilo AppArmor per il programma *tcpdump*. Quello che segue è un esempio di una regola Falco definita all'interno di un file di regole per MongoDB, un popolare database relazionale.

```
#include <tunables/global>
```

```
/usr/sbin/tcpdump {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  #include <abstractions/user-tmp>

  capability net_raw,
  capability setuid,
  capability setgid,
  capability dac_override,
  network raw,
  network packet,

  # for -D
  capability sys_module,
  @{PROC}/bus/usb/ r,
  @{PROC}/bus/usb/** r,

  # for -F and -w
  audit deny @{HOME}/.* mrwkl,
  audit deny @{HOME}/.*/ rw,
  audit deny @{HOME}/.*/** mrwkl,
  audit deny @{HOME}/bin/ rw,
  audit deny @{HOME}/bin/** mrwkl,
  @{HOME}/ r,
  @{HOME}/** rw,

  /usr/sbin/tcpdump r,
}
```

Sysdig Falco

Sysdig è una compagnia leader nel settore di monitoring e sicurezza dei sistemi, e Falco[14] è un progetto di Sysdig relativo al rilevamento delle anomalie a livello delle applicazioni e container. Falco ottiene i dati dal framework di Sysdig, che utilizza una varietà eterogenea

di fonti dei dati. L'utente può generare facilmente dei file di configurazione che definiscono condizioni sotto le quali il comportamento dell'applicazione o container è considerato o meno normale. A tempo di esecuzione vengono generati opportuni alert se una condizione viene violata. I file di configurazione strutturano le regole tramite il linguaggio di markup YAML e permettono di definire gli eventi considerati anomali anche su condizioni complesse.

```
- rule: Unexpected spawned process mongo
  desc: Detect a process started in a mongo container outside of
        an expected set
  condition: spawned_process and not proc.name in
           (mongo_allowed_processes) and app_mongo
  output:    Unexpected process spawned in mongo container
           (command=%proc.cmdline pid=%proc.pid user=%user.name
           %container.info image=%container.image)
  priority: NOTICE
```

AquaSecurity e Twistlock

AquaSec e Twistlock[2, 22] sono compagnie leader nel settore della sicurezza di applicazioni e specializzate nella sicurezza di containers, piattaforme cloud e applicazioni cloud-native, e pipeline DevOps. I prodotti orientati alla sicurezza dei container Docker offrono scansioni statiche delle Docker images prima della loro esecuzione sugli host, per rivelare possibili vulnerabilità come malware, embedded secrets ed errori di configurazione. Il supporto alla sicurezza a tempo di esecuzione per i container è simile per entrambe le piattaforme e prevede principalmente la definizione di policy da parte dell'utente. Nel caso di Twistlock è prevista anche l'integrazione con tecniche di machine learning per il *profiling* del comportamento normale di un container.

2.4.2 Anomaly based

Il concetto di anomalia (*outlier* in gergo) in generale dipende da come viene creato l'insieme di dati che si prende in esame e non ha una definizione univoca [79, 51]. Definizioni astratte e ragionevoli di anomalia possono essere *"An observation (or subset of observations) which ap-*

pears to be inconsistent with the remainder of that set of data"[62] e *"Patterns in data that do not conform to a well defined notion of normal behavior"*[31].

I sistemi di rilevamento di anomalie possono essere divisi in tre famiglie principali: sistemi *supervised*, *unsupervised* e *semisupervised* [51, 31].

Nel caso dei sistemi supervised l'insieme dei dati utilizzati per creare un modello predittivo (detto *training set*) contiene eventi che sono stati etichettati a priori come normali o anomali. Un approccio tipico prevede che il sistema formi un modello predittivo alla fine della fase di training e che confronti ogni nuova istanza di eventi con esso per classificarla come anomala o normale. Le sfide principali di questo approccio sono la relativa difficoltà di creare un insieme esaustivo di eventi anomali e bilanciare la quantità di eventi anomali e non nel training set.

I sistemi unsupervised non necessitano di un training set, e il loro funzionamento si basa sull'assunzione che gli eventi anomali sono molto più rari di quelli normali. Nonostante possono essere adattati ad un'ampia gamma di casi, questi metodi possono soffrire di un degrado nelle prestazioni quando incontrano un insieme di eventi che non rispecchia la loro assunzione[31].

I sistemi semisupervised utilizzano un training set formato solamente da istanze di comportamento normale. Siccome non richiedono esempi di comportamento anomalo, sono in generale più applicabili dei sistemi supervised [31]. Un approccio comune in questi sistemi è quello di costruire un modello predittivo per il comportamento normale e usare tale modello per identificare le anomalie nei dati usati come test.

In teoria è possibile creare sistemi semisupervised che utilizzano come training set insiemi formati da soli eventi anomali, e utilizzare il modello generato per identificare le anomalie nei dati usati come test. Anche se diverse tecniche sono state proposte [37, 36], questo approccio è utilizzato raramente, principalmente perché è difficile ottenere un training set che copre in maniera esaustiva ogni possibile comportamento anomalo [31].

Un altro aspetto che differenzia i sistemi di rilevamento di anomalie è la maniera in cui le stesse sono classificate, cioè la forma che assume l'output del sistema di rilevamento. I due tipi di output principali sono basati su *score* o su *labels*.

Gli output di tipo score assegnano alle istanze dei dati in entrata un punteggio che rispecchia la probabilità stimata che esso costituisca un'anomalia. L'utente tipicamente può ordinare

i risultati per punteggio e selezionare una soglia sopra la quale considerare gli eventi come anomali. Gli output di tipo label assegnano alle istanze un'etichetta binaria che le classifica come normali o anomale. Mentre le tecniche basate su score permettono all'utente di specificare direttamente una soglia di classificazione contestualmente al caso d'uso, le tecniche basate su label possono essere controllate indirettamente tramite la modifica di parametri interni.

Strumenti basati su metriche

Studi recenti hanno come oggetto il rilevamento di anomalie nei container tramite l'uso di dati sulle metriche.

Du et al.[40] hanno realizzato un sistema di rilevamento anomalie in tempo reale che usa come fonte di dati metriche relative ai containers quali l'uso della CPU, memoria e network. I dati sono utilizzati come input di un sistema basato sul machine learning. Gli eventi anomali sono collegati ad eventi come i CPU faults, memory faults, latenza di rete e perdita di pacchetti.

Zou et al.[80] hanno fatto uso di tre categorie diverse di dati sui contaienr quali log, metriche (CPU, memoria,...) e informazioni di configurazione per classificare anomalie. La loro ricerca fa uso di una versione ottimizzata dell'*isolation forest*, un algoritmo solitamente utilizzato nell'unsupervised learning[31]. Le anomalie sono divise in categorie associate alle risorse del sistema quali CPU, memoria, I/O, network. Esempi di anomalie includono CPU *spin locks*, overflow della memoria e congestione di rete. Sia i lavori di Gao et al. che di Du et al. classificano le anomalie basandosi su metriche relative all'uso di risorse a tempo di esecuzione, ma non è detto che l'attacco di un agente malevolo implichi variazioni significative in queste metriche.

Strumenti basati su syscall

Altri studi invece utilizzano dati sulle syscall come elementi significativi per il rilevamento delle anomalie nei container.

I primi studi riguardo lo studio delle syscall e della loro frequenza come elemento utile per la classificazione di eventi anomali precedono la tecnologia dei containers. I primi esempi vedono Hofmeyer et al.[52, 46] utilizzare metodi statistici su una trace relativa a singoli processi per classificare eventuali comportamenti anomali. Maggi et. al[55] includono dati sugli

argomenti specifici con cui le syscall sono chiamate e modelli di Markov per modellare il comportamento normale di processi. Muratza et al.[58] sfruttano l'interazione semantica tra le syscall. L'idea alla base dello studio è rappresentare le syscall come stati tra moduli del kernel, analizzare l'interazione tra gli stati e identificare le anomalie confrontando la probabilità di occorrenza degli strati in una trace anomala rispetto ad una normale. Lo studio di Grimmer et al.[50] invece prevede di modellare la trace di syscall come grafi di probabilità e applicare ad essi tecniche statistiche.

Alarifi et al.[25, 26] hanno esteso tecniche basate sull'analisi delle frequenze delle syscall al monitoring del comportamento di intere macchine virtuali anziché di singoli processi, con risultati promettenti. Infine Abed et al.[24] utilizzano con successo tecniche simili riducendo l'insieme di syscall osservate a quelle delle trace di un singolo container. Il numero ridotto di processi interni ad un container rispetto a quelli di una macchina virtuale permette in generale di ottenere una precisione maggiore nell'uso di tecniche basate sulla frequenza delle syscall.

KubAnomaly

KubAnomaly[67] è uno strumento di rilevamento delle anomalie per container pensato principalmente per container gestiti da Kubernetes[72], ma la cui strategia è applicabile ai container a prescindere dall'uso di un sistema di orchestration. KubAnomali utilizza Sysdig come framework di raccolta dati. L'idea alla base è selezionare un ristretto insieme di eventi chiave che possono identificare facilmente il comportamento di un container. L'insieme degli eventi di interesse è rappresentato in figura 2.8, ed è composto da syscall ed eventi di accesso al filesystem considerati significativi. I dati vengono normalizzati utilizzando il framework *scikit*[77], e il comportamento normale e anomalo è classificato tramite un sistema di unsupervised learning che fa uso di framework di machine learning come *Keras*[71] e *TensorFlow*[78]. La scelta di ridurre al minimo l'insieme di eventi interessanti è principalmente dovuta all'intendo di mantenere basso l'overhead, in quanto i modelli di machine learning introducono un carico computazionale non indifferente rispetto al volume di dati usati per il training.

2.5 Soluzione Proposta

Al termine di una prima fase di ricerca, i Kprobe (nello specifico i Jprobe e i Kretprobe) sono stati selezionati come fonte di eventi per i fini del progetto. Il motivo principale della scelta è dato dal fatto che i Kprobe sono uno strumento di tracciamento dinamico già incluso nel Kernel standard e quindi non richiedono modifica del codice di cui si vuole tenere traccia. Inoltre la loro implementazione aggiunge un overhead minimo, che si manifesta solo una volta che il probe è installato [43].

La tecnologia eBPF è stata scelta dopo un'analisi dei diversi tracing framework che possono usare direttamente i Kprobe come fonti dei dati. La scelta di BCC come front-end lato user space è stata ovvia in quanto si tratta attualmente del framework più comodo e semplice per scrivere programmi eBPF e interagire con la macchina virtuale e le maps, senza però rinunciare a nessuna delle possibilità offerte dalla tecnologia eBPF.

La scelta di eBPF ha inoltre l'intenzione di evidenziare come questa tecnologia sia valida nell'ambito dell'osservabilità in tutti gli ambiti e non solo in quello del networking. Un esempio recente dell'uso di eBPF per l'osservabilità è dato da Sysdig Falco, che l'ha recentemente adottata come alternativa al tracing framwework basato su moduli del Kernel[29].

Per quanto riguarda la strategia per implementare la sicurezza di containers e processi, si è deciso di optare per una modalità anomaly-based piuttosto che una policy-based. La scelta è data dall'obiettivo di realizzare uno strumento che non richiedesse conoscenza a priori dell'entità da monitorare e configurazione esplicita in base ad essa. In ogni caso i due approcci non sono da considerarsi mutualmente esclusivi, e l'uso di un modello predittivo anomaly-based può essere unito ad un sistema di alerts al verificarsi di condizioni specifiche definite esplicitamente. Ad esempio Twistlock integra parzialmente un sistema di apprendimento dinamico con la sua architettura prevalentemente policy-based.

La scelta di quali dati utilizzare e in che maniera è stata influenzata principalmente dall'obiettivo di mantenere basso l'overhead anche nella fase di elaborazione degli eventi recuperati e al tempo stesso di riuscire a rilevare con successo i comportamenti anomali. La maggior parte delle strategie proposte negli studi citati in 2.4.2 che si basano su metriche impiegano tecniche di machine learning e statistiche che portano con se un certo carico computazionale.

Le tecniche basate su eventi come syscall invece trattano eventi più espliciti delle metriche (le syscall ed eventualmente i loro argomenti) e solitamente questo si traduce in algoritmi più semplici per il rilevamento delle anomalie.

Si è optato quindi per un approccio basato esclusivamente sull'analisi delle frequenze delle syscall che non comprendesse modellare negli eventi anche gli argomenti con cui esse sono invocate e i valori di ritorno. Diversi studi principali citati in 2.4.2 usano dati aggiuntivi sulle syscall come l'interazione semantica tra di esse e gli argomenti con cui sono invocate, o tentano di rappresentare gli stati dei moduli del kernel tramite esse. Questi approcci hanno in comune però l'uso di algoritmi che hanno un costo computazionale non banale anche in fase di monitoring. Il tipo di eventi considerati significativi è da un certo punto di vista affine a quello dello strumento KubAnomaly, a meno di segnalare esplicitamente gli eventi che riguardano l'accesso alle root directories.

Gli eventi generati dall'invocazione delle syscall sono stati processati utilizzando una variante dell'algoritmo trattato da Abed et al.[24], che introduce un overhead estremamente basso sia in fase di learning che di monitoring. Nello specifico è stato esteso il suo funzionamento per il monitoring in tempo reale.

Event name	Description	Type
Read	Container exhibits read behavior	System call
File_IO	Container contains file operation	System call
Write	Container exhibits write behavior	System call
Accept	Container accepts connection on a socket	System call
Network_Http	Container has http flow	System call
Clone	Container creates a new program	System call
Select	Container monitors socket, waiting for data	System call
Poll	Container waits for stream event	System call
Rename	Container exhibits rename behavior	System call
Chdir	Container changes the current process working directory	System call
Kill	Container sends a kill signal	System call
Postgresql	Container exhibits postgresql flow	System call
Mkdir	Container exhibits mkdir behavior	System call
Brk	Allocate a small amount of memory	System call
Mmap	Allocate memory	System call
Munmap	Free memory	System call
Bin	Container accesses file under bin directory	Root directory access
Home	Container accesses file under home directory	Root directory access
Etc	Container accesses file under etc directory	Root directory access
Boot	Container accesses file under boot directory	Root directory access
Dev	Container accesses file under dev directory	Root directory access
Host	Container accesses file under host directory	Root directory access
Media	Container accesses file under media directory	Root directory access
Mnt	Container accesses file under mnt directory	Root directory access
Opt	Container accesses file under opt directory	Root directory access
Proc	Container accesses file under proc directory	Root directory access
Root	Container accesses file under root directory	Root directory access
Run	Container accesses file under run directory	Root directory access
Srv	Container accesses file under srv directory	Root directory access
Usr	Container accesses file under usr directory	Root directory access
Var	Container accesses file under var directory	Root directory access

Figura 2.8: Eventi tracciati da KubAnomaly.

Capitolo 3

Progetto

3.1 Struttura

Secondo quanto detto nel capitolo precedente, è stata portata avanti una valutazione dei punti di forza delle singole tecnologie di tracing in ambiente Linux rispetto a quello che è lo scopo del progetto. I Kprobe sono stati scelti come event sources, eBPF come tracing framework e BCC come front-end.

L'elaborato del progetto consiste in uno strumento costituito da due programmi. Un primo programma *Filter* scritto in un linguaggio *restricted-C* viene compilato in bytecode eBPF ed eseguito sulla macchina virtuale eBPF. Questo programma implementa la logica necessaria al filtraggio e l'aggregazione degli eventi di interesse, nel nostro caso le syscall relative all'esecuzione di un container sull'host. Un secondo programma *Classifier* è scritto in linguaggio Python e fa uso del front-end BCC per gestire le chiamate bpf() e interagire con il Filter. Lo strumento è progettato per funzionare in due modalità, dette *learning* e *monitoring*.

In modalità learning lo strumento prende come argomento l'identificatore di un container o di un singolo processo e tenta di dedurre il comportamento "normale" per esso. Il Filter considera come eventi di interesse le syscall relative al container preso in esame ed invia i dati al Classifier. Il Classifier raccoglie gli eventi dal Filter, li aggrega ed applica su di essi un algoritmo (mostrato in seguito) per creare un file che classifica il comportamento "normale" del container (o processo) in esame.

In modalità monitoring lo strumento prende come argomento l'identificatore di un contai-

ner o processo per il quale ha già classificato il comportamento normale e tenta di rilevare in tempo reale anomalie per lo stesso quando esso viene mandato in esecuzione sull'host.

3.2 Prerequisiti

La versione minima del Kernel Linux che implementa le cosiddette *main feature* di eBPF è la 3.15, ma la versione minima necessaria per eseguire correttamente lo strumento oggetto della relazione è la 5.6, in quanto funzionalità aggiuntive e funzioni helper eBPF richiedono versioni minime del Kernel superiori alla 3.15 [5]. Inoltre si deve essere certi che il Kernel sia stato compilato con i seguenti *flags* attivi:

```
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
# [optional, for tc filters]
CONFIG_NET_CLS_BPF=m
# [optional, for tc actions]
CONFIG_NET_ACT_BPF=m
CONFIG_BPF_JIT=y
# [for Linux kernel versions 4.1 through 4.6]
CONFIG_HAVE_BPF_JIT=y
# [for Linux kernel versions 4.7 and later]
CONFIG_HAVE_EBPF_JIT=y
# [optional, for kprobes]
CONFIG_BPF_EVENTS=y
```

che solitamente possono essere consultati facendo riferimento ai file
/proc/config.gz o */boot/config-<kernel-version>*.

Inoltre, occorre che sia presente sull'host un interprete Python e il framework BCC [4].

3.3 Container in Linux

Container è il nome più comunemente associato alle tecnologie di *OS-Level Virtualization*[75], che hanno come scopo la creazione e gestione di diverse istanze isolate di user-space su un

singolo host. Per container si intende una di queste istanze, e ogni container ha visione e accesso ad un sottoinsieme delle risorse dell'host.

L'implementazione dei container in Linux si basa principalmente sui meccanismi dei *Cgroups* e dei *Namespaces*[11, 20]. I cgroups forniscono la possibilità di limitare l'uso da parte di singoli processi delle risorse del sistema (CPU, memoria, disk I/O, traffico di rete, ecc.), mentre i namespaces permettono di alterare la visione che i processi hanno dell'host. I vari namespaces sono stati introdotti in maniera incrementale nel tempo, e un adeguato supporto per l'implementazione dei container in Linux è stato raggiunto con la versione del Kernel 3.8 grazie all'introduzione degli User namespace [53].

3.3.1 Namespaces

I namespaces (*NS* d'ora in avanti) presenti attualmente in Linux (Kernel 5.6) sono i seguenti:

- Mount (mnt)

  I Mount NS implementano l'isolamento dei mount points visti dai processi appartenenti ad ogni istanza di NS. I processi appartenenti a diverse istanze Mount NS vedranno in generale diverse gerarchie di directories.

- Process ID (pid)

  I PID NS implementano l'isolamento degli spazi dei process ID (*PID*), il che significa che processi appartenenti a diversi PID NS possono avere lo stesso PID. In particolare, il PID assegnato al primo processo eseguito in un PID NS è 1, analogamente al PID assegnato al primo processo eseguito in ambiente Linux (es: *init, systemd*). Inoltre, nuovi processi ereditano il PID NS dal padre e syscall che generano nuovi processi quali *fork(2), vfork(2) e clone(2)* produrranno processi con PID unici per quel dato PID NS.

- Network (net)

  I Network NS implementano copie logiche dello stack di rete. Ogni Network NS possiede le proprie interfacce di rete e indirizzi IP, regole di routing e altre risorse relative al networking.

- Interprocess Communication (ipc)

  Gli IPC NS implementano l'isolamento di alcuni meccanismi di *Inter-Process Communication* (IPC), nello specifico quelli che creano istanze di oggetti IPC non identificati da pseudofile (es. pipes). I tipi di IPC oggetto degli IPC NS sono attualmente i *System V IPC objects* e le POSIX *message queues*.

- UTS

  Gli UTS NS implementano l'isolamento di due identificatori relativi all'host: l'*hostname* e il *NIS domain name*. I processi che fanno parte dello stesso UTS NS otterranno gli stessi risultati da chiamate quali *uname(2)*, *gethostname(2)*, *getdomainname(2)*, che potranno essere in generale diversi da quelli di processi in altri UTS NS.

- User ID (user)

  Gli User NS implementano isolamento di identificatori e attributi relativi alla sicurezza e ai meccanismi Linux di *privilege escalation*, in particolare *user ID* e *group ID* e capabilities. Ad esempio, l'user e group ID di un processo possono differire se osservati dall'interno o dall'esterno di un User NS. In particolare, un processo può avere un user ID uguale a 0 (privileged user, superuser) eseguire azioni privilegiate all'interno di un namespace, ma avere un user ID *unprivileged* fuori da quel User NS.

- Control group (cgroup) Namespace

  I Cgroup NS permettono di manipolare la vista che i processi hanno del proprio cgroup filesystem, relativo ai pathname */proc/[pid]/cgroup* e */proc/[pid]/mountinfo*.

- Time

  I Time NS permettono ai processi di osservare diversi *system times* in maniera simile a quanto gli UTS NS fanno per le informazioni hostname e domain name.

3.3.2 Docker

L'implementazione attualmente più popolare dei container software è fornita dal progetto *Docker*[30]. Docker permette di identificare un container in tre maniere: un *long UUID* da 64 carat-

teri, uno *short UUID* (i primi 12 caratteri del long UUID) e un *Name* definito dall'utente. Nello specifico lo short UUID corrisponde al nome dell'UTS NS utilizzato da Docker per il rispettivo container, e come si vedrà in seguito per determinare se un processo fa parte di un container Docker[13].

3.4 Algoritmo BoSC

Il sistema di rilevamento anomalie realizzato è di tipo semisupervised e genera un output labeled, ed usa una variante di un algoritmo già proposto da Abed et al. per il rilevamento di anomalie basata sull'analisi delle syscall [24]. La modifica principale consiste nella capacità di effettuare la costruzione del modello di comportamento normale e il rilevamento delle anomalie in tempo reale.

L'idea alla base è quella di tenere traccia di tutte le syscall effettuate dai processi appartenenti ad un singolo container durante la sua esecuzione e di raggrupparle in strutture dati che tengono conto della loro prossimità temporale. Questi dati aggregati vengono filtrati ed utilizzati per creare un database che rappresenta il comportamento normale.

Gli aggregati prendono il nome di *Bags of Sistem Calls* (BoSC) e sono stati proposti inizialmente da Kang et al. nel 2005 in merito all'analisi di anomalie dei singoli processi Linux [35]. Lavori successivi hanno dimostrato risultati positivi nell'utilizzo di questa rappresentazione per il rilevamento di anomalie nelle macchine virtuali [25, 60, 26]. L'algoritmo incorpora l'aggregazione dei dati in BoSC con il concetto di *sliding window* (finestra scorrevole), già usato per l'aggregazione degli eventi relativi alle syscall [46].

3.4.1 Definizioni

Prendiamo in esame una sequenza di syscall S effettuate ordinate in ordine temporale, $S :< s_1, s_2, ..., s_n >$, con s_i l'i-esima syscall. Per sliding window di ampiezza z si intende una porzione di lunghezza fissata della sequenza, caratterizzata da indici di inizio e fine: $S(x_i, x_f) :< s_i, s_{i+1}, ..., s_{f-1}, sf >$. Per sliding window *successiva* a $S(x_i, x_f)$ si intende la sliding window $S(x_{i+1}, x_{f+1})$

Data una sequenza di syscall, una BoSC è una lista ordinata $< c_1, c_2, ..., c_n >$ dove n è il numero totale di syscall distinte e c_i è il numero di occorrenze di una syscall s_i nella sequenza presa in esame.

Si noti che data una BoSC $< c_1, c_2, ..., c_n >$ costruita utilizzando una sliding window di ampiezza z come input, vale $\sum_{i=1}^{n} c_i = z$. Un esempio di BoSC con $n = 20$ syscall distinte e $z = 10$ ampiezza della sliding window è il seguente:

$$[0, 1, 0, 2, 0, 0, 0, 0, 1, 0, 4, 0, 0, 0, 0, 0, 1, 0, 0, 1]$$

Data una sequenza di syscall S, $S =< e_1, e_2, ..., e_n >$ è un partizionamento di S in *epochs* (epoche) di lunghezza prefissata k. Ad esempio $e_1 :< s_1, s_2, ..., s_k >$ e in generale $e_i :< s_{ki+1}, s_{ki+2}, ..., s_{k(i+1)} >$

3.4.2 Funzionamento

Learning

In fase di learning l'algoritmo prende in input una sequenza di syscall effettuate che costituiscono il training set, detta *trace*. Nel caso dello strumento in esame la trace è costituita da tutte le syscall effettuate dal container o processo da monitorare.

L'algoritmo scorre l'intera trace e mantiene aggiornata una struttura dati che tiene il conto del numero di occorrenze per ogni diversa syscall. L'algoritmo stabilisce inoltre una soglia *other_treshold* che serve a caratterizzare le syscall raramente usate. Questa strategia viene impiegata durante la creazione del modello di comportamento normale ai fini del risparmio di spazio, memoria, e tempo di elaborazione [24, 25]. La soglia è pari al numero totale di tipi diversi di syscall di cui il programma tiene traccia. Una syscall che ha nella trace un numero di occorrenze minore della soglia verrà classificata come "other".

L'algoritmo divide l'intera trace in epoch e, per ogni epoch, scorre la sliding window generando le BoSC relative. Le syscall che non superano la other_treshold vengono incluse e conteggiate nella BoSC aggregandole come un tipo particolare "other". Sulla base dei risultati di lavori precedenti si è scelta un ampiezza della finestra pari a 10 [68, 52].

Le BoSC vengono inserite in una *hash table* che ha come chiavi i valori delle BoSC stesse e come valori dei contatori di frequenza che tengono il conto delle occorrenze delle loro oc-

correnze. Quando una BoSC viene generata e non è presente nella tabella, essa viene inserita con frequenza 1. Se la stessa BoSC viene generata in futuro, il valore della sua frequenza viene incrementato di 1.

L'algoritmo mantiene in memoria lo stato della hash table corrente e anche dello stato che essa aveva alla fine della epoch precedente. Inoltre al termine di ogni epoch k viene calcolato C_k vettore dei cambiamenti di frequenza, cioè un vettore di interi positivi i cui valori rappresentano l'aumento delle occorrenze di ogni BoSC nella hash table allo stato k rispetto al suo stato nell'epoch $k-1$.

Alla fine di ogni epoch k viene confrontato il vettore di cambiamento di frequenza C_k dell'epoch appena terminata con il vettore C_{k-1}. Il confronto viene effettuato utilizzando come metrica la similarità del coseno:

$$\cos\theta_k = \frac{C_k \cdot C_{k-1}}{\|C_k\| \cdot \|C_{k-1}\|} = \frac{\sum_{i=1}^{n_k} C_k[i] C_{k-1}[i]}{\sqrt{\sum_{i=1}^{n_k} C_k[i]^2} \sqrt{\sum_{i=1}^{n_{k-1}} C_{k-1}[i]^2}}$$

Dove $C_k[i]$ è l'i-esimo elemento del vettore C_k e n_k è il numero delle differenti BoSC presenti nella hash table dopo l'epoch k. Il valore della similarità del coseno è uguale a 1 quando due vettori sono identici.

La condizione di arresto del processo di learning è che la metrica di similarità di due epoch consecutive sia maggiore di un certa soglia *similarity_treshold*. Il valore utilizzato per la soglia è di 0.99. Al termine, lo stato più recente della hash table descrive il comportamento normale del container o processo.

Monitoring

L'algoritmo prende in input una hash table risultante dall'esecuzione della modalità di learning di un container o processo e una trace di syscall in tempo reale relative all'esecuzione dello stesso container o processo.

La trace viene utilizzata per generare le BoSC tramite sliding window in maniera analoga alla modalità di learning. Ogni BoSC generata viene usata come chiave di ricerca nella hash table che descrive il comportamento normale appreso. Se la ricerca fallisce, ciò rappresenta una mancata corrispondenza (*mismatch*) ed un potenziale elemento di anomalia. Un contatore tiene il conto del numero di mismatch nella epoch corrente, e quando esso supera una certa

soglia *mismatch_treshold*, viene segnalata un'anomalia, e il contatore viene azzerato. Il valore di soglia *mismatch_treshold* è uguale al 10% della ampiezza delle epoch.

Limite superiore delle Anomalie

Per come è stato definito l'algoritmo, date $d_{trace}, d_w, d_e, tresh$ rispettivamente le dimensioni della trace, sliding window, epoch e valore di *mismatch_treshold*, il numero massimo di anomalie rilevabili data una trace totalmente anomala (ad esempio formata da syscall che non appaiono mai nel comportamento normale) equivale a:

$$\lfloor\lfloor\frac{d_e - d_w}{tresh}\rfloor\frac{d_{trace}}{d_e}\rfloor \approx \frac{d_{trace}}{tresh}$$

3.5 Il Filter

3.5.1 Funzionalità

Il programma eBPF implementa le funzioni handlers da collegare ai Kprobe, la logica necessaria a filtrare le syscall di interesse e l'invio dei dati al programma Classifier. Il Filter deve riconoscere se una syscall appartiene effettivamente al container o processo osservato e in caso positivo costruire un report che raccoglie alcuni dati sull'evento. L'invio di questi report al Classifier costituisce la trace sulla quale applicare l'algoritmo di rilevamento anomalie.

3.5.2 Comunicazione con il Classifier e Maps

Come si è detto i programmi eBPF non hanno accesso a variabili globali e heap, e strutture dette maps sono usate per mantenere lo stato tra diverse invocazioni del programma eBPF. Il FIlter fa uso di maps eBPF inizializzate tramite macro BCC *BPF_HASH* per mantenere in memoria le informazioni di cui ha bisogno:

- *config_map* Utilizzata per flags di configurazione. Il Classifier scrive direttamente in questa map prima dell'avvio del programma eBPF.
- *pids_map* Come verrà mostrato in seguito, è utilizzata per l'identificazione di container e processi.

- *taskname_buf* Utilizzata per ricevere l'id del container o processo da monitorare. Il Classifier scrive direttamente in questa map prima dell'avvio del programma eBPF.

Inoltre il Filter inizializza un ring buffer *events* tramite il quale inviare gli eventi al Classifier, inizializzato tramite la macro BCC *BPF_PERF_OUTPUT*.

3.5.3 Identificare Container e Processi

pids_map

Il Filter fa uso di una map *pids_map* eBPF di tipo BPF_HASH (array associativo) per mantenere un insieme rappresentante tutti i PID dei processi da monitorare. Quando un probe viene innescato da una syscall, l'handler eBPF definito per quel probe ottiene il PID del processo che ha effettuato la syscall tramite la funzione eBPF helper bpf_get_current_pid_tgid(void)[8]. Se il PID è presente nella pids_map, l'handler ottiene ulteriori dati sull'evento tramite altre eBPF helpers e dalla task_struck ottenuta tramite *bpf_get_current_task()*[8].

Containers

Nel Kernel-space, un container è rappresentato da un insieme di cgroup e namespace, e non esiste un identificatore che mette in relazione gli elementi di tale insieme. L'introduzione di container id espliciti in Kernel-space è stata proposta ma tuttora non accettata [34]. In generale non è possibile identificare tramite un metodo unico i container senza conoscere a prescindere la loro implementazione (Docker, LXC, Kubernetes...). Le implementazioni più diffuse quali Docker e Kubernetes fanno però corrispondere il container id con il nome dell'UTS Namespace al quale i processi del container appartengono. Le informazioni riguardanti i namespaces a cui il processo appartiene possono essere ottenute tramite la struct *nsproxy*[18], a sua volta membro della struct *task* (ottenuta tramite l'helper eBPF *bpf_get_current_task()*):

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
```

```
        struct net              *net_ns;
        struct time_namespace *time_ns;
        struct time_namespace *time_ns_for_children;
        struct cgroup_namespace *cgroup_ns;
};
```

Il Filter ottiene dal Classifier l'id (short UUID) del Docker container da monitorare da una map apposita. In risposta ad ogni syscall *execve(2), ececveat(2)*, l'UTS Namespace del processo relativo alla chiamata viene confrontato con il'id, e in caso positivo il PID del processo viene inserito nella pids_map di PID da monitorare. In questa maniera il processo viene identificato con successo come appartenente al container specificato. L'UTS NS è ottenuto navigando nella task_struct del processo tramite la seguente funzione [48]:

```
static char *get_task_uts_name(struct task_struct *task) {
    return task->nsproxy->uts_ns->name.nodename;
}
```

Processi

Il Filter ottiene il nome dell'eseguibile da monitorare dal Classifier tramite un'apposita map in maniera analoga a quanto visto per i Container. In risposta ad una *execve(2), execveat(2)*, viene ottenuto il *comm* (nome dell'eseguibile meno il path) relativo al processo corrente, tramite l'helper eBPF *bpf_get_current_comm()*. Se il comm e corrisponde al nome dell'eseguibile contenuto nella map, il PID del processo corrente viene aggiunto alla pids_map.

Forking

Sia nel caso dei container che dei singoli processi, si deve tener conto di eventuali processi figli di processi già monitorati. In entrambi i casi i PID dei processi figli vengono aggiunti in maniera efficiente alla pids_map tramite l'uso di Kretprobe installati sulle chiamate capaci di generare processi figli, ovvero *fork(2), vfork(2), clone(2)*. Gli handlers dei Kretprobe ottengono il valore di ritorno di queste chiamate se esse sono state effettuate da processi già appartenenti a pids_map. I valori di ritorno sono ottenuti tramite la macro *PT_REGS_RC(ctx)*, e valutati se

maggiori 0 (0 è il valore restituito al processo padre, -1 il valore nel caso di errore). In caso positivo il PID del processo corrente (il processo figlio) viene aggiunto alla pids_map.

Rimozione dalla pids_map

Un handler particolare è associato al Kprobe installato sulla syscall *exit(2)*. In aggiunta alle operazioni in comune con gli handler delle altre syscall, questo handler rimuove il PID del processo corrente dalla pids_map. Questa operazione è necessaria in quanto lo strumento potrebbe monitorare un processo e, dopo la terminazione dello stesso, il PID nella pids_map potrebbe essere riutilizzato dal sistema operativo per un processo diverso dal precedente.

3.5.4 Handlers

Definire gli handlers in eBPF

Il Filter implementa un handler per syscall generiche e degli handler per alcune syscall che hanno un ruolo nella creazione e terminazione di processi. Come verrà mostrato successivamente, alcuni meccanismi della creazione di processi in ambiente Unix vengono sfruttati per gli scopi del Filter.

Secondo la documentazione di eBPF gli handler per i probes associati all'entry point delle syscall (Jprobes) devono avere un prototipo che rispetti un formato specifico [6].

```
syscall__SYSCALLNAME(struct pt_regs *ctx, [, argument1 ...])
```

SYSCALLNAME è il nome della syscall di cui tenere traccia, e *pt_regs* è un puntatore ad una struttura usata dal Kernel per salvare i registri della CPU nello spazio di memoria del processo quando viene effettuata una syscall. Gli argomenti *[argument1 ...]* possono essere omessi se l'handler non ha intenzione di accedere agli argomenti con cui la syscall è stata invocata, altrimenti devono corrispondere a quelli del prototipo della syscall. Per esempio, per definire un Jprobe per *execve* occorre usare:

```
int syscall__execve(struct pt_regs *ctx,
    const char __user *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
```

```
{
    [...] \\Codice dell'handler
}
```

Il formato con cui vanno dichiarati gli handler per i Kretprobes invece è il seguente: [6].

```
kretprobe__kernel_function_name(struct pt_regs *ctx)
```

kernel_function_name è il nome della funzione di cui si vuole osservare la terminazione. Il valore di ritorno può essere ottenuto tramite la macro helper di eBPF *PT_REGS_RC*, con argomento il puntatore alla struttura dove il Kernel ha copiato i registri CPU. Per esempio:

```
int kretprobe__tcp_v4_connect(struct pt_regs *ctx)
{
    int ret = PT_REGS_RC(ctx);
    [...] \\ Ulteriore codice dell'handler
}
```

La variabile *ret* conterrà il valore di ritorno della *tcp_v4_connect.*

Handler generico

Come già accennato, le syscall capaci di generare nuovi processi (*execve(2), execveat(2))* sono trattate diversamente. Tutte le altre syscall hanno installato un Kprobe sul proprio entry point (Jprobe) e relativi handlers, che rimandano però allo stesso codice, e ricevono lo stesso trattamento da parte del Filter. Il Filter fa uso della seguente macro per definire correttamente i prototipi degli handler Jprobe secondo la sintassi eBPF:

```
#define TRACE_SYSCALL(name, id)                          \
int syscall__##name(struct pt_regs *ctx)                 \
{                                                        \
    trace_generic(ctx, id);                              \
    return 0;                                            \
}
```

La variabile *id* segue un formato condiviso tra Filter e Classifier che associa ad ogni syscall un identificatore numerico.

Handler Kretprobe

Come già accennato, gli unici Kretprobe usati sono installati sulle syscall responsabili della creazione di processi figli (*fork(2), vfork(2), clone(2)*. Il Filter fa uso della seguente macro per defnire correttamente i prototipi degli handler Kretprob secondo la sintassi eBPF:

```
#define TRACE_RET_FORK(name, id)                    \
int trace_ret_##name(struct pt_regs *ctx)           \
{                                                   \
    trace_ret_fork_generic(ctx);                    \
    return 0;                                       \
}
```

3.5.5 Formato degli eventi

Il Filter inizializza un'apposita struct per aggregare i dati interessanti per un data syscall. Le istanze di questa struct vengono inviate in spazio utente tramite il ring buffer e la funzione ebpf *perf submit*[6].

```
typedef struct {
    u32 real_pid;
    u32 ns_pid;
    u32 ns_id;
    u32 mnt_id;
    u64 ts;
    char comm[TASK_COMM_LEN];
    char uts_name[TASK_COMM_LEN];
    enum syscall_id call;
    char path[128];
} data_t;
```

3.6 Il Classifier

3.6.1 Funzionalità

Il Classifier si occupa di effettuare le operazioni necessarie a caricare il Filter come programma eBPF, installare tutti i probe, inizializzare il ring buffer e registrare le funzioni da eseguire all'arrivo di un evento sul buffer. Il Classifier fa uso di strutture dati apposite per la rappresentare una istanza dell'algoritmo BoSC già presentato.

Le bags di syscall sono rappresentate come tuple. L'indice dell'elemento della tupla corrisponde al codice id della syscall, il valore alla frequenza della syscall all'interno della bag. I dati sul comportamento normale sono rappresentati come un dizionario Python che ha per chiavi le tuple rappresentanti le bag e per valori

Una volta in ascolto, il Classifier riceve gli eventi dal Filter e li salva nelle strutture dati e applica l'algoritmo BoSC. Il Classifier si occupa anche della scrittura e lettura su file dei dati sul comportamento normale appreso per un dato container o processo.

3.6.2 Comunicazione con il Filter

Maps

Come già detto, il Classifier comunica dei parametri al Filter tramite l'accesso a delle map eBPF condivise. Le funzioni primitive offerte per interfacciarsi alle map eBPF sono le seguenti:

```
void bpf_map_lookup_elem(map, void *key. ...);
void bpf_map_update_elem(map, void *key, ..., __u64 flags);
void bpf_map_delete_elem(map, void *key);
```

La libreria Python del framework BCC astrae il programma eBPF ad un oggetto di tipo *BPF* e fornisce una serie di funzioni per interagire con esso, evitando così l'uso diretto della verbosa syscall *bpf* [9]. BCC offre anche funzioni per accedere facilmente alle maps e anche una sintassi succinta alternativa applicata ad un oggetto di tipo BPF. Per accedere al valore in una map si può usare la seguente sintassi *bpf_item["map_name"][key]*, con *bpf_item* l'oggetto che rappresenta il programma eBPF, *map_name* il nome della map per come è stata inizializzata nel programma eBPF e *key* la chiave a cui è stato associato il valore.

Nel caso di scrittura in una map da parte di un programma in user space che utilizzi BCC, è necessario però fare uso delle funzioni offerte dalla libreria Python *ctypes* per convertire i valori in una rappresentazione C-compatible, perché in generale i tipi delle variabili e gli standard di rappresentazione non coincidono[12]. Ad esempio, per scrivere il valore *false* del tipo booleano Python in una map, occorre dichiarare:

```
bpf["config_map"][key] = ctypes.c_uint32(False)
```

Ring Buffer

Oltre all'accesso alle map, BCC fornisce meccanismi per la lettura degli eventi inviati tramite ring buffer da parte del programma eBPF. Una volta inizializzato l'oggetto di tipo bpf in Python, è possibile aprire l'estremità nello user space di un ring buffer associato al programma eBPF e stabilire una funzione *callback* che verrà invocata ogni volta che un evento sarà presentato allo user space tramite il buffer. La sintassi è la seguente:

```
table.open_perf_buffers(callback, page_cnt=N, lost_cb=None)
```

table è una map di un oggetto bpf, *callback* il nome della funzione callback per l'arrivo di un evento sul buffer, *page_cnt* la dimensione del ring buffer e *lost_cb* il nome di una seconda callback opzionale per la gestione degli eventi persi. Infatti una dimensione insufficiente del ring buffer unita ad una generazione degli eventi troppo rapida da parte del programma eBPF può portare al riempimento del buffer, con conseguente perdita di successivi eventi.

3.6.3 Implementazione dell'algoritmo

Il Classifier definisce una classe Python *BagManager* che rappresenta un'istanza dell'algoritmo BoSC ed è dotata di tutti gli attributi e metodi appropriati ad eseguire l'algoritmo e a mantenerne lo stato.

Learning

Dopo aver inizializzato il programma eBPF, il Classifier aggiunge gli eventi alla trace di un oggetto BagManager. Dopo che l'utente decide di terminare la fase di learning, il Classifier scorre la trace

Una volta la procedura di learning è stata ultimata, i dati per quello specifico container o processo sono salvati su un file in formato *json* e il programma termina stampando informazioni sulla durata della procedura di learning.

Monitoring

Dopo aver inizializzato il programma eBPF, il Classifier legge (se presente) il file json appropriato che descrive il comportamento normale del processo o container che si vuole monitorare, e utilizza le informazioni per inizializzare un oggetto di tipo BagManager. Gli eventi in arrivo sul buffer sono utilizzati per creare delle bag in tempo reale e confrontarle con i dati caricati secondo quanto visto durante la presentazione dell'algoritmo BoSC. Durante il monitoring il programma stampa informazioni sul numero di eventi, anomalie rilevate ed eventuali eventi persi.

Capitolo 4

Validazione e Risultati

In questo capitolo verranno presentati i risultati finali e la validazione relativa allo strumento sviluppato. La validazione è un'attività di controllo mirata a confrontare il risultato di una fase del processo di sviluppo con i requisiti del prodotto – tipicamente con quanto stabilito dal contratto o, meglio, dal documento di analisi dei requisiti [47]. Un comune esempio di validazione è il controllo che il prodotto finito abbia funzionalità e prestazioni conformi con quelle stabilite all'inizio del processo di sviluppo.

Possiamo dividere i test di validazione in due categorie. I cosiddetti *Test di unità (o dei moduli)* prendono in considerazione il più piccolo elemento software previsto dall'architettura del sistema: il modulo. Il controllo su un modulo ha come obiettivo principale la correttezza funzionale delle operazioni fornite dal modulo. Invece i *Test di sistema* valutano ogni caratteristica di qualità del prodotto software nella sua completezza, avendo come documento di riscontro i requisiti inizialmente forniti.

4.1 Test di unità

Per quanto riguarda i test di unità, l'architettura eBPF rende difficile la scomposizione di programmi in moduli separati, per cui è stata fatta la scelta di testare il programma eBPF Filter nella sua interezza assieme alle funzionalità del programma Classifier che gestiscono l'inizializzazione, il caricamento del Filter e la ricezione degli eventi dal Kernel-space. Questi due elementi sono stati considerati come un unico modulo. Le restanti operazioni fornite dal Clas-

sifier e identificate dalla classe Python *BagManager* costituiscono un secondo modulo che è stato testato in maniera separata prima di procedere ai test di sistema.

4.1.1 Modulo eBPF + Classifier

L'obiettivo del test è verificare la correttezza funzionale della generazione e riconoscimento degli eventi per un dato processo. Il modulo è stato testato usando come *driver* del test un programma elementare che esegue un insieme conosciuto a priori di syscall, sia per tipo che per quantità. Il Classifier è stato inizializzato per il monitoring del processo driver. Lo *stub* del test è rappresentato dalla stampa a schermo della lista degli eventi che il Classifier raccoglie. Il confronto tra la lista di syscall codificate nel programma driver e quelle effettivamente stampate a schermo dal Classifier è stato usato per verificare la correttezza.

Lo stesso approccio è stato utilizzato per testare il corretto riconoscimento dei container e filtraggio degli eventi. Due diversi programmi driver sono stati eseguiti in due container separati, mentre il Classifier è stato invocato per monitorare uno solo di essi. Gli eventi stampati a schermo fanno riferimento solo all'insieme di syscall del programma driver eseguito nel container corretto. Il modulo costituito dal Filter più la parte di inizializzazione e raccolta eventi del Classifier è stato così testato con successo.

4.1.2 Modulo BagManager

Il secondo modulo è stato testato inizialmente usando come *driver* del test una semplice *trace* di eventi creata artificialmente detta *trace_1* e data in input ad un'istanza della classe BagManager per la procedura di training. Il BagManager ha creato un database BoSC di comportamento normale, che viene memorizzato come file in formato *json*. Il database BoSC è stato esaminato e rispecchia il contenuto della *trace_1*. Il database viene poi letto dal file e usato per inizializzare una seconda istanza BagManager, che prende in input la *trace_1* ed esegue la parte di monitoring dell'algoritmo ”in tempo reale”. Lo *stub* del test è rappresentato dalla stampa a schermo delle anomalie. Si osserva che la procedura di monitoring come atteso non genera nessuna anomalia, in quanto la stessa *trace_1* è stata usata sia per il learning che per il monitoring. Successivamente è stata creta una seconda trace artificiale detta *trace_2* molto diversa

dalla prima, che è stata usata come input per il monitoring di una istanza BagManager, sempre inizializzata con il comportamento normale relativo alla *trace_1*. In questo secondo caso vengono rilevate il numero massimo di anomalie data la lunghezza della trace monitorata.

4.2 Test di Sistema

4.2.1 Requisiti

Per i test di Sistema il riscontro è dato dal rispetto dei seguenti requisiti:

- Lo strumento è di facile utilizzo da parte di un utente competente nell'uso di altri strumenti tramite riga di comando (*usabilità*)
- Successo nel determinare il profilo di comportamento normale di un processo o container in fase di learning, e successivamente a rilevare eventuali anomalie in fase di monitoring (*funzionalità*).
- Lo strumento impiega un lasso di tempo ragionevole nell'elaborare il database di comportamento normale per un processo o container (*perfomance*).
- Lo strumento occupa poco spazio sulla memoria principale una volta in esecuzione, e il databse di comportamento di un processo o container normale occupa poco spazio sulla memoria secondaria (*storage use*).
- L'accesso a funzionalità che dovrebbero essere riservate è impedito da meccanismi di sicurezza. Essendo l'attività di monitoring dell'host solitamente riservata all'amministratore di sistema, l'esecuzione dello strumento deve richiedere privilegi da *superuser* (*security*).
- Nel caso vengano generati molti eventi in poco tempo, le fasi di learning e di monitoring non devono perdere dati (*load, stress*).

4.2.2 Usability Test

Lo strumento realizzato implementa l'opzione *–help* per la stampa di una pagina di aiuto all'utilizzo che rispetta lo standard POSIX[23]. L'usabilità dello strumento è stata verificata mediante un gruppo di 5 utenti. Ai partecipanti (studenti di Informatica dell'Università di Pisa) è stato somministrato il seguente test:

```
Leggi il file Readme del tool (include la pagina di help) al seguente link:
https://github.com/andreabonanno/eBPFtest/blob/master/README.md

Che comando inseriresti per:
1) Imparare il comportamento del container e90b8831a4b8?
2) Monitorare il comportamento del container e90b8831a4b8?
3) Imparare il comportamento dell'applicazione firefox?
4) Monitorare il comportamento dell'applicazione firefox?
```

I comandi sono stati formulati correttamente nel 90% dei casi (18 comandi corretti su 20). Lo strumento quindi rispecchia i requisiti di usabilità.

4.2.3 Facility Test

Definizioni

I termini *precision* e *recall* (precisione e recupero) assumono un significato specifico in alcuni ambiti, tra cui quello della classificazione statistica [76]. La precision per una classe è il numero di veri positivi (il numero di oggetti etichettati correttamente come appartenenti alla classe) diviso il numero totale di elementi etichettati come appartenenti alla classe (la somma di veri positivi e falsi positivi, che sono oggetti etichettati erroneamente come appartenenti alla classe). La precision è conosciuta anche come *TPR* (*True Positive Rate*). Il termine recall in questo contesto è definito come il numero di veri positivi diviso il numero totale di elementi che effettivamente appartengono alla classe (per esempio la somma di veri positivi e falsi negativi, che sono oggetti che non sono stati etichettati come appartenenti alla classe ma dovrebbero esserlo). Un'altra misura interessante è data *TNR*(*True Negative Rate*), conosciuto anche come

Sensitivity.

$$Precision = \frac{vero_positivo}{vero_positivo + falso_positivo}$$

$$Recall = \frac{vero_positivo}{vero_positivo + falso_negativo}$$

$$TrueNegativeRate = \frac{vero_negativo}{vero_negativo + falso_positivo}$$

Nel contesto della classificazione delle anomalie da parte dello strumento realizzato si è fa riferimento alle misure sopracitate. Le trace generate dai tool utilizzati per il learning sono stati considerate come capaci di causare solo elementi negativi e quelle generate dai tool usati per simulare il comportamento anomalo come capaci di causare esclusivamente da elementi positivi.

Modalità di test

Esiste un consistente numero di studi sui sistemi di rilevamento rivolti ai processi, ma un numero molto inferiore di essi è rivolto al rilevamento delle anomalie a livello dei container. Anche nel caso di studi riguardanti i container nello specifico, spesso non vengono forniti abbastanza dati espliciti sulla trace per riprodurre le metodologie di test. Un esempio è lo studio di KubAnomaly[67], che fornisce dataset contenenti dati aggregati sulla trace delle syscall e degli altri eventi, rendendo difficile applicare lo stesso metodo di test allo strumento in esame. Inoltre, nel caso di studi sul rilevamento delle anomalie basata su syscall, i *datasets* che rappresentano trace di comportamento normale e soprattutto anomalo sono pochi, spesso datati e a volte non contengono eventi *labeled* come normali o anomali. Il problema della mancanza di dataset di riferimento variegati e aggiornati per il *benchmarking* di sistemi di rilevamento di anomalie è stato sollevato in un'analisi di Maggi et al.[55]. Il dataset ai quali si fa più spesso riferimento [35, 46, 55] sono quello della Universiy of New Mexico (UNM)[61], che raccoglie le trace "sintetiche" (generate tramite script) e "live" (ottenute dall'utilizzo da parte di un utente) di diversi programmi. L'altro principale dataset di riferimento è il *1999 DARPA Intrusion Detection Evaluation Dataset*[57], che però è considerato datato.

Per raccogliere i dati relativi all'uso dello strumento nel caso dei container è stato usata una metodologia di test adattata su quella già utilizzata in Abed et al.[24]. Il container testato è stato creato tramite l'immagine Docker ufficiale per *MariaDb*, uno dei database relazionali più

utilizzati[33]. Seguendo la strategia di test vista nello studio di riferimento, il comportamento normale durante il processo di learning è stato generato utilizzando il tool di benchmarking *mysqlslap*[39]. Una trace di comportamento anomalo è stata generata tramite l'utilizzo del tool *metasploit*[73] tramite un attacco di tipo brute-force alle credenziali di accesso database.

Un secondo tipo test è stato effettuato utilizzando i dataset UNM, nello specifico quelli relativi all'applicazione *sendmail*.

L'ambiente dei test consiste in una macchina virtuale Ubuntu Linux 19.10 virtualizzata via KVM. L'host dell'hypervisor assegna alla macchina virtuale 6 core della propria CPU (Xeon E5-2687W v3, 10 core, 3.10-3.50GHz).

Test Container su caso limite anomalo

Per i test di funzionalità si è inizialmente osservato lo strumento sotto alcuni casi limite, usando le stesse metodologie di test applicate da Abed et al.[24].

Inizialmente è stato effettuato il processo di learning simulando il comportamento normale tramite il tool msqlslap. In eguito il test è stato effettuato esponendo il Classifier ad una trace ”completamente anomala”, generata tramite il tool Metasploit nei confronti del container MariaDB. Il risultato atteso è la generazione del numero massimo di anomalie per una data dimensione della trace, che come è stato detto in 3.4.2 è approssimabile a $\frac{d_{trace}}{tresh}$, con d_{trace} dimensione della trace e $tresh$ limite di mismatch per epoch dopo il quale segnalare l'anomalia. I risultati dei test rispecchiano l'aspettativa e vedono il Classifier generare il numero di anomalie massimo per una data lunghezza della trace. I test è stato ripetuto per diverse lunghezze della trace ($2 \times 10^4, 3 \times 10^4, 4 \times 10^4$), dando sempre gli stessi risultati. Il test sul caso limite con traccia completamente anomala porta risultati in linea con i test effettuati da Abed et al.[24], ovvero Precision e Recall massime in fase di monitoring. I risultati sono riportati nella tabella 4.1.

Test Container su caso limite benigno

Il test precedente che imita quello effettuato da Abed et al.[24] è stato replicato utilizzando in fase di monitoring il tool di benchmark msqlslap, lo stesso utilizzato per il learning, e quindi inviando al container solo richieste ”benigne”. Essendo lo stesso tipo di richieste utilizzate in

Parametri Classifier	
Sliding Window Size	10
Epoch Size	5000
Mismatch Treshold	500

VP = Veri Positivi, VN = Veri Negativi, FP = Falsi Positivi, FN = Falsi Negativi

Test su caso limite anomalo: MariaDB container			
Lunghezza Trace	20082	30013	40093
Attacchi effettuati	40	60	80
Anomalie Segnalate (VP)	40	60	80
Anomalie Assenti (VN)	0	0	0
Precision	1	1	1
Recall	1	1	1

Tabella 4.1: Tabella risultati test con traccia completamente anomala. Non è stato introdotto comportamento benigno, tutte le anomalie vengono sempre rilevate, Precision e Recall sono massime.

fase di learning, il risultato atteso è una completa assenza di anomalie. Durante i test si sono variati in fase di monitoring anche i parametri riguardanti il numero di client concorrenti del tool msqlslap (il learning è stato effettuato con un parametro pari a 30), e il test è stato ripetuto per diverse lunghezze della trace, dando sempre gli stessi risultati. Si fa notare che le misure di Precision e Recall non sono significative in questo caso perchè non sono stati introdotti attacchi da classificare (nello specifico la Precision è sempre uguale a 0 e la Recall non è definita).

Si nota che si ottiene un TNR ottimale finchè il tipo di richieste inviate al container è simile al carico di lavoro con il quale è stato effettuato i learning (30 client concorrenti). Se si ripete il test con un numero aumentato di client concorrenti, le syscall generate possono con

Parametri Classifier	
Sliding Window Size	10
Epoch Size	5000
Mismatch Treshold	500

VP = Veri Positivi, VN = Veri Negativi, FP = Falsi Positivi, FN = Falsi Negativi

Test su caso limite begnigno: MariaDB container			
Lunghezza Trace	20284	20526	20354
msqlap concurrency	50	100	150
Anomalie Segnalate (FP)	0	1	11
Anomalie Assenti (VN)	40	41	40
True Negative Rate	1	0.97	0.78

Tabella 4.2: Tabella risultati test con traccia completamente normale. Non essendo introdotte anomalie, Precision e Recall non sono definite per questo test. I falsi positivi aumentano se il comportamento benigno introdotto diventa complesso e si discosta da quello usato per il learning.

probabilità sempre più crescente intrecciarsi nella trace in maniere non previste dal modello di comportamento normale sviluppato in precedenza. Quello che accade è che si tratta di sequenze ”benigne” ma considerate anomale perchè non hanno un corrispettivo nel modello di comportamento normale e causano un alto numero di mismatch. Queste ipotesi sono rispecchiate nel test, che vede un degrado progressivo del TNR via via che si rende il carico di lavoro potenzilamente diverso da quello che si è usato per la fase di learning. I risultati sono riportati nella tablela 4.2

Test Container su trace mista

Il test precedente sui casi limite che si ispira a quello effettuato da Abed et al.[24] è stato esteso con un nuovo test che fa uso di una trace ”mista” ottenuta eseguendo il tool mysqlslap che genera la trace ”completamente normale” e iniettando attacchi controllati tramite il tool metasploit, che ha il compito di portare attacchi capaci di generare una trace ”completamente anomala”. I tool stessi sono stati analizzati per stabilire il numero di eventi che essi causano per una unità di tempo e dedurre il loro contributo alla trace mista. Gli attacchi sono stati effettuati tramite l’utility metasploit *auxiliary/scanner/mysql/mysql_login* che genera una serie di attacchi di tipo bruteforce alle credenziali di accesso del database. Gli attacchi sono stati discretizzati creando delle raffiche di durata controllata, capaci di causare nella trace del container un numero di eventi pari alla soglia di rilevazione (*mismatch_treshold*). L’effetto desiderato è quindi quello di un’anomalia attesa per ogni attacco portato al container. Gli attacchi sono stati portati in numero controllato e ad intervalli non regolari, in maniera tale da poter ricollegare il comportamento dello strumento di rilevamento al momento in cui è stato portato l’attacco. Un attacco portato e non rilevato come anomalia è considerato un Falso Negativo. Un’anomalia rilevata quando non è stato portato un attacco è considerata un Falso Positivo.

Test su processi

Il Classifier è stato adattato per eseguire le procedure di learning e monitoring prendendo come input una trace di syscall tramite file di testo e non tramite un tracing framework. Si tratta quindi di un test non ”live”. La trace relativa all’applicazione *sendmail* nel dataset UNM[61] è stata usata per il learning. I parametri del Classifier sono gli stessi del test precedente (window size 10, epoch size 5000 e mismatch treshold 500).

La procedura di learning è terminata in 3 epochs (15000 syscalls) generando un file di comportamento normale composto da 484 bags.

Un test per i falsi positivi è stato effettuato dando in input in fase di monitoring la stessa trace utilizzata per il learning. Il Classifier non ha rilevato nessuna anomalia, come atteso.

Per la procedura di monitoring vera e propria sono state usate le trace che simulano un attacco di ”sunsendmailcp intrusion”, ”decode intrusion” e ”syslogd intrusion”. Le tre trace

Parametri Classifier	
Sliding Window Size	10
Epoch Size	5000
Mismatch Treshold	500

VP = Veri Positivi, VN = Veri Negativi, FP = Falsi Positivi, FN = Falsi Negativi

Test trace mista: MariaDB container			
	Test 1	Test 2	Test 3
Trace totale	20689	29748	39711
Attacchi effettuati (VP)	13	20	30
Anomalie segnalate (VP + FP)	16	23	34
Falsi Positivi	3	3	5
Falsi Negativi	0	0	1
Precision	0.81	0.87	0.88
Recall	1	1	0.97

Tabella 4.3: Tabella risultati test su trace mista. Introducendo un numero controllato di anomalie durante il comportamento benigno, si ottiene una Precision media di 0.85 e una Recall media di 0.99.

anomale hanno una dimensione molto ridotta rispetto alla trace usata per il learning (1500 circa rispetto a più di un milione). In ognuno dei tre casi, il Classifier segnala una singola anomalia. In questo test non possono essere fatte assunzioni precise sul numero atteso di anomalie che il Classifier avrebbe dovuto segnalare. Questo perché date le syscall ”anomale” generate da sendmail, i dataset UNM non specificano il loro punto di inizio nella trace ne la loro quantità.

Osservazioni comparative con test di riferimento

Nella scelta delle metodologie dei facility test si è cercato per quanto possibile di emulare test esistenti in studi simili. In particolare, lo strumento realizzato ha ottenuto risultati uguali nel test sui casi limite effettuato da Abed et al.[24]. Il test è stato esteso con una metodologia originale che prevede una trace mista e attacchi controllati. Il test condotto secondo tali modalità si può considerare una simulazione più vicina ad una situazione più realistica di attacco. Questo si riflette nel fatto che i numeri di Precision e Recall raggiunti non sono di assoluta precisione (si ottiene infatti una media di Precision 0.85 e Recall 0.99), ma comunque rappresentano attestano la funzionalità dello strumento.

Il test sui processi utilizza lo stesso dataset adoperato da Kang et al[35]. I risultati sui casi limite che usano il dataset di comportamento normale in fase di monitoring sono ottimali, con un True Negative Rate pari a 1. Il monitoring di trace fornite dallo stesso dataset e contenenti attacchi porta alla rilevazione di un’anomalia. Bisogna tuttavia osservare che il dataset manca di dati aggiuntivi con i quali contestualizzare le trace di attacco secondo i parametri dell’algoritmo BoSC. La mancanza di informazioni più specifiche sul dataset non ha permesso un confronto più diretto con i test di Kang et al[24].

Gli studi di riferimento inoltre non offrono dati interessanti sull’esecuzione dei sistemi di rilevamento delle anomalie, come l’uso della memoria principale e secondaria e i tempi di elaborazione per la classificazione del comportamento normale. Avere accesso ad una descrizione più dettagliata dei processi di test avrebbe permesso di fare un confronto più accurato anche su dati diversi dai valori legati alla rilevazione di anomalie, quali le metriche associate all’uso di risorse da parte dello strumento stesso.

Possibili test ulteriori

I test effettuati hanno avuto lo scopo di emulare quelli dei principali studi di riferimento in materia di rilevamento anomalie. Gli studi di stampo prettamente accademico come già accennato fanno ricorso a dataset datati o comunque rappresentanti situazioni artificiali. Una serie di interessanti test ulteriori può sicuramente riguardare un uso dei container più vicino al mondo reale. Un esempio valido tra i lavori osservati è quello di KubAnomaly[67], che ha effettuato un test su un carico di lavoro realistico, monitorando un container responsabile dell'hosting di un sito web Wordpress. Il sito è stato volontariamente reso vulnerabile ad attacchi di tipo SQL injection, command injection e zap attack ed è stato reso liberamente accessibile al pubblico. Il web server è stato monitorato con KubAnomaly per una durata di un mese, dopodichè sono stati raccolti ed elaborati i dati sulle anomalie collegate agli attacchi ricevuti.

Il vantaggio di questo approccio è quello di poter confrontarsi con attacchi reali, ma resta il probelma di fornire al sistema di rilevamento anomalie un dataset corretto ed esaustivo del comportamento normale del container. Inoltre, la raccolta di dati significativi può richiedere anche molto tempo.

Lo strumento realizzato si presta certamente a questo tipo di test di carattere più generale, ma essendo un sistema semisupervised, la procedura di learning deve esser effettuata con un dataset esaustivo e soprattutto ”pulito”, cioè privo di attacchi. La difficoltà principale a portare avanti questo tipo di test è realizzare un ambiente che simuli (preferibilmente tramite uso da parte di utenti reali) l'uso benigno del container, e al tempo stesso garantire che esso non sia vittima di attacchi per tutta la durata del processo di learning.

4.2.4 Performance Test

Il test sulla performance riguarda il tempo necessario al Classifier per creare il file di comportamento normale per una data trace. Si noti che il tempo di elaborazione non dipende dal numero complessivo di epoch della trace (dimensione della trace completa), ma dal numero di epoch necessario a raggiungere un equilibrio nel database BoSC, come mostrato in 3.4.2. Una raggiunta la condizione per la qual il Classifer ritiene di aver appreso il comportamento normale, il resto della trace non viene elaborato.

La tabella sottostante fa riferimento ad alcuni tempi tipici di elaborazione rispetto al numero di epoch (e alla loro dimensione) necessarie a completare il processo di learning con successo.

Performance Test: MariaDB container				
	Test 1	Test 2	Test 3	Test 4
Tempo di elaborazione (s)	1.2	1.5	0.37	0.52
Numero di epoch	3	3	3	3
Dimensione epoch	5000	5000	10000	10000

4.2.5 Storage use Test

L'utilizzo dello spazio in memoria principale del Classifier è stato misurato durante l'esecuzione dei test precedenti tramite l'uso del modulo Python *memory_profiler*.

per quanto riguarda lo spazio utilizzato in memoria secondaria, è stato misurato lo spazio occupato dai file che descrivono il comportamento normale.

Storage Test: MariaDB container					
	Test 1	Test 2	Test 3	Test 4	Test 5
Memoria principale (MiB)	113	114	114	112	111
Memoria secondaria (KB)	340	335	349	323	340
Numero di Bag nel database	1141	1124	1173	1084	1142

4.2.6 Security Test

Ogni interazione da parte dell'utente con l'architettura eBPF avviene attraverso la syscall *bpf(2)*, che fallisce con codice *EPERM* se l'utente non ha gli opportuni privilegi *SYS_CAP_ADMIN*[10]. BPF prevede una serie di parametri *tunables* che riguardano principalmente il JIT (Just In Time compiler) e che hanno una serie di implicazioni sulla sicurezza. Una trattazione più specifica è presente al Capitolo 11 di "BPF Perfomance Tools" [49]. Considerando lo scopo dello strumento, il requisito di un utente privilegiato permette allo strumento nel suo complesso di rispettare i requisiti di sicurezza.

4.2.7 Load Test

Durante la fase di sviluppo dello strumento è stata osservata una perdita di eventi durante le fasi di avvio dei processi e containers monitorati. Una successiva fase di ricerca ha portato a formulare l'ipotesi che il Classifier non riuscisse a consumare gli eventi abbastanza velocemente rispetto a quelli generati dal Filter, con conseguente riempimento del ring buffer e perdita di eventi. La dimensione default del ring buffer usato da BCC è di 8 eventi. Aumentando tramite gli opportuni parametri la dimensione del buffer a 512, non si sono osservate ulteriori perdite di eventi durante test.

Capitolo 5

Conclusioni

Il recente aumento dell'uso dei container in ambienti di produzione porta con se la necessità di creare soluzioni opportune per la visibilità e la sicurezza. Le qualità desiderabili di queste soluzioni sono un basso overhead, che comporta la capacità di gestire un grande numero di eventi, e la facilità di deployment e utilizzo. Inoltre gli strumenti stessi non devono interferire in maniera significativa con le attività dell'host, né aprire nuove problematiche sulla sicurezza aumentando la superficie di attacco del sistema.

In questo elaborato sono state valutate diverse opzioni e approcci esistenti al problema, e si è mostrato come il Kernel Linux sia perfettamente in grado di fornire gli strumenti necessari alla propria introspezione, in questo caso rivolta ai container. Nel corso degli anni infatti il Kernel ha sviluppato un variegato ecosistema di fonti di dati statici e dinamici e tracing framework per la propria introspezione. Dopo una fase di ricerca basata sugli obiettivi di una resa ottimale di perfomance e versatilità, si è identificato nella tecnologia eBPF il candidato ideale per lo scopo. L'uso della tecnologia eBPF è un trend in crescita ed è considerata una delle più promettenti; inoltre, le sue potenzialità non sono ancora state sfruttate appieno. La tecnologia eBPF è completamente inclusa nel Kernel e non richiede la modifica dello stesso o l'aggiunta di moduli. La gamma di possibilità offerte da eBPF e la sua versatilità sono tali che è stato possibile adattarla al monitoring dei containers tramite l'uso di un compatto programma ad hoc. Attività che in precedenza richiedevano lo sviluppo di un Kernel personalizzato ora possono essere ottenute, in modo più efficiente, con programmi eBPF, entro i limiti di sicurezza della sua macchina virtuale.

La validità di eBPF è avvalorata dalla presenza di framework come BCC, che semplificano e rendono più lineare il workflow della scrittura di programmi per la macchina virtuale e interazione con la stessa.

Una volta ottenuta l'osservabilità sui container, è stata condotta una ricerca su quelle che sono le strategie più promettenti per la rilevazione delle anomalie in tempo reale. Si è optato per un modello basato su apprendimento semisupervised e non su policy definite a priori per il comportamento atteso dei container. Questo ha permesso di realizzare uno strumento capace di modellare il comportamento normale di un container con intervento minimo da parte dell'utente e senza necessità di una configurazione esplicita. L'algoritmo realizzato si ispira a lavori precedenti rivolti principalmente al monitoring di singoli processi e si basa sull'analisi delle frequenze con le quali diverse syscall si presentano nella trace del container. Il risultato è uno strumento a bassissimo overhead capace di confrontare in tempo reale i nuovi eventi riguardanti un container con un modello di comportamento normale già modellato, e di segnalare eventuali anomalie. I test eseguiti mostrano una precisione nel rilevamento delle anomalie prossima al 90% nel caso di attacchi a container inseriti all'interno di un flusso di esecuzione normale.

5.1 Lavori futuri

Una problematica riscontrata nello sviluppo di questa tesi è la scarsa reperibilità di dataset di riferimento con i quali effettuare testing e validazione dello strumento. Diversi lavori nello stesso ambito hanno segnalato e discusso la questione, facendo riferimento a dataset datati e spesso creati appositamente per rispecchiare situazioni lontane dal comportamento normale di un processo o container. Anche gli esempi di attacchi e comportamenti anomali nei dataset disponibili sono stati criticati come irrealistici e non sempre aderenti a quella che può essere una reale situazione di azione malevola nei confronti di un ambiente di produzione.

Una seconda problematica è stata riscontrata sempre in fase di test, nello specifico riguardante gli studi di riferimento nello stesso ambito che non fanno uso dei dataset più comuni, ma invece creano modalità di test specifiche. Questi studi offrono pochi dettagli sull'ambiente di test e soprattutto sulle modalità con cui sono stati generati i comportamenti normali e le

anomalie nei container o processi. Questa problematica ha reso difficile riuscire a effettuare un confronto diretto con quelli che sono i risultati di altri studi.

Questi temi evidenziano come tale mancanza costituisce possibilità di approfondimento per lavori futuri incentrati più sulla creazione di dataset affidabili e metodologie di test di facile condivisione e riproducibilità.

Per quanto riguarda lo strumento realizzato, il suo contributo principale è l'implementazione concreta di algoritmi già proposti per la rilevazione di anomalie, con l'aggiunta della capacità di monitoring in tempo reale, e l'integrazione con meccanismi ad alta performance quale eBPF. Ulteriori ambiti di sviluppo possono riguardare l'integrazione di sistemi policy-based con il modello di apprendimento automatico. Inoltre, come già osservato negli studi di riferimento, i modelli basati su analisi delle frequenze di syscall fanno uso di alcuni parametri di configurazione (ad esempio la soglia di mismatch dopo la quale segnalare l'anomalia) che sono selezionati principalmente in maniera empirica. Tali studi hanno hanno indagato solo parzialmente gli effetti che la scelta di tali parametri ha sulla precisione del modello e sulla sua *responsivness* dello stesso al verificarsi delle anomalie. Possibili aree per ulteriore sviluppo potrebbero riguardare l'indagine di questi effetti e tentativi di formalizzare metodi per ottimizzare la scelta di tali parametri.

Appendice A

Codice Sorgente

Il codice sorgente del Filter e del Classifier, assieme ad un file readme che illustra l'uso dello strumento, e di publico accesso e può essere trovato al seguente link:
https://github.com/andreabonanno/eBPFtest

Bibliografia

[1] Apparmor. ArchWiki. https://wiki.archlinux.org/index.php/AppArmor.

[2] Aqua. AquaSec. https://www.aquasec.com/.

[3] Bpf and xdp reference guide. https://docs.cilium.io/en/v1.7/bpf/.

[4] Bpf compiler collection (bcc): Installing bcc. GitHub. https://github.com/iovisor/bcc/blob/master/INSTALL.md.

[5] Bpf compiler collection (bcc): Kernel configuration. GitHub. https://github.com/iovisor/bcc/blob/master/INSTALL.md#kernel-configuration.

[6] Bpf compiler collection (bcc): Reference guide. GitHub. https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md.

[7] Bpf features by linux kernel version. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#bpf-features-by-linux-kernel-version.

[8] *"BPF-HELPERS - list of eBPF helper functions"*. "http://man7.org/linux/man-pages/man7/bpf-helpers.7.html".

[9] *"BPF(2) Linux Programmer's Manual "*. "http://man7.org/linux/man-pages/man2/bpf.2.html".

[10] *"CAPABILITIES(7) Linux Programmer's Manual"*. "https://www.man7.org/linux/man-pages/man7/capabilities.7.html".

[11] *"CGROUPS(7) Linux Programmer's Manual"*. "http://man7.org/linux/man-pages/man7/cgroups.7.html".

[12] ctypes — a foreign function library for python. python.org. https://docs.python.org/3/library/ctypes.html#fundamental-data-types.

[13] *"Docker Docs: Docker run reference"*. "https://docs.docker.com/engine/reference/run/#name—name".

[14] Falco. Sysdig Falco. https://falco.org/.

[15] ftrace - function tracer. kernel.org. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[16] Kernelshark. kernelshark.org. https://kernelshark.org/Documentation.html.

[17] Linux socket filtering aka berkeley packet filter (bpf). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[18] linux/include/linux/nsproxy.h. https://github.com/torvalds/linux/blob/master/include/linux/nsproxy.h#L31.

[19] linux/include/uapi/linux/bpf.h. https://github.com/torvalds/linux/blob/v4.20/include/uapi/linux/bpf.h#L45.

[20] *"NAMESPACES(7) Linux Programmer's Manual"*. "http://man7.org/linux/man-pages/man7/namespaces.7.html".

[21] Perf_event_open(2) linux programmer's manual. man7.org. https://www.man7.org/linux/man-pages/man2/perf_event_open.2.html.

[22] Twistlock. Twistlock. https://www.twistlock.com/.

[23] *"Utility Conventions - Utility Argument Syntax"*. "https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html#tag_12_01".

[24] A. S. Abed, T. C. Clancy, and D. S. Levy. Applying bag of system calls for anomalous behavior detection of applications in linux containers. In *2015 IEEE Globecom Workshops (GC Wkshps)*, 2015.

[25] S. S. Alarifi and S. D. Wolthusen. Detecting anomalies in iaas environments through virtual machine host system call analysis. In *2012 International Conference for Internet Technology and Secured Transactions*, pages 211–218, 2012.

[26] Suaad Alarifi and Stephen Wolthusen. Anomaly detection for ephemeral cloud iaas virtual machines. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, pages 321–335, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[27] Anup Buchke Aman Singh. Performance monitoring unit. rts.lab.asu.edu. http://rts.lab.asu.edu/web_438/project_final/Talk 9 Performance Monitoring Unit.pdf.

[28] Dean Berris. Catapult. GitHub. https://github.com/catapult-project/catapult.

[29] Gianluca Borello. Sysdig and falco now powered by ebpf. Sysdig Blog, 2 2019. https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/.

[30] Eric Carter. 2018 docker usage report. Sysdig Blog, 5 2018. https://sysdig.com/blog/2018-docker-usage-report/.

[31] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), July 2009.

[32] Michael Cherny. Blackhat 2017: Multi-stage attack targeting container developers. AquaSec Blog. https://blog.aquasec.com/host-rebinding-and-shadow-containers-at-blackhat-2017.

[33] Docker Community. Docker official images - mariadb. DockerHub. https://hub.docker.com/_/mariadb.

[34] Jonathan Corbet. Namespaces in operation, part 5: User namespaces. LWN.net, 3 2018. https://lwn.net/Articles/750313/.

[35] D. Fuller D. Kang and V. Honavar. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In *Sixth Annual IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop. IEEE*, pages 1039–1044 vol.2, 2005.

[36] D. Dasgupta and N. S. Majumdar. Anomaly detection in multidimensional data using negative selection algorithm. In *Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, volume 2, pages 1039–1044 vol.2, 2002.

[37] Dipankar Dasgupta and Luis Fernando Nino. A comparison of negative and positive selection algorithms in novel pattern detection. volume 1, pages 125 – 130 vol.1, 02 2000.

[38] Datadog. 8 surprising facts about docker adoption. datadoghq.com. https://www.datadoghq.com/docker-adoption/.

[39] MariaDB Documentation. Mariadb server documentation - mysqlslap. mariadb.com. https://mariadb.com/kb/en/mysqlslap/.

[40] Qingfeng Du, Tiandi Xie, and Yu He. *Anomaly Detection and Diagnosis for Container-Based Microservices with Performance Monitoring: 18th International Conference, ICA3PP 2018, Guangzhou, China, November 15-17, 2018, Proceedings, Part IV*, pages 560–572. 11 2018.

[41] Jake Edge. A look at ftrace. LWN.net, 3 2009. https://lwn.net/Articles/322666/.

[42] Ananth N. Mavinakayanahalli et al. Probing the guts of kprobes. 7 2006. https://landley.net/kdocs/ols/2006/ols2006v2-pages-109-124.pdf.

[43] Jim Keniston et al. Kernel probes (kprobes). The Linux Kernel Archives. www.kernel.org/doc/Documentation/kprobes.txt.

[44] Matt Fleming. A thorough introduction to ebpf. LWN.net, 12 2017. https://lwn.net/Articles/740157/.

[45] Matt Fleming. Using user-space tracepoints with bpf. LWN.net, 5 2018. https://lwn.net/Articles/753601/.

[46] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[47] Carlo Montangero" "Giovanni Cignoni. *"verifica e validazione"*, 2014. "http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/is-a/dispensatestingconv.pdf".

[48] Brendan Gregg. *BPF Performance Tools (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, dec 2019.

[49] Brendan Gregg. *BPF Performance Tools (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, dec 2019.

[50] Martin Grimmer, Martin Röhling, Matthias Kricke, Bogdan Franczyk, and Erhard Rahm. Intrusion detection on system call graphs. 02 2018.

[51] Victoria Hodge. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22:85–126, 10 2004.

[52] Steven Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6, 11 1999.

[53] Michael Kerrisk. Namespaces in operation, part 5: User namespaces. LWN.net, 2 2013. https://lwn.net/Articles/532593/.

[54] Cloud Platform BU Kit Colbert, CTO. Dockercon 2017: Containers go mainstream? VmWare Cloud Native App Blog. https://blogs.vmware.com/cloudnative/2017/04/24/dockercon-2017-containers-go-mainstream/.

[55] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2010.

[56] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[57] Lincoln Laboratory MIT. 1999 darpa intrusion detection evaluation dataset. ll.mit.edu. https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset.

[58] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, and M. Couture. A host-based anomaly detection approach by representing system calls as states of kernel modules. In *2013 IEEE*

24th International Symposium on Software Reliability Engineering (ISSRE), pages 431–440, 2013.

[59] Karen Scarfone Murugiah P. Souppaya, John Morello. Application container security guide. *ACM Comput. Surv.*, 2017.

[60] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, February 2006.

[61] University of New Mexico. Sequence-based intrusion detection. cs.unm.edu. https://www.cs.unm.edu/ immsec/systemcalls.htm.

[62] Keith Ord. Outliers in statistical data : V. Barnett and T. Lewis, 1994, 3rd edition, (John Wiley & Sons, Chichester), 584 pp., [UK pound]55.00, ISBN 0-471-93094-6. *International Journal of Forecasting*, 12(1):175–176, March 1996.

[63] Steven Rostedt. *TRACE-CMD(1)*. https://man7.org/linux/man-pages/man1/trace-cmd.1.html.

[64] Steven Rostedt. Using the trace_event() macro (part 3). lwn.net. https://lwn.net/Articles/383362/.

[65] Steven Rostedt. Ftrace kernel hooks: More than just tracing. blog.linuxplumbersconf.org, 2014. https://blog.linuxplumbersconf.org/2014/ocw/system/presentations/1773/original/ftrace-kernel-hooks-2014.pdf.

[66] Marta Rybczyńska. Bounded loops in bpf for the 5.3 kernel. LWN.net, 7 2019. https://lwn.net/Articles/794934/.

[67] Chin-Wei Tien, Tse-Yung Huang, Chia-Wei Tien, Ting-Chun Huang, and Sy-Yen Kuo. Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches. *Engineering Reports*, 1(5):e12080, 2019.

[68] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on*

Security and Privacy, Proceedings - IEEE Symposium on Security and Privacy, pages 133–145. Institute of Electrical and Electronics Engineers Inc., 1999. 1999 IEEE Symposium on Security and Privacy ; Conference date: 09-05-1999 Through 12-05-1999.

[69] Vince Weaver. The unofficial linux perf events web-page. web.eece.maine.edu. http://web.eece.maine.edu/ vweaver/projects/perf_events/.

[70] Wikipedia contributors. Halting problem — Wikipedia, the free encyclopedia, 2020. [Online; accessed 4-June-2020].

[71] Wikipedia contributors. Keras — Wikipedia, the free encyclopedia, 2020. [Online; accessed 11-June-2020].

[72] Wikipedia contributors. Kubernetes — Wikipedia, the free encyclopedia, 2020. [Online; accessed 11-June-2020].

[73] Wikipedia contributors. Metasploit project — Wikipedia, the free encyclopedia, 2020. [Online; accessed 1-June-2020].

[74] Wikipedia contributors. Microservices — Wikipedia, the free encyclopedia, 2020. [Online; accessed 10-June-2020].

[75] Wikipedia contributors. Os-level virtualization — Wikipedia, the free encyclopedia, 2020. [Online; accessed 14-May-2020].

[76] Wikipedia contributors. Precision and recall — Wikipedia, the free encyclopedia, 2020. [Online; accessed 1-June-2020].

[77] Wikipedia contributors. Scikit-learn — Wikipedia, the free encyclopedia, 2020. [Online; accessed 11-June-2020].

[78] Wikipedia contributors. Tensorflow — Wikipedia, the free encyclopedia, 2020. [Online; accessed 11-June-2020].

[79] Arthur Zimek and Erich Schubert. *Outlier Detection*, pages 1–5. Springer New York, New York, NY, 2017.

[80] Z. Zou, Y. Xie, K. Huang, G. Xu, D. Feng, and D. Long. A docker container anomaly monitoring system based on optimized isolation forest. *IEEE Transactions on Cloud Computing*, 2019.