Alla mia famiglia,
che mi ha sempre sostenuto
in ogni mia scelta.

Introduzione

Questa è l'introduzione.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Lo stato dell'Arte

In questo capitolo si va ad illustrare lo stato dell'Arte delle tecnologie utilizzate. Si illustreranno le principali qualità degli Intrusion Detection Systems e le caratteristiche principali che hanno portato durante i test alla scelta di un software rispetto che un altro. Si passerà poi a presentare sFlow, illustrandone i benefici e le principali differenze con NetFlow e di come esso viene attualmente utilizzato per affiancare un IDS in reti molto estese e complesse. Infine si darà una breve presentazione dello stack ELK, (Elasticsearch-Logstash-Kibana) e di come esso sia utilizzato nell'ambito della Network Security.

1.1 Intrusion Detection System

Un Intrusion Detection System (IDS) [4] è un dispositivo o un' applicazione software che monitora una rete o un sistema per rilevare eventuali attività dannose o violazioni delle policy. Qualsiasi attività o violazione rilevata viene in genere segnalata ad un amministratore o raccolta a livello centrale utilizzando un Security Information and Event Management (SIEM). Un SIEM combina output provenienti da più sorgenti e utilizza tecniche di filtraggio degli allarmi per distinguere le attività dannose dai falsi allarmi.

Esiste un' ampia gamma di IDS, che varia dal software antivirus fino ai sistemi gerarchici che controllano il traffico di un' intera backbone. La classificazione più comune è tra:

- **Network-based Intrusion Detection Firmware (NIDS)**: un sistema che monitora il traffico di rete passante attraverso alcuni punti strategici di una rete. Esempi famosi sono: Suricata [6] , Snort [7] e BRO [8] .

- **Host-based Intrusion Detection (HIDS)** : un software che monitora alcuni file importanti del sistema operativo su cui è installato. Un esempio famoso di HIDS è OSSec [5]

Il panorama degli Intrusion Detection System (IDS) è al giorno d'oggi in continua evoluzione. Tuttavia è possibile operare una seconda e importante classificazione in base a due criteri principali che ne determinano il funzionamento:

- Sistemi con *Signature-based detection*

- Sistemi con *Anomaly-based detection*

Un IDS *Signature-based* analizza i pacchetti passanti su una rete utilizzando il concetto di *signature*: Una signature è un pattern che corrisponde ad un tipo di attacco noto. [7] Esempi di signature possono essere:

- un tentativo di connessione a TELNET con username "root", che corrisponde ad una violazione delle policy di sicurezza

- un email con oggetto "Immagini gratis!" e un allegato con nome "freepics.exe", che sono caratteristici di un attacco noto

- tentativi ripetuti nel tempo di connessione SSH ad intervalli sospetti, che identificano un possibile attacco bruteforce su SSH.

Il rilevamento signature-based è molto efficace nel rilevare minacce note, ma in gran parte inefficace nel rilevare minacce precedentemente sconosciute, minacce mascherate dall'uso di tecniche di evasione e molte varianti di minacce note. Per esempio, se un aggressore ha modificato il malware nell' esempio precedente per usare un nome file di "freepics2.exe", una firma che cercava "freepics.exe" non corrisponderebbe. Il rilevamento basato sulla firma è il metodo di rilevamento più semplice in quanto confronta il campione corrente, come un pacchetto o una voce di registro, con un elenco di firme utilizzando operazioni di confronto tra stringhe.

Un IDS che usa *Anomaly-based* detection, utilizza il concetto di anomalia: ovvero una deviazione del comportamento della rete osservato al momento attuale da quello che è considerato normale in base a quanto osservato in precedenza. Un IDS che utilizza un rilevamento Anomaly-based ha profili che rappresentano il comportamento normale di utenti, host, connessioni di rete o applicazioni. I profili sono sviluppati monitorando le caratteristiche dell'attività tipica per un periodo di tempo. Ad esempio, un profilo di una rete potrebbe indicare che l' attività Web comprende in media il 13% della larghezza di banda della rete al confine Internet durante le normali ore di lavoro giornaliere. L' IDS utilizza quindi metodi statistici per confrontare le caratteristiche dell'attività corrente con le soglie relative al profilo, ad esempio rilevando quando l'attività Web comprende una larghezza di banda significativamente maggiore del previsto e avvisando un amministratore dell'anomalia. L' IDS inoltre potrebbe utilizzare tecniche di intelligenza artificiale per determinare se un comportamento della rete sia da ritenersi normale o anomalo.

Sebbene nell'ultimo periodo l'intelligenza artificiale stia facendo la sua comparsa in ogni ambito dell'informatica gli IDS signature based rappresentano tuttora un'importante fetta (se non la maggioranza) degli IDS in uso nei più importanti data center del mondo ed è per questo che vale la pena studiarli.

In questo elaborato ci si focalizzerà sugli IDS signature based e se ne

analizzeranno le loro prestazioni combinate ad altre tecnologie che verranno introdotte in seguito.

Come anticipato sopra, tra i maggiori esponenti degli IDS attualmente utilizzati abbiamo:

- Snort: Un IDS sviluppato a partire dagli anni '90, acquisito da Cisco nel 2013 e che è tuttora il più utilizzato in ambito enterprise.
- Suricata: Un IDS del nuovo millennio, sviluppato a partire dal 2009 da Open Information Security Foundation (OISF) e che vanta molteplici vantaggi sopra gli altri IDS.

In questo elaborato si è preferito utilizzare per motivi di performance e di implementazione, Suricata. I dettagli di questa scelta saranno chiari più avanti quando saranno state introdotte le principali caratteristiche di Suricata.

1.1.1 Suricata, una breve introduzione

Suricata è un IDS Open Source sviluppato da OISF che fa uso di pattern matching per il riconoscimento di *threat*, violazioni della policy e comportamenti malevoli. Esso è inoltre capace di rilevare numerose anomalie nel protocollo all'interno dei pacchetti ispezionati. Le anomalie rilevabili, tuttavia, sono diverse da quelle degli IDS anomaly-based citati sopra. Le prime infatti sono scostamenti dall'utilizzo lecito di protocolli ben definiti e standardizzati : *Ad esempio: richieste DNS che non sono destinate alla porta 53, oppure richieste HTTP con un header malformato.* Il secondo tipo di anomalia invece è relativo ad un deviazione dal comportamento standard della specifica rete su cui l'IDS è stato tarato, ed è quindi un concetto di anomalia molto più lasco.

Deep Packet Inspection

La rilevazione di queste anomalie in Suricata va sotto il nome di *Deep Packet Inspection* o *Stateful Protocol Analysis*, ovvero il processo di confronto

di determinati comportamenti accettati dal signolo protocollo (HTTP, FTP, SSH, ecc...) con il comportamento osservato al momento del campionamento. Se un IDS usa questa tecnica vuol dire che esso è in grado di comprendere e monitorare lo stato della rete, del trasporto e dei protocolli applicativi che possiedono una nozione di stato. Ad esempio, quando un utente avvia una sessione del File Transfer Protocol (FTP), la sessione si trova inizialmente nello stato non autenticato. Gli utenti non autenticati devono eseguire solo alcuni comandi in questo stato, come la visualizzazione delle informazioni della guida in linea o la fornitura di nomi utente e password. Inoltre una parte importante dello stato di comprensione è l' accoppiamento delle richieste con le risposte, quindi quando si verifica un tentativo di autenticazione FTP, l' IDS può determinare se il tentativo è riuscito trovando il codice di stato nella risposta corrispondente. Una volta che l' utente si è autenticato con successo, la sessione si trova nello stato autenticato e agli utenti è permesso eseguire una qualsiasi delle decine di comandi disponibili. Mentre eseguire la maggior parte di questi comandi mentre sono in stato non autenticato sarebbe considerato sospettoso.

Prestazioni

Una singola istanza di Suricata è capace di ispezionare traffico proveniente da una rete multi-gigabit, questo grazie al forte paradigma multi thread utilizzato nel core del programma. Inoltre esso dispone di un supporto nativo per l'accelerazione hardware, l'analisi di pacchetti con GPUs e supporta nativamente PF_RING [9] e AF_PACKET, due tipologie di socket altamente performanti.

Pattern Matching

Il Pattern Matching è il processo di confronto del pacchetto osservato sulla rete è una signature salvata all'interno di quelle che vengono definite *rules*. Il suo funzionamento può essere riassunto dalla figura 1.1. [2] Ogni pacchetto passante sull'interfaccia di rete monitorata dall'IDS viene quindi

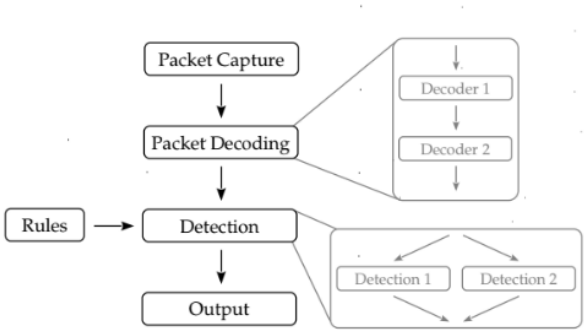

Figura 1.1: Funzionamento del pattern matching

decodificato e poi analizzato parallelamente per riscontrare similitudini con più pattern (rules). Una tipica regola per il suddetto patter matching è nella forma seguente:

```
alert tcp 1.1.1.1 8909 -> 192.168.1.0/24 80
```

Viene quindi indicata l'azione da intraprendere (in questo caso 'alert'), il protocollo , indirizzo/i di sorgente, porta/e sorgente, indirizzo/i di destinazione e porta/e di destinazione del pacchetto con cui fare match. Queste regole possono essere personalizzate oppure è possibile scaricarne di già confezionate da molteplici siti che offrono questo servizio. La praticità dell'utilizzare questa specifica sintassi sta nel fatto che essa è quasi del tutto identica a quella di Snort. Per cui le regole per l'uno o per l'altro software sono intercambiabili.

Altre caratteristiche

Infine una delle caratteristiche fondamentali di Suricata sta nel fatto che esso può funzionare in due modi distinti:

- In modalità online: viene monitorata una interfaccia specifica in modalità *promisqua*, ossia tutti i pacchetti passanti per quella determinata interfaccia vengono decodificati e analizzati.

- In modalita offline: viene monitorato un file pcap contenente del traffico "registrato" in precedenza e che costituisce un punto di riferimento per l'analisi prestazionale delle regole o dell'istanza di Suricata da analizzare.

Concludiamo quindi elencando i motivi che hanno portato durante le fasi di sperimentazione alla scelta di Suricata rispetto ad altri software:

- Velocità (Multi-threading)
- Capacità di analizzare pcap in modalità offline
- Open Source
- Possibilià di analizzare facilmente i log grazie al formato in json
- Il fatto che si tratti di un software "giovane", pensato fin da subito per rispondere alle attuali esigenze di monitoraggio

1.2 sFlow

L' esplosione del traffico internet sta portando a larghezze di banda superiori e una maggiore necessità di reti ad alta velocità. Per analizzare e ottimizzare le reti è necessario un sistema di monitoraggio efficiente. [12]

sFlow [1] è una tecnologia sviluppata dalla InMon Corporation per monitorare il traffico all'interno di grandi reti contenenti switches e routers che utilizza il campionamento di pacchetti. In particolare, esso definisce i meccanismi di campionamento implementati in un *sFlow Agent* e il formato dei dati campionati mandati da tale Agent.

Il monitoraggio si compone di due elementi fondamentali:

- **sFlow Agent**: ovvero un qualsiasi apparato in grado di campionare i pacchetti secondo le specifiche di sFlow e di inviarli ad un *Collector*. l'Agent è un componente molto versatile ed estremamente performante dell'architettura sFlow che può essere impersonato anche da uno switch o da un router, senza degradarne le prestazioni. Il campionamento e la raccolta dei dati del nodo viene fatta in hardware e non presenta overhead nemmeno su reti Gigabit.

- **sFlow Collector**: ovvero una macchina in qualsiasi parte del mondo in grado di raccogliere i dati sFlow e di elaborarli.

L'architettura e le modalità di campionamento usati in sFlow offrono numerosi vantaggi tra cui quello di avere una visione di tutta la rete (*network-wide*) in tempo reale. La visione *network-wide* è possibile poichè sempre più produttori stanno equipaggiando i loro apparati di rete con un modulo sFlow; inoltre recentemente è stato aggiunto il supporto a sFlow anche all'interno di *OpenVSwitch*, un famoso software usato nel campo della *Software Designed Networking*. La visione in tempo reale invece è permessa dal fatto che i pacchetti campionati vengono mandati al collector non appena essi passano per l'agent. Un ulteriore punto a favore sta nel fatto che si tratta di un' architettura estremamente scalabile che permette di posizionare gli agent in

diversi punti della rete, o anche in reti diverse, garantendo una visione totale della rete anche in contesti *Multi-homed*.

1.2.1 Il campionamento

Il campionamento è la parte fondamentale del protocollo sFlow, ed è anche il motivo per cui esso si distacca da altre tecnologie simili come NetFlow.

sFlow usa due modalità operative:

- **Counter Sampling** : un campione dei contatori delle interfacce dell'apparato, che viene schedulato internamente dall' Agent su base temporale (*polling*).
- **Packet Based Sampling**: il campionamento di uno ogni N pacchetti sulla base di un opportuno parametro N (*sampling rate*). Questo tipo di campionamento non permette risultati accurati al 100% ma quantomeno permette di avere risultati di un'accuratezza accettabile e comunque parametrizzabile

Il pacchetto campionato viene esportato verso l'sFlow Collector tramite UDP [11]. La mancanza di affidabilità nel meccanismo di trasporto UDP non influisce in modo significativo sulla precisione delle misurazioni ottenute dall' Agent sFlow, poiché se i Counter Sampling vengono persi, al superamento dell' intervallo di polling successivo verranno inviati nuovi valori. Se invece vengono persi i Packet Flow Sample, questo si riflette in una leggera riduzione della frequenza di campionamento effettiva. L' uso di UDP inoltre riduce la quantità di memoria necessaria per il buffer dei dati. UDP è più robusto di un meccanismo di trasporto affidabile (es. TCP) perchè sotto sovraccarico l' unico effetto sulle prestazioni complessive del sistema è un leggero aumento del ritardo di trasmissione e un maggior numero di pacchetti persi, nessuno dei quali ha un effetto significativo sul sistema di monitoraggio.

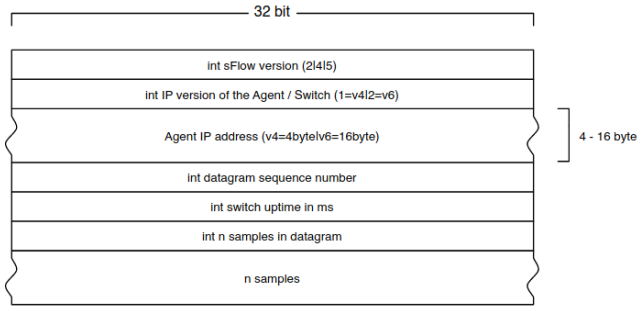

Figura 1.2: Datagram sFlow

Sflow Datagram

Il payload del pacchetto UDP inviato al Collector contiene l' sFlow Datagram, il quale è composto da: versione di sFlow, ip dell'Agent, sequence-number, il numero di campioni contenuti e fino a 10 campioni tra flow samples e counter samples (figura 1.2).

Un *flow sample* consiste di due parti:

- un *packet data*: che contiene l'header del pacchetto campionato (il quale consiste nel pacchetto originale troncato fino ad una lunghezza parametrizzabile)

- un *extended data*: che contiene informazioni aggiuntive, ad esempio in caso di uno switch con VLANs abilitate fornisce la VLAN sorgente e di destinazione.

Come possiamo notare quindi avvengono due tipi di campionamento diversi: un campionamento di un pacchetto ogni N e un troncamento del pacchetto designato fino ad una dimensione massima T (solitamente fissata ad un massimo di 256 Bytes). Occorre notare che il troncamento del pacchetto effettuato ci permette di avere una visibilià dal Layer 2 al Layer 7 dello stack OSI [11]

E‘ necessario puntualizzare che il datagram contenente il *Packet Sample* a differenza del *Counter Sample* viene inviato al collector appena il pacchetto viene campionato o comunque non oltre un secondo più tardi dal campionamento, anche se non si è riempito il buffer di 10 campioni raccolti.

1.2.2 Tools per l'utilizzo di sFlow

La InMon Corporation mette a disposizione due tool fondamentali per l'utilizzo dello standard sFlow, che svolgono i ruoli dell'Agent e del Collector.

hostsflowd

Hostsflowd è un tool per sistemi UNIX [1] che svolge la funzione di Agent, ovvero osserva i pacchetti passanti per una determinata interfaccia e applica ad essi i meccanismi di campionamento discussi sopra. Esso permette di specificare tramite il suo file di configurazione: la frequenza di campionamento (*Sampling Rate*), l'indirizzo del collector e la sua porta, l'intervallo per il polling del *Counter Sample* e la grandezza dell'header del pacchetto campionato.

Si tratta di un tool molto versatile e dai requisiti molto bassi, tuttavia durante i test è emersa una limitazione importante, ossia che non è possibile troncare pacchetti ad una dimensione più grande di 256 Byte, inoltre non è possibile impostare un sampling rate di 1/1 (ovvero campionare tutto). Questo potrebbe non essere una limitazione determinante in produzione ma in fase di testing si è stati costretti a prendere strade alternative, che hanno portato allo sviluppo del simulatore *pcap2sflow*. Ulteriori motivazioni di questa scelta saranno fornite nel capitolo 2.

sflowtool

Sflowtool è un toolkit per decodificare dati sFlow. Esso svolge quindi la funzione di Collector, ed è in grado di stampare in STDOUT i dati decodifi-

[1] Ne esiste una versione anche per Windows

cati o, cosa molto più importante nel contesto di applicazione di questa tesi, spacchettare i campioni e riscriverli in formato pcap.

Ad esempio con il comando seguente è possibile esportare i pacchetti campionati in un file pcap che conterrà i campioni raccolti dall'Agent, troncati secondo i parametri specificati nell'Agent:

```
sflowtool -t > capture.pcap
```

Alternative a sFlow

Ci sono molte tecnologie che a prima vista potrebbero sembrare simili ad sFlow, probabilmente per la presenza della parola "flow", tipo NetFlow, OpenFlow o IPFIX. Tuttavia essi si differenziano da sFlow per alcuni motivi fondamentali: NetFlow e IPFIX sono protocolli di esportazione del flusso che mirano ad aggregare i pacchetti in flussi. Successivamente, i record di flusso vengono inviati a un punto di raccolta per la conservazione e l' analisi. [12] sFlow, tuttavia, non ha alcuna nozione di flussi o aggregazione di pacchetti. Inoltre, mentre l' esportazione del flusso può essere eseguita con campionamento 1:1 (cioè considerando ogni pacchetto), questo non è tipicamente possibile con sFlow, in quanto non è stato progettato per farlo. Il campionamento è parte integrante di sFlow, con l' obiettivo di fornire scalabilità per il monitoraggio in tutta la rete. [13]

La caratteristica fondamentale necessaria per la riuscita del progetto portato avanti in questa tesi stà proprio nel fatto che i dati non siano dati aggregati, come avviene in NetFlow o IPFIX. Un aggregazione dei dati non permette infatti un'analisi degli header dei pacchetti invalidando quindi la funzionalità di monitoraggio contro gli attacchi informatici.

Come ulteriore prova si fornisce un esempio di dati provenienti da un sistema di monitoraggio che utilizza NetFlow (con un ragionamento analogo è possibile fornire la stessa prova per IPFIX).

Output del comando *nfdump*:

Date flow start	Duration	Proto	Src IP Addr:Port		Dst IP Addr:Port	Packets	Bytes	Flows
2010-09-01 00:00:00.459	0.000	UDP	127.0.0.1:24920	->	192.168.0.1:22126	1	46	1
2010-09-01 00:00:00.363	0.000	UDP	192.168.0.1:22126	->	127.0.0.1:24920	1	80	1

Figura 1.3:

1.3 Packet sampling e Intrusion Detection

La sicurezza delle reti, specie di quelle molto estese, è un problema tutt'oggi presente e di un importanza vitale per qualsiasi azienda che operi nel settore ICT (Information and Communications Technologies). Le minacce possono presentarsi in qualunque momento [3] e provenire sia dall'interno che dall'esterno. Riuscire ad identificare queste minacce in tempo è il primo passo verso la soluzione di questo difficile problema. Per fare ciò è necessario avere un'ampia e continua sorveglianza della rete. Storicamente la sorveglianza di una grande rete era (ed è tuttora) affidata a sonde (*probes*), posizionate in punti strategici della rete. Questo è stato sufficientemente accurato fino ad ora, ma visto l' aumento delle switched point to point network, il monitoraggio *probe-based* non è più sufficiente. Implementare il monitoraggio già dall'interno di uno switch o di router sta diventando sempre più una necessità. Tuttavia le esigenze di mercato tendono a preferire l'ampiezza di banda sulla sicurezza, per cui la funzione di monitoraggio deve essere relegata come funzione secondaria all'interno di questi apparati di rete. È necessario quindi che questa funzione operi con il minimo overhead possibile, al fine di non degradare le prestazioni dell'apparato. È qui che entra in gioco la tecnologia sFlow, essa permette di delegare la parte di analisi del traffico ad un altro componente della rete (*il collector*), lasciando allo switch o al router le risorse per effettuare le normali decisioni di smistamento dei pacchetti.

Il rilevamento delle intrusioni basato sui flow è un modo innovativo di rilevare le intrusioni nelle reti ad alta velocità. Il rilevamento delle intrusioni basato sui flow controlla solo l' header del pacchetto e non ne analizza il *payload*. Analizzare il *payload* del pacchetto infatti sarebbe computazionalmente inaccettabile su reti di grandi dimensioni e con un elevato flusso di dati.

Questa limitazione però nel caso di sFlow si presuppone che non invalidi la qualità del sistema di monitoraggio poichè la stragrande maggioranza degli attacchi è riconoscibile tramite un'analisi degli header al di sopra del livello di Livello 3.

Vediamo ora più nel dettaglio come è possibile costruire con sFlow un sistema di monitoraggio efficace e che rispetta le richieste di mercato:

- Il sistema deve avere una sorveglianza continua network-wide: sFlow può essere configurato su ogni apparato di rete
- I dati devono essere sempre disponibili per rispondere efficacemente: sFlow manda immediatamente i pacchetti campionati al Collector con UDP
- I dati devono essere sufficientemente dettagliati per caratterizzare l'attacco: sFlow permette di esportare più di 128 Byte (comprendendo quindi ad esempio anche header di livello 7)
- Il sistema di monitoraggio non deve esporre gli apparato ad attacchi: sFlow non impatta sulle prestazioni degli apparati poichè viene effettuato in hardware inoltre il consumo di banda è limitato poichè i datagram sFlow sono ottimizzati

1.3.1 Metodi di rilevazione flow-based

L'analisi dei dati provenienti dai *flow generators* al fine di garantire una rilevazione efficace della maggior parte degli attacchi ha portato nel corso degli anni all'utilizzo di molteplici approcci, che utilizzano tre macroaree dell'intrusion detection moderna:

- *Statistical Analysis*
- *Machine Learning Analysis*
- *Signature-based Analysis*

I metodi statistici (*Statistical Analysis*) costruiscono un profilo del traffico di rete normale utilizzando una funzione statistica dei parametri del traffico di rete; questo profilo del traffico normale viene utilizzato per controllare il traffico in arrivo al momento attuale; la somiglianza tra il traffico di rete e il profilo del traffico di rete normale viene calcolata utilizzando appunto metodi statistici. Se la misura di somiglianza è al di sopra della soglia predefinita, il flusso è marcato come dannoso, altrimenti come flusso normale. [15] I metodi statistici non necessitano di conoscenza pregressa di un determinato attacco per determinare se un campione contiene effettivamente un attacco, e questo si è dimostrato particolarmente efficace nel rilevare attacchi di tipo DoS (*Denaial of Service*). Essi tuttavia possono essere aggirati mantenendo l'impatto dell'attacco al di sotto della soglia predefinita. Inoltre l'efficacia di questi metodi in reti reali è minata dalla difficoltà intrinseca che sta nel calcolare le statistiche sul traffico.

I metodi che utilizzano tecniche di *Machine Learning* si basano principalmente, su reti neurali, *Support Vector Matrix* (SVM), *Decision Tree* e *Clustering*. Essi presentano numerosi vantaggi rispetto ai metodi statistici quali l'adattamento del comportamento in base ad una finestra temporale. Inoltre i risultati degli esperimenti condotti con Machine Learning hanno evidenziato un'ottimo tasso di riconoscimento degli attacchi in un processo di "apprendimento" relativamente breve. Tuttavia anche questo metodo ha degli svantaggi importanti come la difficoltà di generare dati campioni per l'apprendimento, inoltre quest'ultimo si è rivelato computazionalmente molto costoso. Infine essi hanno un alto tasso di falsi positivi (ovvero allarmi che vengono generati da un pacchetto che inveve era benevolo).

In ultimo abbiamo la *Signature-based Analysis*, tema principale di questo elaborato, che utilizza l'header dei pacchetti campionati per fare pattern matching con un dataset di regole statiche. Questo approccio porta con se tutti i vantaggi e gli svantaggi degli IDS Signature-based discussi nei paragrafi precedenti, con il solo svantaggio aggiunto di non poter ricorrere all'utilizzo della Stateful Analysis poichè il campionamento non garantisce che tutti i

pacchetti di una determinata sessione siano catturati. Inoltre occorre ribadire che tale metodo è utilizzabile solo nel caso in cui si abbiano a disposizione almeno gli header di Livello 3, per quanto detto nella sezione "Alternative a sFlow", da cui ci si accorge che sFlow rappresenta l'unica alternativa di applicazione.

1.4 Elasticsearch e lo stack ELK

L'analisi dei dati, specie se di grosse dimensioni, come i dati generati dai test o dai log, è un problema che va affrontato con sistematicità al fine di poter trarre delle conclusioni descrittive sul lavoro svolto. È stato necessario quindi utilizzare tool appositamente studiati per la *Data Analysys* come lo stack ELK, ed in particolare Kibana.

Elasticsearch [16] è un motore di ricerca basato su Lucene. È un software con capacità di ricerca full-text, multitenant e distribuito. Possiede un' interfaccia web HTTP e una strutura schema-free di documenti con la sitassi JSON. A differenza di un *Database Management System* relazionale, esso è *schema-free* anche se non *schema-less* (come MongoDB per esempio) il che significa che non è necessario definire i tipi (string, integer, ecc.) dei dati prima di inserirli, ma comunque possibile. [18] Elasticsearch è sviluppato in Java ed è rilasciato con licenza Open Source Apache. I client ufficiali sono disponibili in Java,. NET (C), PHP, Python, Apache Groovy e molti altri linguaggi. Secondo DB-Engines, [17] Elasticsearch è il motore di ricerca più popolare in ambito enterprise e rappresenta un tool fondamentale nell'analisi e nella centralizzazione dei log.

Elasticsearch infatti è sviluppato, dallo stesso team, di pari passo con Logstash e Kibana e i tre programmi insieme formano quello che viene chiamato lo stack ELK.

Esso si compone di tre componenti fondamentali:

- **Logstash** : è una "pipeline di elaborazione dati" che può raccogliere dati da varie fonti, trasformarli e inviarli a vari consumatori, tra cui Elasticsearch
- **Elasticsearch** : come descritto prima, un database No-SQL particolarmente ottimizzato per la ricerca
- **Kibana** : è uno strumento di visualizzazione web-based che si integra con Elasticsearch atto a fornire un modo efficace per navigare e visualizzare i dati, utilizzando una varietà di grafici, grafici e tabelle.

Figura 1.4: Lo stack ELK

Lo stack ELK è ad oggi lo standard *de facto* per l'esplorazione dei log, esso si va ad affiancare a tutti i programmi critici all'interno di una infrastruttura e risulta particolarmente utile nell'esplorazione dei log degli IDS. Collegato ai log di un IDS permette infatti acquisire in pochissimi secondi conoscenza su cosa accade nella rete e su che tipo di attacchi sono in corso. I dati vengono fruiti all'operatore in maniera veloce e comprensibile tramite dashboard, fornendo una fotografia estremamente informativa dello stato attuale.

In figura è illustrato il funzionamento della pipeline di analisi usando lo stack ELK.

Capitolo 2

Proposta

In questo capitolo si va ad illustrare nel dettaglio quella che è stata ritenuta una possibile proposta di soluzione al problema del monitoraggio basato su packet sampling. Si andranno a descrivere i vari contesti di sperimentazione, i tool utilizzati e le modalità di utilizzo di questi ultimi. Si andrà poi ad introdurre il software *pcap2sflow*, sviluppato per simulare il comportamento di sFlow e garantire una riproducibilità dei test effettuati. Infine si descriveranno le modalità di catalogazione dei dati raccolti.

2.1 sFlow + Suricata

Data la natura del protocollo sFlow, ovvero quella di poter esportare l'header di un numero parametrizzabile di pacchetti passanti attraverso un interfaccia di rete verso un altro componente della rete, la conclusione più immediata a cui si è giunti e stata quella di analizzare questi header campionati con l'IDS Suricata. Come visto nel capitolo precedente, Suricata permette identificare possibili *threat* tramite l'utilizzo di *rules*. La particolarità di questo metodo sta nel fatto che molte delle signature descritte all'interno delle rules, fanno riferimento proprio all'heder del pacchetto. Questo rende Suricata un ottimo candidato per l'analisi di questi dati.

Come illustrato nel capitolo precedente, InMon Corp. mette già a disposizione dei tool per realizzare questo tipo di architettura, ovvero **hostsflowd** e **sflowtools**. Andiamo quindi ad illustrare quello che è stato il primo approccio verso l'analisi di questa soluzione.

2.1.1 L'architettura

L'idea è quella di collocare un solo collector in un punto strategico di una rete e inoltrare il traffico passante per tale collector verso un altro host su cui è installato l'Agent e Suricata.

L'intera sperimentazione si basa su due macchine virtuali identiche chiamate *Generator* e *Collector* con le caratteristiche elencate in tabella.

Numero di CPU	2
Architettura del processore	x86_64
RAM	3GB
Numero delle interfacce di rete	2
Sistema operativo	Linux
Distribuzione	Debian 8
Versione del Kernel	4.9.0-4-amd64

Installazione e configurazione

Una volta create le macchine virtuali usando il programma VirtualBox [19] e vi è stato installato Debian 8 [1].

Si è passato poi alla configurazione delle interfacce di rete, su entrambe le macchine, tramite il file `/etc/network/interfaces` come segue:

Generator

[1] Si tralasciano i dettagli sulla creazione delle macchine e sull'installazione di Debian, ma è comunque presente in bibliografia un riferimento rigurardo alla procedura seguita [28] [29]

```
# The primary network interface che comunica con il Collector
allow-hotplug enp0s3
iface enp0s3 inet dhcp

#Interfaccia di servizio
auto enp0s8
iface enp0s8 inet static
address 10.0.1.2
netmask 255.255.255.0

#interfaccia visibile dall'esterno per utilizzo con SSH
auto enp0s9
iface enp0s9 inet static
address 192.168.56.2
netmask 255.255.255.0
```

Collector

```
# The primary network interface che comunica con il Generator
allow-hotplug enp0s3
iface enp0s3 inet dhcp

#Interfaccia di servizio (non utilizzata)
auto enp0s8
iface enp0s8 inet static
address 10.0.1.3
netmask 255.255.255.0

#interfaccia visibile dall'esterno per utilizzo con SSH
auto enp0s9
iface enp0s9 inet static
```

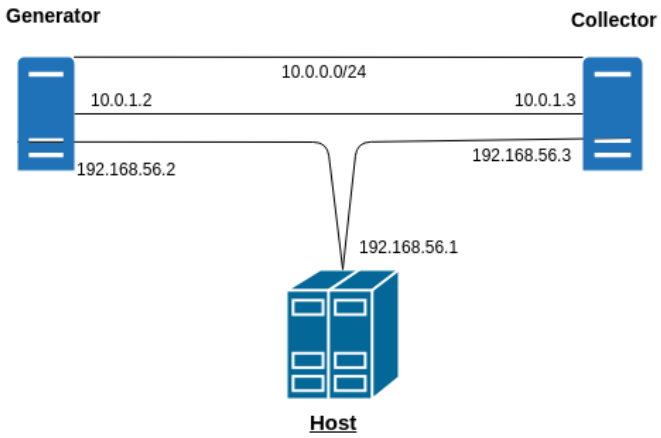

Figura 2.1: Topologia della rete

```
address 192.168.56.3
netmask 255.255.255.0
```

La topologia della rete aveva quindi una struttura uguale a quella rappresentata in figura 2.1.

Si sono quindi installati i due programmi hostsflow e sflowtools su Collector e su Generator rispettivamente. Poi si è installato Suricata su Collector e si sono scaricate le regole base dal sito di *Emerging Threats*. Inoltre si sono abilitati i log nel formato json. [20] Per concludere si è proceduto all'installazione dello stack ELK per l'analisi dei log.

L'architettura finale del setup creato per le sperimentazioni aveva la struttura mostrata in figura 2.2 .

Si ha quindi che, hostsflow monitora l'interfaccia di rete `enp0s8` e invia i campioni incapsulati in datagrams sFlow a sflowtools sulla porta standard 6343, quest'ultimo spacchetta i campioni e li riscrive. Successivamente i pacchetti trascritti da sflowtool vengono dati da analizzare a Suricata, il quale opera una *offline analysis*. Infine i log generati da Suricata vengono trascritti all'interno di Elasticsearch utilizzando lo script *elasticsearch_feeder.py*. Ora

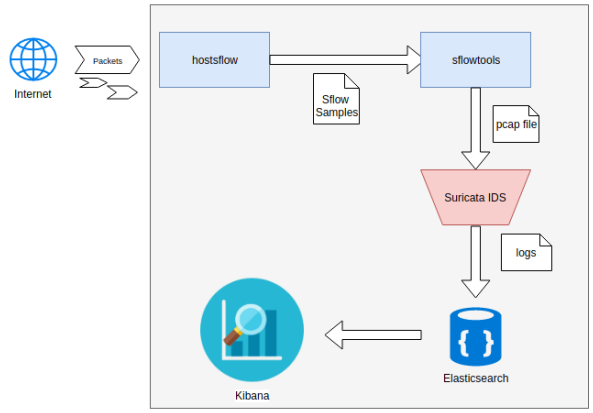

Figura 2.2: Architettura I

che i dati sono indicizzati in Elasticsearch è possibile esplorarli e creare delle visualizzazioni all'interno dell'interfaccia web di Kibana.

2.2 Datasets

Al fine di garantire la riproducibilità dei risultati ottenuti da questo studio, si è scelto di utilizzare una collezione di datasets spesso utilizzati nella letteratura per il testing degli Intrusion Detection Systems [21].

I dataset scelti per il testing sono stati:

- DARPA Intrusion Detection Data Set (1999) [22]
- CTU-13 Dataset [23]
- ICtF 2010 *(International Capture the Flag, ed. 2010)* [24]

Il *DARPA Intrusion Detection Data Set (1999)* è un dataset composto da traffico misto, acquisito in tre settimane di training di un IDS. Nella prima e la terza settimana non sono presenti attacchi mentre nella seconda settimana sono stati eseguiti degli attacchi principalmente mirati a far scattare le regole degli IDS oggetto di test.

Il *CTU-13* è un dataset sul traffico *botnet* che è stato acquisito nell' Universitá CTU, Repubblica Ceca, nel 2011. L' obiettivo del dataset era quello di ottenere un' ampia cattura del traffico delle botnet reali mescolato al traffico normale e al traffico di sottofondo. Il set di dati CTU-13 consiste in tredici acquisizioni (chiamate scenari) di diversi campioni di botnet. In ogni scenario è stato eseguito un malware specifico, che ha utilizzato diversi protocolli e ha eseguito diverse azioni

Infine *ICtF 2010* è un dataset ricavato dal traffico generato durante l'edizione del 2010 dell'International Capture the Flag di Santa Barbara. Si tratta quindi di un dataset derivante da una competizione in cui si mettono alla prova le capacità di *penetration testing* dei partecipanti. Esso costituisce per questo motivo un ottimo dataset di testing poichè , a differenza dei precedenti, non è stato costruito appositamente per studiare determinati comportamenti di un IDS in specifiche condizioni.

2.3 Metodologie di test

Si vanno adesso ad elencare le metodologie dei test effettuati, la terminologia, l'organizzazione delle directory e lo schema del database di Elasticsearch.

2.3.1 Terminologia

- ***Sampling Rate*** : è la frequenza di campionamento operata dall'Agent sFlow, ad esempio un Sampling Rate uguale a 4 vuol dire che ogni 4 pacchetti uno viene campionato

- ***Troncamento*** : è la dimensione massima del pacchetto scelto per il campionamento , tale pacchetto viene troncato fino a t Bytes e inviato al collector.

- ***Setup On-Wire*** : è il setup per cui il traffico viene inoltrato interamente a Suricata per l'analisi. Esso corrisponde al normale utilizzo del software in questione.

- ***Alert*** : è un messaggio di allarme dell' IDS che notifica l'amministratore di una violazione delle policy di sicurezza.

L'obbiettivo dei test effettuati è verificare se gli alert sollevati da Suricata in modalità On-Wire sono paragonabili a quelli che si ottengono fornendo ad esso solo i dati parziali derivanti da sFlow. Inoltre si vuole verificare l'incisività dei parametri di sampling rate e troncamento e se essi siano indipendenti l'uno dall'altro.

2.3.2 Struttura delle directory

Con l'obiettivo di tenere organizzati i risultati dei numerosi test effettuati si è scelto di seguire una rigorosa struttura delle directory e dei file.

Partendo dalla directory *root* del progetto si trovano le sottodirectory dei dataset analizzati, all'interno di esse vi sono i test numerati da 1 a n e all'interno di ogni directory di test troviamo i files: `alert-debug.log`, `eve.json`, `fast.log`, `pcap_size.txt`, `stats.log`, `test_info.txt`. Dove:

- `alert-debug.log`, `eve.json`, `fast.log`, `stats.log` rappresentano i log di Suricata relativi al test in esame

- `pcap_size.txt`, `test_info.txt` rappresentano rispettivamente la quantità di traffico generato dal tipo di campionamento scelto e i parametri di campionamento.

```
/root
  /dataset-name
    /number of test
      alert.debug
      eve.json
      fast.log
      pcap_size.txt
      stats.log
      test_ingo.txt
```

2.3.3 Il cuore degli esperimenti

Inizialmente, soprattutto nelle prime fasi di analisi di fattibilità dell'applicazione oggetto di studio i test sono stati effettuati a mano, utilizando come solo supporto una checklist, che assicurava la coerenza della struttura delle directory così come del contenuto dei risultati dei test stessi.

Sulla macchina Generator è stato configurato hostsflowd tramite il suo file di configurazione `/etc/hsflowd.conf` includendo i seguenti campi:

```
  # hsflowd configuration file
  # http://sflow.net/host-sflow-linux-config.php


sflow {
     agent = enp0s8
     polling = 30
     sampling = 2
     collector { ip=10.0.0.3 udpport=6343 }
     pcap { dev =  enp0s8 }
     headerBytes = 128
}
```

e si avvia hsflowd con il comando :

```
  hsflowd -d
```

In questo modo l'Agent hsflowd analizza il traffico passante per l'interfaccia di rete enp0s8 e invia i datagram sFlow alla macchina Collector, sul quale è in esecuzione il programma sflowtool

```
sflowtool -p 6343 -t | suricata -c /etc/suricata/suricata.yaml -r -
```

Nella seconda parte del comando di sopra, il traffico viene passato a Suricata che effettua un analisi offline trattando l'input come se fosse un normale file pcap.

A questo punto tutto è pronto e il traffico dei dataset può essere iniettato nella rete all'interfaccia monitorata da hsflowd. Quindi, posizionandosi sul Generator stesso si esegue il comando:

```
tcpreplay -i enp0s8 <file pcap da analizzare>
```

Problemi È tuttavia emerso un problema da questa metodologia, tcpreplay cerca di iniettare il traffico nell'interfaccia rispettando, come e giusto che sia, il timing di arrivo dei pacchetti registrati nel file pcap. Questo sebbene assicuri una precisione certamente desiderabile oltre a garantire la riproducibilità dei test, non è una strada percorribile (almeno per gli obiettivi di questa tesi) poichè i test richiederebbero troppo tempo per essere portati a termine. È altresí vero che è possibile istruire tcpreplay di non rispettare il timing, ma questo andrebbe a invalidare quei regole di Suricata che fanno affidamento su di esso, come ad esempio il rilevamento del famoso attacco *Slow HTTP*.

Inoltre, riguardo al voler determinare la l'incisività dei parametri sampling rate e troncamento, è emerso che il programma utilizzato come Agent, hsflowd, non è sufficientemente parametrizzabile. In particolare la dimensione massima del pacchetto che può essere esportato, ovvero quello che viene chiamto *header size* e che abbiamo identificato come troncamento, è *hard-coded* ad un massimo di 256 Bytes. Sebbene sia stato spiegato dalla InMon Corp. che tale dimensione massima è dovuta all'architettura di sFlow e che è necessaria per garantire una scalabilità del prodotto, alle finalità di testing questa si è rivelata una grossa limitazione, soprattuto quando i risultati ottenuti si sono posti in contraddizione con quanto ci si aspettava.

In definitiva, sebbene questo primo approccio è stato determinante per verificare la fattibilità di affiancare sFlow a Suricata esso non è per nulla soddisfacente ai fini della valutazione di efficienza.

2.4 Una soluzione di testing alternativa

Per quanto descritto nella sezione precedente, si è resa necessaria una nuova modalità di test, che sia meno dipendente dall'implementazione di tutta la tecnologia è che lasci sufficienti margini di sperimentazione, anche se questo vuol dire uscire dai paradigmi della tecnologia sFlow.

2.4.1 Pcap2sFlow

Si è sviluppato quindi un emulatore di sFlow che converte un file pcap in input, in un altro file pcap contenente pacchetti campionati ricalcando le tecniche sFlow.

Si tratta di un programma scritto in C che utilizza la *libpcap* [25], una libreria C per analizzare il traffico *live* o *captured*.

In particolare si sono utilizzate le seguenti funzioni di libreria:

```
pcap_dumper_t *pcap_dump_open(pcap_t *p, const char *fname);
```

che apre il livecapture o il file pcap da cui leggere i pacchetti,

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
```

che processa i pacchetti da un livecapture o, come nel nostro caso da un *"savefile"*, ovvero un file in formato pcap, e ovviamente infine

```
void pcap_close(pcap_t *p);
```

per chiudere il "savefile".

Meccanismi di campionamento

Per emulare il comportamento di sFlow occorre applicare lo stesso metodo di campionamento utilizzato da quest'ultimo per scegliere quale pacchetto campionare e come troncare il pacchetto in questione.

Si è scelto di prendere in considerazione solo il *Packet Sample* e di tralasciare il *Counter Sample* poichè quest'ultimo non gioca alcun ruolo nel IDS studiato (si veda la sezione 1.2.1).

Il Pachet Sampling all'interno di sFlow utilizza un meccanismo di randomizzazione per determinare il pacchetto designato al campionamento. Pcap2sFlow allo stesso modo divide il file da campionare in *windows* (finestre) della dimensione del parametro *sampling rate*. All'interno di ogni finestra esso sceglie in modo random il pacchetto da campionare. Di seguito sono riportati i listati dell'implementazione di tali funzioni.

```
void select_random (cap_stat *stat) {
  //seleziona il prossimo pacchetto in modo random partendo
  //dall'ultima finestra considerata
  srand ( time(NULL) ) ;
  int x = rand();
  stat->s = stat->start + ( x % stat->n ) ;
}
void next_window (cap_stat *stat){
    //analizza la successiva finestra di pacchetti
    stat->start = stat->stop + 1;
    stat->stop = stat->stop + 1 + stat->n;
    select_random(stat);
}

bool to_sample ( cap_stat *sampler_info ) {
    //se n == 1 allora disattiva il campionamento e salva tutti
    //i pacchetti
    if (sampler_info->n == 1) return true;

    //controlla se il pacchetto corrente deve essere campionato
    if (sampler_info->curr_sample < sampler_info->start) {
        //se sono andato avanti con la window ma il cursore e‘
```

```
        //ancora indietro questo pacchetto sara‘ sicuramente
        //da scartare
        sampler_info->curr_sample++;
        return false;
    }
    if (sampler_info->curr_sample == sampler_info->s) {
        //se il pacchetto corrente e‘ quello designato per essere
        //campionato avanza e rispondi true
        sampler_info->curr_sample++;
        next_window(sampler_info);
        return true;
    }
    else {
        sampler_info->curr_sample++;
        return false;
    }
}
```

Nel testo dell'RFC 3176 viene detto che il seed della funzione `select_random` non dovrebbe essere alimentato con il tempo, come invece è stato fatto in pcap2sflow, poichè esso potrebbe non essere abbastanza casuale in reti con un elevato *throughput*. Infatti in quel contesto la velocità di aggiornamento delle windows potrebbe essere più veloce della velocità di aggiornamento del tempo. Tuttavia si è ritenuto che nel contesto applicativo dell'esperimento questo non possa accadere.

Infine occorre troncare il pacchetto secondo una dimensione parametrizzabile, e questo viene effettuato dalla seguente funzione:

```
void save_packet (cap_stat *s, const struct pcap_pkthdr *header,
                  const u_char *packet){
    if (s->truncate_b == 0) {
        //0 viene considerato come valore di default per
```

```
        //"salva tutto il pacchetto"
        pcap_dump((u_char *)s->pdumper, header, packet);
        return;
    }
    if (header->caplen > s->truncate_b) {
        u_char *new_packet = malloc(s->truncate_b);
        memcpy(new_packet,packet,s->truncate_b);
        pcap_dump((u_char *)s->pdumper, header, new_packet);
        free(new_packet);
    }
    else {
        pcap_dump((u_char *)s->pdumper, header, packet); } }
```

Si è inoltre utilizzata la funzione di libreria *getopt* per rendere il programma altamente parametrizzabile e *"script-friendly"*. Con il parametro `--help` è possibile visualizzare tutte le opzioni disponibili:

```
root@collector:# ./pcap2sflow --help
usage: ./pcap2sflow --infile <input_file.pcap> --outfile
        <output_file.pcap>

options:
-t      --trunkate-pkts <bytes>    truncate packet size to (bytes)
                                   size
-n      --no-truncate-pkts         do not truncate packets
-s      --sample-n <N>             sample 1 every N packets seen
-N      --no-sample                do not sample every N packets,
                                   take them all!
-h      --help                     show this help
```

2.5 Metodologia di test 2

Data l'enormità dei dataset da testare ed il numero di parametri che si intendevano testare, il lavoro di testing e catalogazione è stato interamente svolto mediante l'uso di script in Bash e Python. Quando ci si è chiesti se sviluppare un programma unico per il test o una *suite* di tools si è scelto di abbracciare la filosofia Linux del *"Do One Thing And Do It Well"* [30], infatti un programma unico, date le molte sfaccettature del progetto, sarebbe stato poco pratico.

Solitamente (tranne nel caso del dataset di DARPA) il *dump* del traffico era diviso in segmenti della dimensione di 400 Megabyte circa. Lo script sviluppato si occupa quindi di prendere ogni singolo file, campionarlo con pcap2sflow e far analizzare a Suricata il file così generato. Infine si occupa di salvarne i risultati, ovvero i log e la dimensione del file pcap dopo il campionamento.

In questo modo è stato possibile semplificare l'architettura illustrata precedentemente in figura 2.2 con quella della figura 2.3, in cui si può notare che il software pcap2sflow ha sostituito sia hostsflow che sflowtools, impersonando quindi sia l'Agent che il Collector.

Si è prestata particolare attenzione durante le fasi di test alle performance e al tempo di generazione dei risultati. Ad esempio, sebbene si possa campionare un file pcap e subito darlo ad analizzare a Suricata, con il comando:

```
./pcap2sflow -i <pcap file> -s 100 -t 128 -o - | \
      suricata -c /etc/suricata/suricata.yaml -r -
```

per ogni test che si sarebbe andato ad effettuare Suricata avrebbe dovuto ricaricare tutto il core e tutte le regole. Questo si traduceva in uno spreco considerevole di tempo che avrebbe allungato il tempo di esecuzione dei test. Allora si è optato per una soluzione più moderna della classica PIPE di Linux che utilizza l'interfaccia a *socket unix* di Suricata. Tramite questa interfaccia

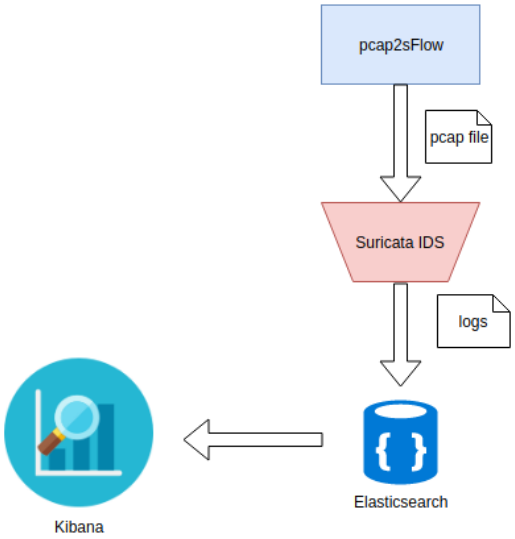

Figura 2.3: Architettura II

è possibile caricare una volta sola il core e le regole e aggiungere in una coda i file pcap da analizzare. Si è utilizzato per questo motivo il comando

```
suricatasc --command "pcap-file <file_pcap> <directory di log>"
```

2.6 Analisi dei risultati con Kibana e Elasticsearch

Passiamo adesso alla descrizione di come i log e i dati in generale derivati dai test effettuati siano stati organizzati per fornirne una rappresentazione informativa. Si è utilizzato, come anticipato in precedenza Kibana ed Elasticsearch, ma non si è utilizzato come avviene di solito, il terzo componente dello stack, ovvero Logstash. Questo perchè la tipologia di dati raccolti necessitava di una manipolazione più approfondita di quella che poteva fornirci Logstash. È stato necessario infatti scrivere un piccolo programma Python, *elasticsearch_feeder* , che attraversa tutte le directory dei risultati dei test

e arricchisce i dati degli alert con quelli dei parametri di campionamento e troncamento.

Un tipico alert di Suricata che è possibile trovare dentro il file eve.json è il seguente:


```
{"timestamp":"1999-03-08T15:48:07.793208-0600","flow_id":3806400292,
"pcap_cnt":516386,"event_type":"alert","src_ip":"172.16.112.149",
"src_port":23, "dest_ip":"135.13.216.191","dest_port":20945,
"proto":"TCP",
  "alert":{"action":"allowed","gid":1,"signature_id":2019284,"rev":3,
  "signature":"ET ATTACK_RESPONSE Output of id command from HTTP server",
  "category":"Potentially Bad Traffic","severity":2}}
```


Ognuno di questi alert rappresenta all'interno del database Elasticsearch un documento. Per ogni dataset è stato definito lo schema dell'index di Elasticsearch, come nell'esempio seguente sul dataset di ICtF2010:


```
{
  "ictf2010": {
    "aliases": {},
    "mappings": {
      "alerts": {
        "properties": { "adaptive_sampling": { "type": "boolean" },
          "alert": {
            "properties": {
              "category": {
                "type": "text",
                "fields": {
                  "keyword": {
                    "type": "keyword",
                    "ignore_above": 256 } } },
              "signature": {
                "type": "text",
```

```
      "fields": {
        "keyword": {
          "type": "keyword",
          "ignore_above": 256 } } } } },
"category": {
  "type": "text",
  "fields": {
    "keyword": {
      "type": "keyword",
      "ignore_above": 256 } } },
"dest_ip": {
  "type": "text",
  "fields": {
    "keyword": {
      "type": "keyword",
      "ignore_above": 256 } } },
"dest_port": { "type": "long" },
"filename": {
  "type": "text",
  "fields": {
    "keyword": {
      "type": "keyword",
      "ignore_above": 256 } } },
"proto": {
  "type": "text",
  "fields": {
    "keyword": {
      "type": "keyword",
      "ignore_above": 256 } } },
"sampling": { "type": "long" },
"signature": {
```

```
            "type": "text",
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256 } } },
          "src_ip": {
            "type": "text",
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256 } } },
          "timestamp": {
            "type": "date",
            "format": "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'SSSSSSZ" },
          "traffic": { "type": "long" },
          "truncate": {
            "type": "long" } } } },
      "settings": {
        "index": {
          "refresh_interval": "10s",
          "number_of_shards": "5",
          "provided_name": "ictf2010",
          "number_of_replicas": "1", } } } }
```

Una volta creato l'index si è passati alla popolazione del database con il programma Python elasticsearch_feeder.py Vista la grande quantità di log [2], si è ottimizzato l'inserimento utilizzando la funzione di Elasticsearch *bulk index*.

Si è allora disabilitato temporaneamente il refresh degli index [26] [27]

```
PUT /ictf2010/_settings
```

[2]Si sono raggiunti 12 GB di log solo per il testing del dataset ICtF2010

```
{
    "index" : {
        "refresh_interval" : "-1"
    }
}
```

E alla fine di ogni bulk index il valore è stato ripristinato ed è stato forzato il merging dei segmenti:

```
PUT /ictf2010/_settings
{
    "index" : {
        "refresh_interval" : "10s"
    }
}

POST /twitter/_forcemerge?max_num_segments=5
```

Analisi dei falsi positivi

Di fondamentale importanza nell'analisi prestazionale di un IDS è il conteggio dei falsi positivi, ovvero di quegli alert sollevati dall'IDS a partire da un traffico che era invece benevolo.

La valutazione dei falsi positivi è stata effettuata mediante un altro programma Python, scritto appositamente sempre utilizzando le API di Elasticsearch. Non è stato possibile infatti conteggiare i falsi positivi già dall'interno di Kibana a partire dai dati già presenti poichè Elasticsearch non permette, a differenza dei database relazionali, di effettuare delle subqueries.

Il programma Python, che va sotto il nome di false_positives.py, confronta le signature rilevate con i vari parametri di sampling rate e troncamento testati con le signature del setup On-Wire. Il numero di falsi positivi per ogni signature ricavati dalla sottrazione dei due valori è stata aggiunta in un ulteriore index, appositamente creato per il conteggio dei falsi positivi.

false_positives.py effettua una query con aggregazione sul valore "signature.keyword" nei documenti con sampling rate = 1 e truncate = 600 (presi come riferimento per il setup On-Wire) ne salva i valori e poi per ogni combinazione di parametri effettua una seconda query con aggregazione sullo stesso valore; infine la differenza tra i due valori ottenuta viene usata per creare tanti documenti quanti sono i falsi positivi. Occorre notare che sebbene si creino molti documenti, data la struttura di Elasticsearch e del suo Inverted Index [31], questo non occupa molto spazio e se ne guadagna in comodità una volta che si procederà col creare le visualizzazioni.

Capitolo 3

Discussione dei risultati

In questo capitolo si vanno ad elencare i criteri di valutazione dei risultati ottenuti dai test e se ne discuteranno i contenuti degli stessi.

Infine si proporrà un approccio alternativo al campionamento statico, ovvero il campionamento adattivo.

(ACCENNI SU INVERTED INDEX)

3.1 Criteri di valutazione

In ogni contesto di sperimentazione, ovvero al variare di parametri dei test e di dataset, si sono osservati i comportamenti dei seguenti indicatori:

- **Alert** : la quantità totale di alert sollevati da suricata
- **Falsi positivi** : la quantità di alert che nel setup On-Wire non è stata rilevata, e che per questo motivo costituisce sicuramente un false positivo
- **Signature** : la tipologia di signature rilevata e la loro quantità
- **Traffico**: la quantità in Bytes di traffico inviata per l'analisi a Suricata
- **Capacità a posteriori di identificare la sorgente e la destinazione di un attacco** : Questo indicatore risulta particolarmente utile nel

campo dell'analisi forense, ovvero consente di identificare una volta scoperto l'attacco quali sono stati gli host coinvolti e da dove provenivano gli attaccanti

- **Tempo di reazione** : la differenza di tempo per la rilevazione di un attacco in corso

3.2 Risultati

Per ogni test sono state provate tutte le combinazioni tra sampling rate uguale a 2, 4, 100 e troncamento uguale a 128, 256, 512 bytes, ad eccezione di ICtF2010, per il quale si sono teste diverse combinazioni di sampling rate 4, 100, 200, 400. Poi esse sono state messe in relazione agli stessi test effettuati in modalità On-Wire, con sampling rate uguale a 1 e senza troncamento (nei test indicato con il valore 600 per comodità di rappresentazione).

3.2.1 ICtF2010

Per il dataset ICtF2010 non è stato possibile identificare un rule-set specifico contro cui testare il tasso di rilevamento; si è quindi proceduto ad effettuare i test con il rule-set standard di Emerging-Threats.

In questa prima fase si voleva osservare l'impatto che hanno campionamento e troncamento e se l'esclusione dell'uno o dell'altro portasse a miglioramenti apprezzabili. Si va a valutare in questi test il numero di alert e il loro rapporto con il numero dei falsi positivi.

In figura 3.1 si mettono in relazione il numero degli alert al variare del campionamento con molteplici valori di troncamento. Similarmente in figura 3.2 ma invertendo i parametri.

Come è possibile notare, a parte i risultati ottenuti con il campionamento 1/1 e il troncamento a 128 bytes, la reale differenza tra il numero degli alert viene data dal valore del sampling rate. Per cui un troncamento uguale a 128

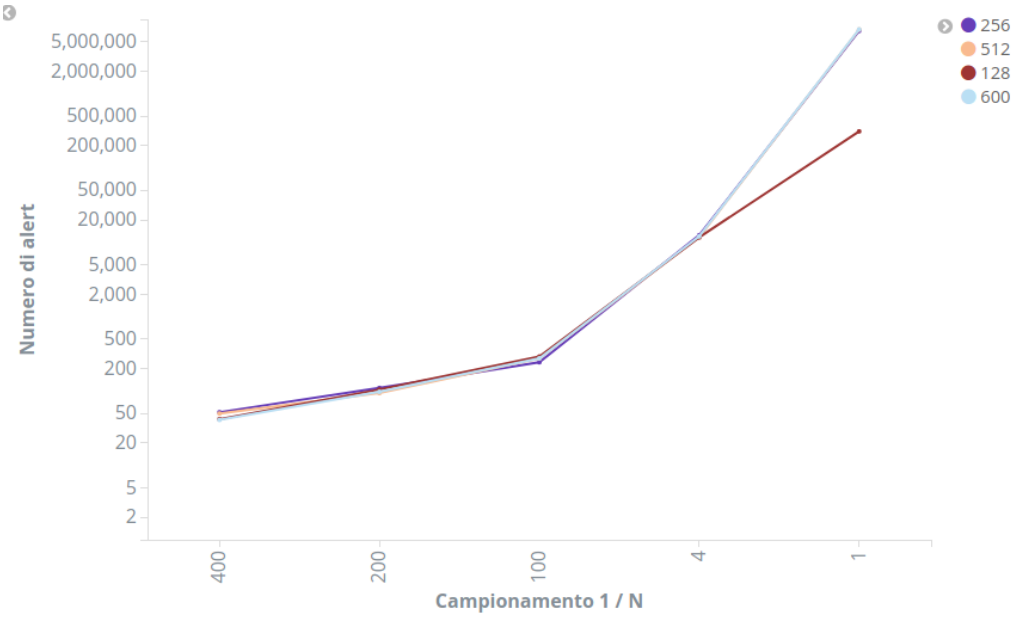

Figura 3.1: Numero di Alert al variare del sampling rate

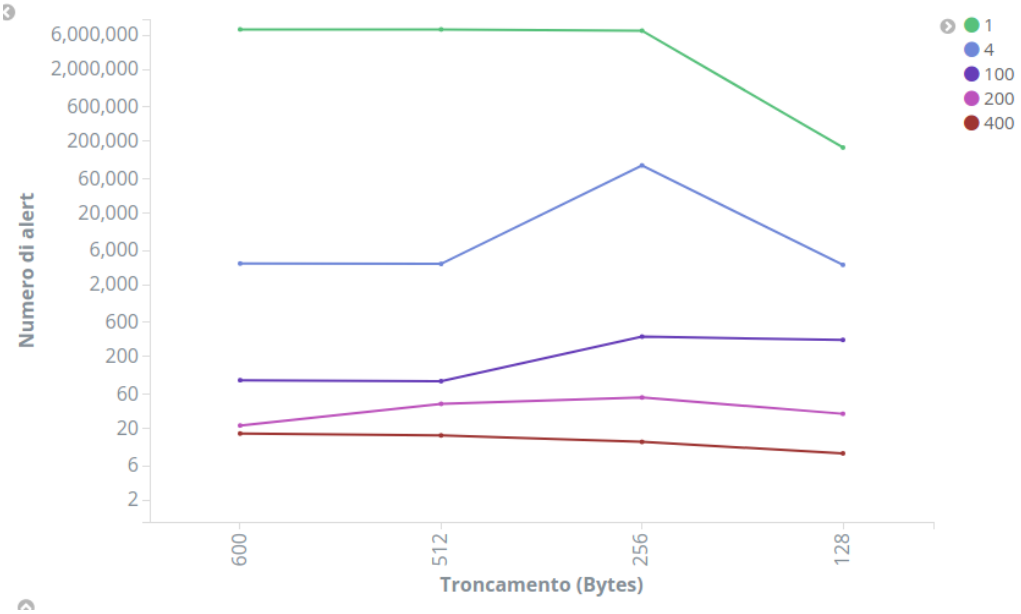

Figura 3.2: Numero di Alert al variare del troncamento

bytes, nel tipo di traffico del dataset ICtF2010, rappresenta un valore limite per garantire l'efficacia del sistema di rilevamento.

Osserviamo poi nelle figure 3.3 e 3.4 come i valori di troncamento e campionamento influiscono sul traffico che viene inviato al collector per l'analisi.

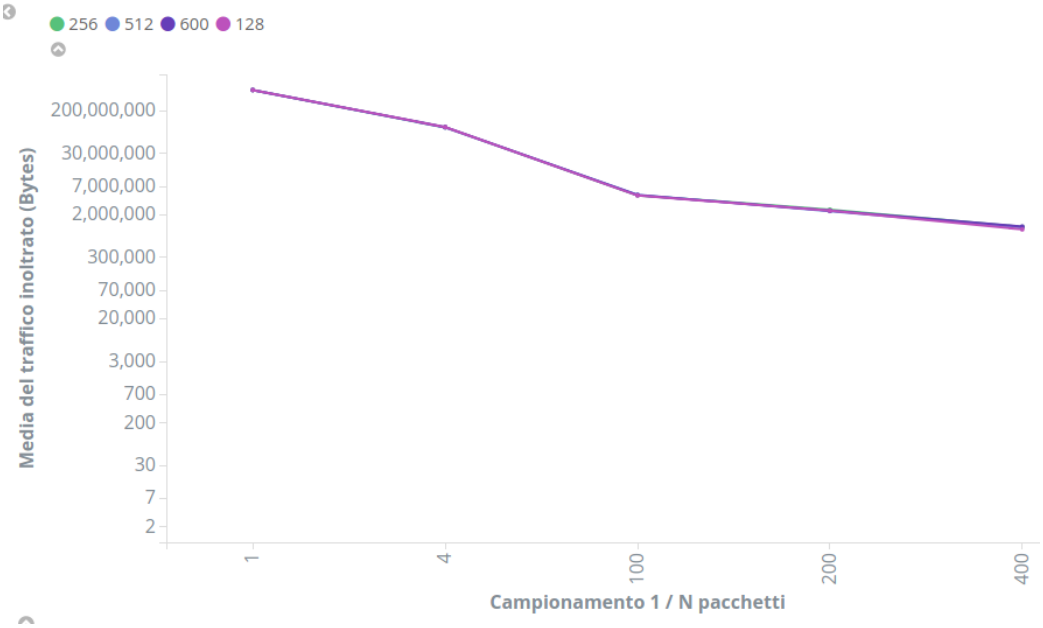

Figura 3.3: Quantità di traffico inoltrata al Collector

Di nuovo è possibile notare come la quantità di traffico inoltrata al Collector, per il dataset ICtF2010 dipende in primo luogo dal valore del sampling rate. Con un sampling rate di 1/100 infatti si riduce la quantità di traffico da inoltrare al collector da 500MB per ogni file pcap analizzato, fino a circa 3MB. Mentre un il troncamento, che sia a 128, 256 o 512, non produce un miglioramento apprezzabile.

Infine osserviamo che un campionamento con sampling rate da 100 in su non permette di rilevare quasi nessuno degli oltre 5 milioni di attacchi rilevati con sampling rate uguale a 4 o ad 1.

Analizziamo a questo punto il contenuto degli alert, ovvero ci soffermeremo ad osservare le signature rilevate, quali sono le più frequenti per un

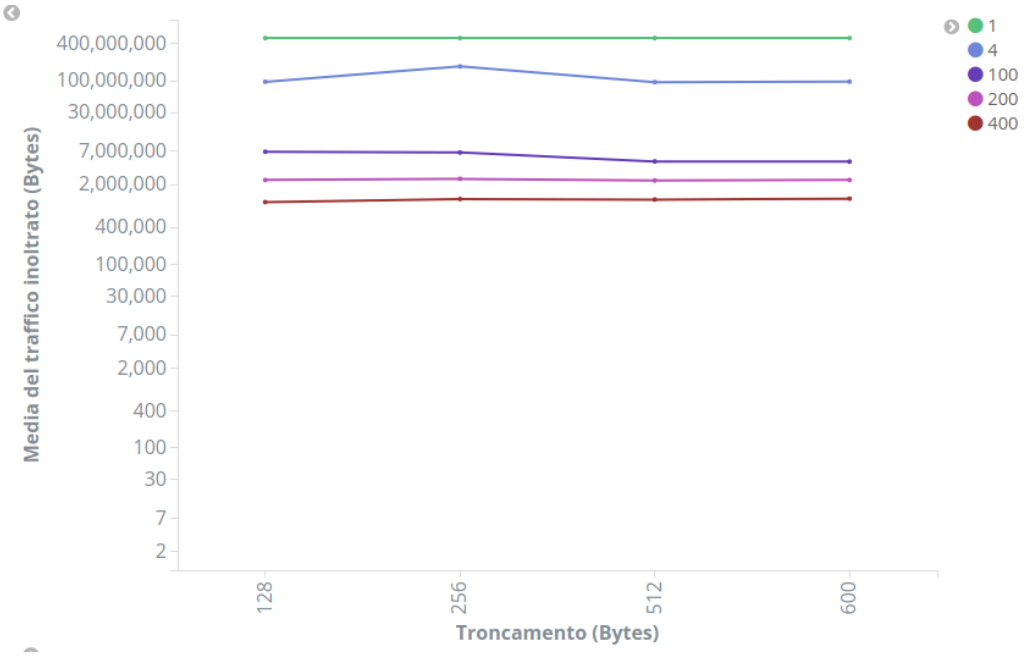

Figura 3.4: Quantità di traffico inoltrata al Collector

determinato tipo di campionamento e se esse siano o meno dei falsi positivi.

Dalla figura 3.5 si vede che i test con troncamento a 128 e sampling rate 4 sono sono quelli che hanno il numero di falsi positivi (in rosso) in rapporto al numero degli alert totali più alto di tutti. Mentre il troncamento a 256 byte e sampling rate 4 è quello che ha una percentuale di falsi positivi più bassa, come mostrato in tabella 3.1.

Infatti, come è possibile vedere nella figura 3.6, il troncamento a 128 byte porta ad un incremento notevole del numero degli alert che vengono attivati

Troncamento	**# Alert**	**# Falsi Positivi**	**%**
128	11688	6472	55%
256	133649	7151	5.3%
512	11786	6459	54.8%

Tabella 3.1: Falsi positivi con sampling rate 4

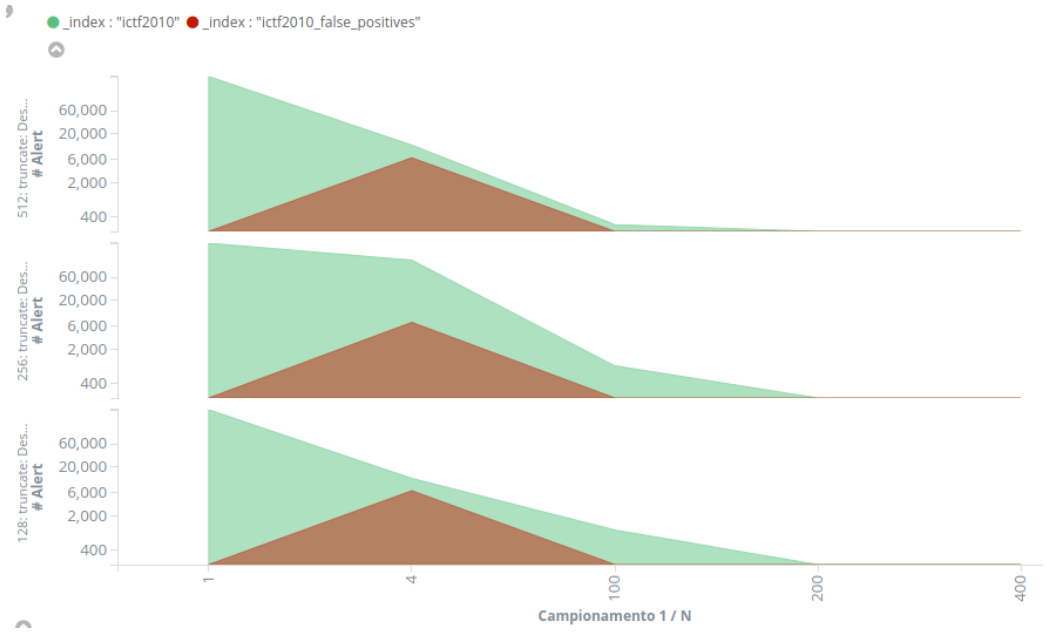

Figura 3.5: Numero di falsi positivi

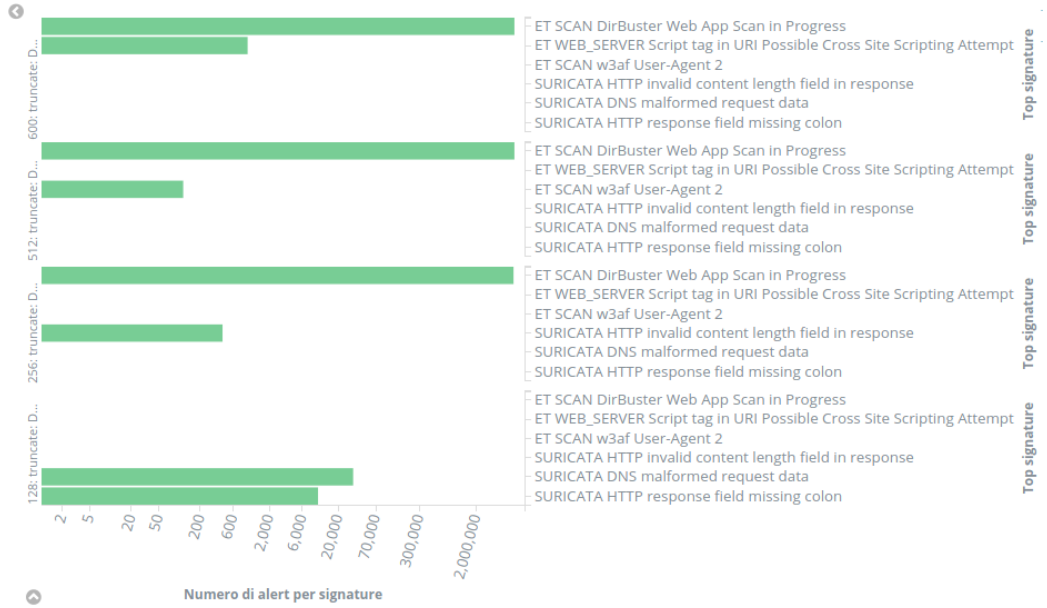

Figura 3.6: Numero di falsi positivi

dalle signatures *"SURICATA DNS malformed request data"* e *"SURICATA HTTP response field missing colon"*, che sono sintomatici di un troncamento eccessivo, e che porta quindi ad avere headers che non rispettano più le specifiche del protocollo (HTTP, DNS o altri).

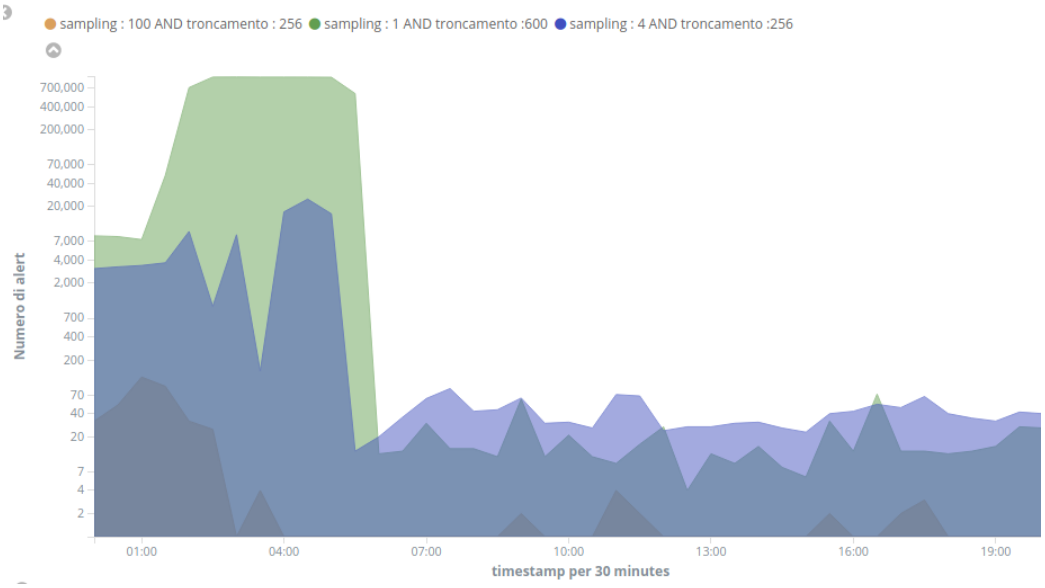

Figura 3.7: Numero alert nel tempo

Fissato il troncamento a 256 bytes mostriamo nella figura 3.7 il numero di alert durante tutta la competizione con sampling rate 4 e 100. Si osserva qui che i test con sampling rate 4 ricalcano abbastanza fedelmente l'attività registrata da Suricata On-Wire (in verde). Invece i test con sampling rate 100 sono stati efficienti nel primo periodo ma del quasi inutili nel secondo. L'andamento più ripido nella prima parte della competizione è compatibile con le prime due fasi del penetration testing, che sono riconocimento del target e scansione [34]. A questo periodo infatti corrisponde un grande numero di alert (circa 6M in totale) con signature *"ET SCAN DirBuster Web App Scan in Progress"*. Questa è la fase in cui i team si esponevano di più al fine di rilevare eventuali falle nel sistema. Non appena passata questa fase, le tecniche si sono notevolmente affinate, portando il numero di alerts a

mantenersi circa costante.

Inoltre come è possibile osservare nel periodo che va dalle 7:00 alle 19:00 il gli alert con sampling rate 4 sono in alcuni punti superiori a quelli del setup On-Wire. Questo è sintomo di un aumento dei falsi positivi in quel periodo di tempo, poichè avendo usato lo stesso rule-set per sampling rate 4 e On-Wire, un numero maggiore di alert nel primo indica che sicuramente vi saranno dei falsi positivi.

Avendo mostrato nella sezione precedente l'effetto dei vari tipi di campionamento e del loro impatto sui tipi di attacco rilevabili andiamo adesso a fare una analisi puramente quantitativa sui due dataset DARPA e CTU-13.

3.2.2 DARPA Dataset

Il Dataset Darpa è stato testato con le sole regole descritte al capitolo precedente. Si effettuerà quindi una analisi della percentuale di attacchi rilevati senza considerare i falsi positivi. Infatti utilizzando solo le regole specifiche per tale dataset non è possibile riscontrare dei falsi positivi semplicemente perchè le regole che li azionerebbero non sono caricate.

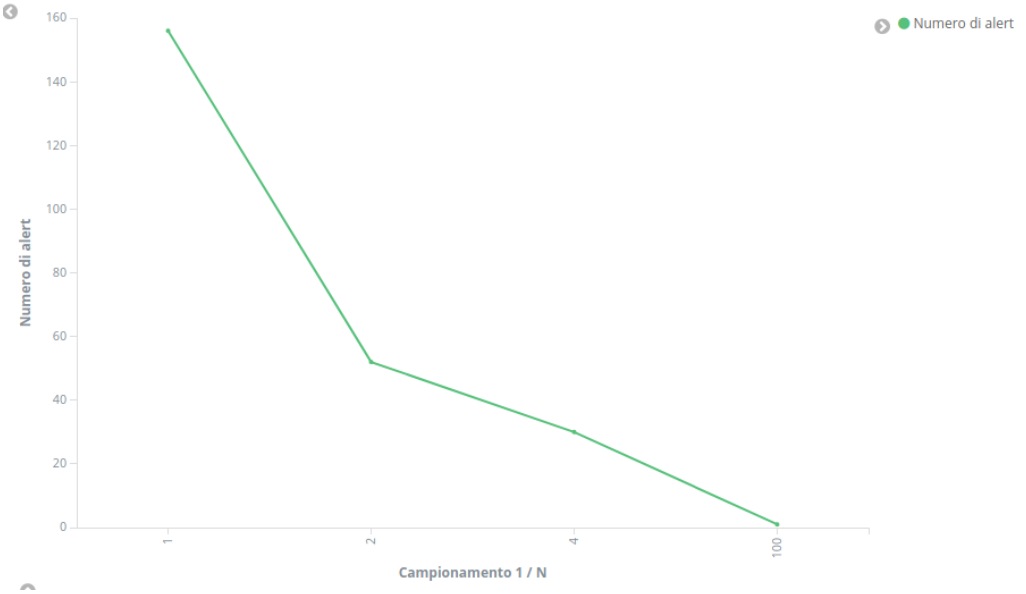

Figura 3.8: Numero di Alert al variare del sampling rate

Come si è osservato nella sezione precedente con il dataset ICtF2010 , le figure 3.8 e 3.2.2 mostrano che il tasso di rilevamento decresce con l'aumentare del sampling rate e invece rimane circa costante al variare del campionamento.

Infine nella tabella 3.2 vengono messi in relazione il numero di alert con il traffico medio inoltrato al Collector e notiamo che per questa analisi quan-

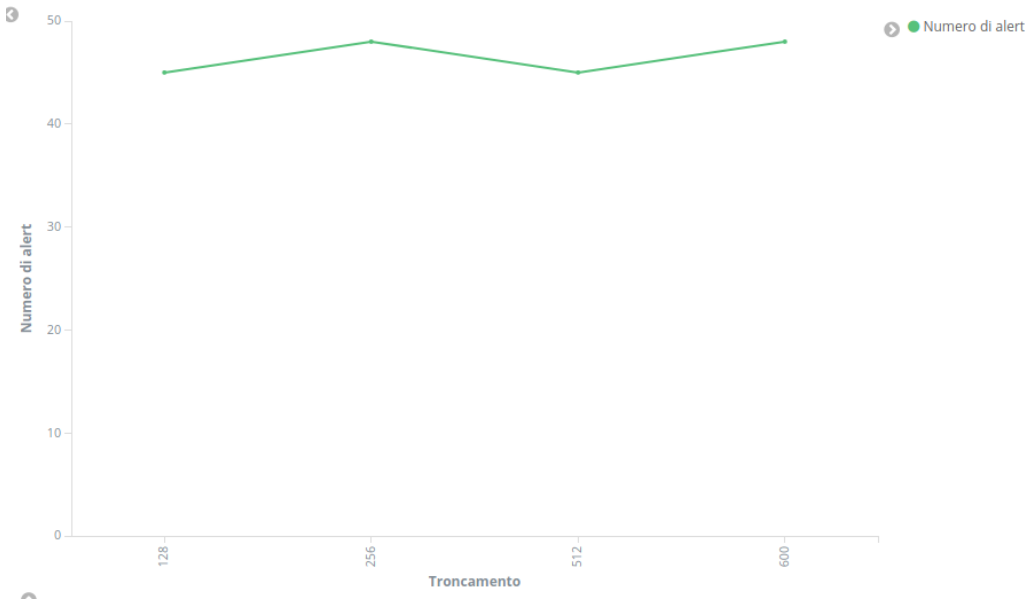

Figura 3.9: Numero di Alert al variare del troncamento

titativa i risultati non ci sorprendono e il numero di alert è direttamente proporzionale alla quantità di traffico, per ogni valore di troncamento. In particolare i test con sampling rate uguale a 100 hanno rivelato che questo tipo di campionamento è del tutto inefficace nel riconoscere un attacco in corso.

Infine vediamo nella figura 3.10 che tutti i test con sampling rate uguale a 2 e 4 sono stati in grado di vedere l'attacco, con una piccola differenza nell'attacco delle 20:00 in cui il test con sampling rate 2 non lo ha rilevato,

Sampling Rate	**# alert**	**% Alert**	**Traffico (MB)**	**% Traffico**
1 (On-Wire)	156	-	372	-
2	52	33.3%	124	33.3%
4	30	19%	74	19%
100	1	0.64%	3	0.8%

Tabella 3.2: Numero di alert al variare del sampling rate

probabilmente dovuto al susseguirsi in quel lasso di tempo piccolo di una scelta poco fortunata su quale pacchetto su 2 campionare.

Questo solleva una ulteriore domanda, ovvero è possibile evadere un IDS basato su packet sampling utilizzando tecniche statistiche per mantenere l'attacco sotto una certa soglia? (sviluppi futuri)

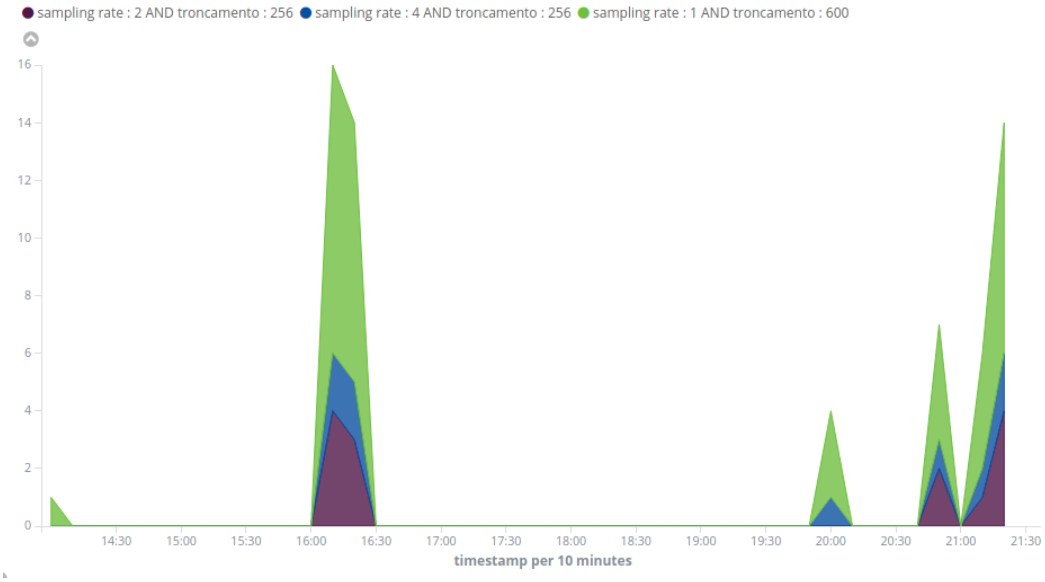

Figura 3.10: Numero di Alert al variare del sampling rate nel tempo

Sampling rate	# Alert	%
1 (On-Wire)	92055	-
2	3363	3.65%
4	995	0.1%

Tabella 3.3: Numero di alert al variare del sampling rate

3.2.3 CTU-13 Dataset

Infine analizziamo il comportamento del sistema di monitoraggio utilizzando il dataset CTU-13. Un dataset, come descritto nel capitolo precedente formato da traffico derivante da botnets. Si è scelto di testare, per motivi di scarsità di spazio su disco, soltanto il traffico della botnet 4, che contiene un attacco DDoS.

I risultati del tutto simili a quelli dei test con il dataset DARPA per cui riportiamo per brevità solo una tabella (3.3) con i valori del tasso di rilevamento a variare del sampling rate. Quello che è interessante di questo dataset è la capacità di rilevare che un attacco sia in corso. Sappiamo che gli attacchi DDoS per loro natura tendono a inviare alla macchina target una grande quantità di traffico in un periodo di tempo più o meno lungo. Questo tipo di attacco quindi si presta molto ad essere rilevato con tecnologie come quella di sFlow, poichè anche pur essendoci una parte di traffico che passa inosservata per via degli effetti del campionamento, ce ne sarà un'altra parte che invece verrà analizzata portando alla rilevazione dell'attacco. In figura 3.11 riportiamo il numero di alert al variare del campionamento in cui si nota proprio il comportamento descritto sopra.

Sia i test com sampling rate 2 che quelli con 4 hanno portato ad una rilevazione efficace dell'attacco. La grande differenza in numero che è possibile osservare tra questi ultimi e i test effettuati On-Wire non è da ritenersi particolarmente rilevante ai fini della valutazione. Infatti nelle regole che sono state scritte per testare questo dataset si è volutamente omesso un limite superiore agli alert, come invece si fa per questo tipo di attacco per evitare

un flood di alerts. Notiamo invece che non sono presenti alert nei test con sampling rate 100, contrariamente da come ci aspettavamo per questo tipo di attacco, pur accorciando il threshold di rilevamento all'interno della regola non si è riusciti a far scattare l'allarme di un DDoS in corso.

```
alert udp any any -> $HOME_NET 161 (msg:"ET DOS CTU-13 Botnet";
content:"|cb f5 ab 86 05|"; depth:5; threshold: type threshold,
track by_dst, seconds 90, count 3; classtype:bad-unknown;
sid:2016016; rev:8; metadata:created_at 2018-02,
updated_at 2018-02;)
```

Questo significa che pur avendo un *burst* di pacchetti malevoli non si sono rilevati neanche 3 pacchetti nell'arco di 90 secondi con quella signature descritta sopra, risultato alquanto insoddisfacente.

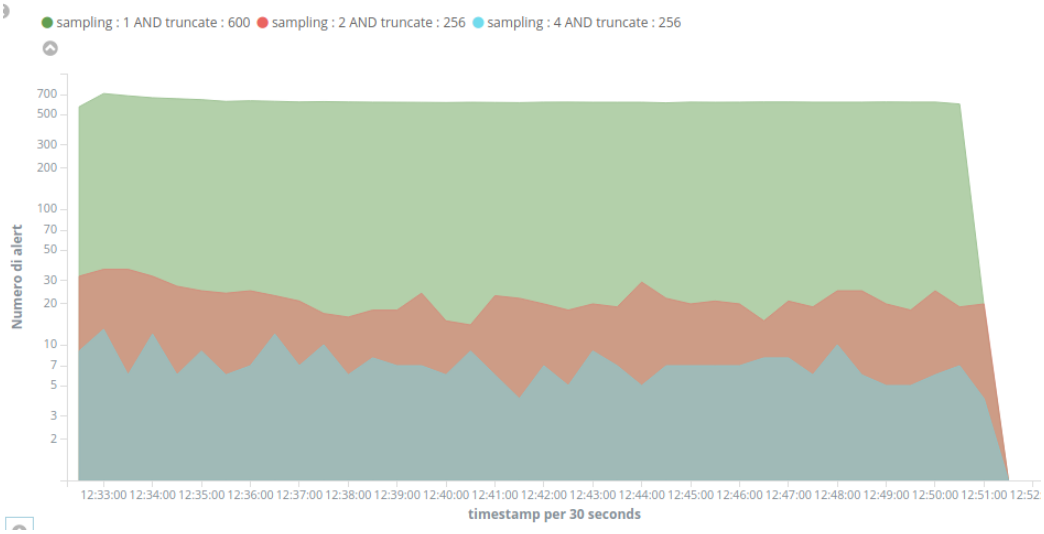

Figura 3.11: Numero di Alert al variare del sampling rate nel tempo

OLD OLD OLD Si è osservato il comportamento dell'IDS Suricata in due scenari diversi, uno in cui sono state utilizzate solo le regole di default e uno in cui si sono utilizzate solo le regole appositamente sviluppate per quel dataset [33].

Risultati con regole di default

Nella figura 3.1 è possible osservare il numero di alert scattati al variare dei parametri sul *sampling rate* per qualunque tipo di troncamento. Il numero degli alert con sampling rate [1]: 1 (On-Wire) è significativamente più alto di quello degli altri e si attesta intorno a 52459 mentre quelli per sampling : 2 e 4 si attestano intorno a 2800 (circa il 5%) e 2200(circa il 4%) rispettivamente. Quello con sampling : 100 si è rivelato invece altamente inefficace sollevando solo 640 attacchi

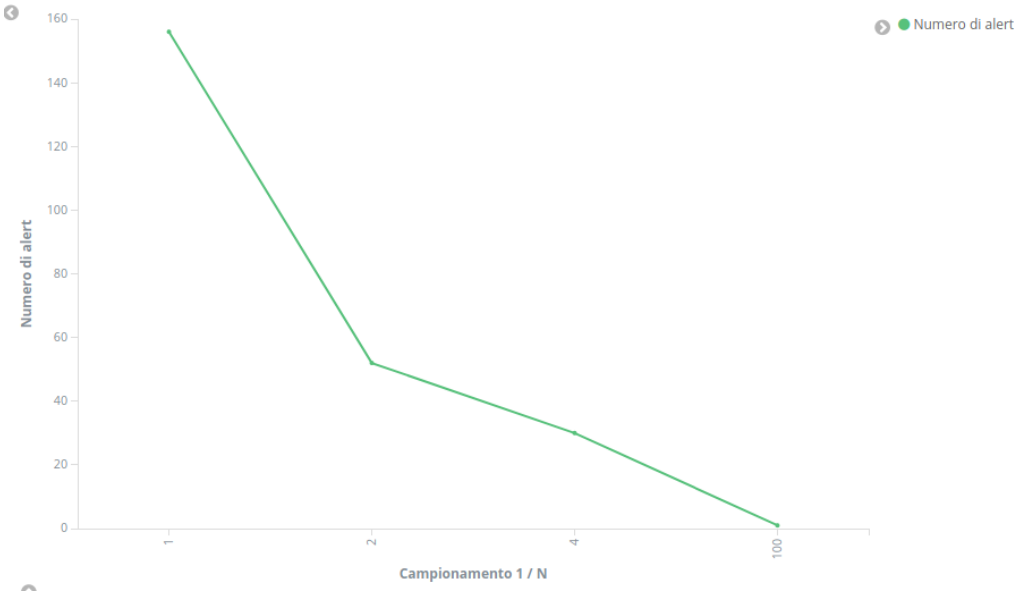

Figura 3.12: Numero di Alert al variare del sampling rate

[1]D'ora in avanti abbreviato con "sampling"

Campionamento 1 / N	Numero di alert	%
1 (On-Wire)	52459	-
2	2891	5,5%
4	2233	4,2%
100	640	1,2%

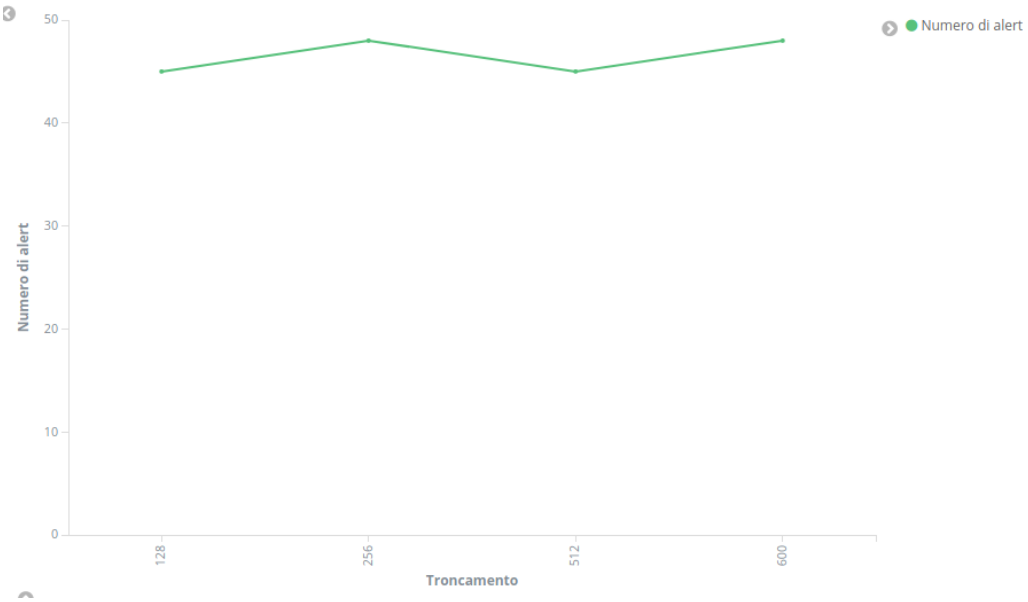

Figura 3.13: Numero di Alert al variare del troncamento

Nella figura 3.2 osserviamo lo stesso il numero di alert, al variare del troncamento. Qui notiamo un comportamento insolito, ad un troncamento effettuato a 128 byte corrisponde un notevole aumento del numero di alert, andando addirittura a superare quello On-Wire. Questo ci anticipa che quel particolare tipo di troncamento, per le regole di default installate potrebbe essere un punto critico. Negli altri casi invece (256 e 512) vi è una differenza del solo 15%, compatibile con il fatto che la dimensione del pacchetto nel caso di 512 permetterebbe di rilevare un maggior numero di attacchi.

Troncamento (Bytes)	Numero di alert	%
128	39194	400%
256	4473	50%
512	5735	65 %
600 (On-Wire)	8821	-

Come è possibile notare nella figura 3.3 il comportamento anomalo del troncamento a 128 byte è dovuto al fatto che si generano un altissimo numero di falsi positivi. Andando ad indagare si è scoperto che questi corrispondono nella maggior parte dei casi alle signatures *"SURICATA SMTP no server welcome message"* e *"SURICATA DNS malformed request data"* ,ovvero due signature che operano Stateful Analysis e che come si era detto nel capitolo precedente non sarebbero state utilizzabili in questo sistema di monitoraggio.

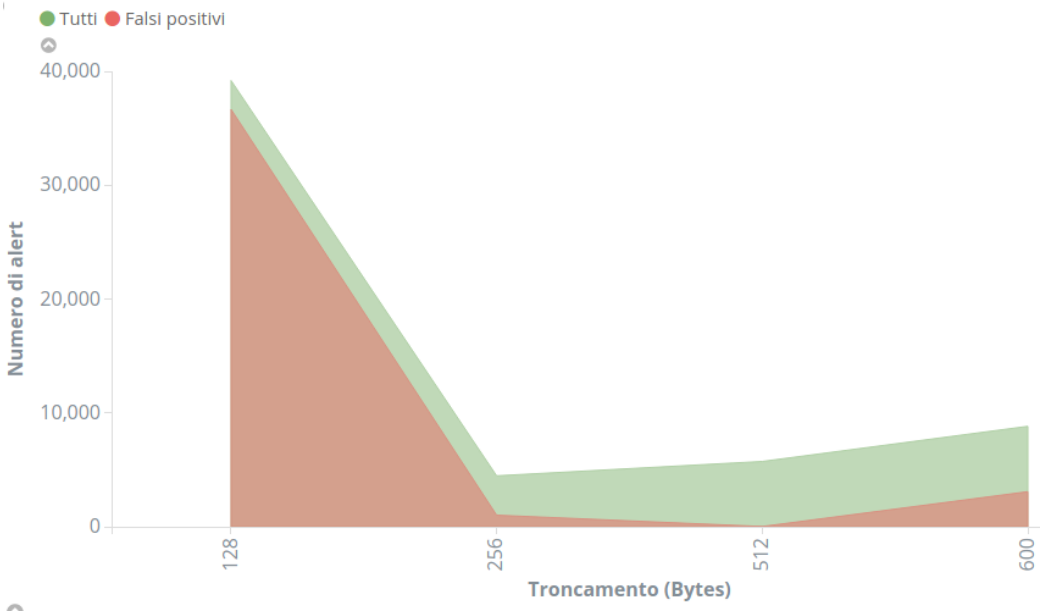

Figura 3.14: Numero dei falsi positivi al variare del troncamento

Si riportano ora due grafici raffiguranti il numero di signature univoche rilevate al variare del campionamento e del troncamento.

Conclusioni

Appendice A

Prima Appendice

Appendice B

Seconda Appendice

Bibliografia

[1] RFC 3176 InMon Corporation's sFlow for monitoring traffic in Switched and Routed Networks

[2] A survery on Network Security Monitoring System, Ibrahim Ghafir, Vaclav Prenosil, Jakub Svoboda, Mohammad Hammoudeh, 2016 4th International Conference on Future Internet of Things and Cloud Workshops

[3] Traffic Monitoring with Packet-Based Sampling for Defense against Security Threats, Joseph Reves ans Sonia Panchen

[4] "NIST â Guide to Intrusion Detection and Prevention Systems (IDPS)" (PDF). February 2007.

[5] https://ossec.github.io/

[6] https://github.com/OISF/suricata

[7] https://www.snort.org/

[8] https://www.bro.org

[9] https://www.ntop.org/products/packet-capture/pf_ring/

[10] sFlow, I Can Feel Your Traffic, Elisa Jasinska, Amsterdam Internet Exchange

[11] Reti di Calcolatori, Larry Peterson, Bruce Davie, Apogeo, 2012

[12] Hofstede, Rick; Celeda, Pavel; Trammell, Brian; Drago, Idilio; Sadre, Ramin; Sperotto, Anna; Pras, Aiko. "Flow Monitoring Explained: From Packet Capture to Data Analysis with NetFlow and IPFIX". IEEE Communications Surveys Tutorials. IEEE Communications Society

[13] http://blog.sflow.com/2009/05/scalability-and-accuracy-of-packet.html

[14] Flow-based intrusion detection: Techniquesand challenges, Muhammad Fahad Umer, Muhammad Sher, Yaxin Bi. Computers & Security 70 (2017) 238-254

[15] Intrusion detection system: A comprehensive review, Hung-JenLiao, Chun-HungRichard Lin,Ying-ChihLin, Kuang-YuanTung

[16] https://www.elastic.co/

[17] https://db-engines.com/en/ranking/search+engine

[18] SCADA STATISTICS MONITORING USING THE Elastic Stack(Elasticsearch, Logstash, Kibana), James Hamilton, Brad Schofield, Manuel Gonzalez Berges, Jean-Charles TournierCERN, Geneva, Switzerland

[19] https://www.virtualbox.org/

[20] https://rules.emergingthreats.net/

[21] Quantitative Analysis of Intrusion Detection Systems: Snort and Suricata, Joshua S. White , Thomas T. Fitzsimmons , Jeanna N. Matthews

[22] https://www.ll.mit.edu/ideval/data/1999data.html

[23] https://mcfp.weebly.com/the-ctu-13-dataset-a-labeled-dataset-with-botnet-normal-and-background-traffic.html

[24] https://ictf.cs.ucsb.edu/pages/the-2010-ictf.html

[25] http://www.tcpdump.org/

[26] https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-for-indexing-speed.html

[27] https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-update-settings.html

[28] https://docs.oracle.com/cd/E26217_01/E26796/html/qs-create-vm.html

[29] https://www.debian.org/releases/jessie/amd64/

[30] Raymond, Eric S. (2003-09-23). "Basics of the Unix Philosophy". The Art of Unix Programming. Addison-Wesley Professional

[31] https://www.elastic.co/guide/en/elasticsearch/guide/current/inverted-index.html

[32] https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Payload_keywords

[33] Khamphakdee, Nattawat Benjamas, Nunnapus Saiyod, Saiyan. (2014). Improving Intrusion Detection System Based on Snort Rules for Network Probe Attack Detection. 2014 2nd International Conference on Information and Communication Technology, ICoICT 2014. 10.1109/ICoICT.2014.6914042.

[34] https://www.cybrary.it/2015/05/summarizing-the-five-phases-of-penetration-testing/

Ringraziamenti