Corso di Laurea in Informatica

TESI DI LAUREA

OS fingerprinting passivo tramite clustering gerarchico

Relatore:
Luca Deri

Candidato:
Rodrigo Casella

ANNO ACCADEMICO 2022/2023

Indice

Desidero ringraziare innanzitutto il professor Luca Deri, il mio relatore, per il tempo e il supporto a me dedicatomi. Inoltre, vorrei dire grazie alla mia famiglia e alle persone a me più care per avermi accompagnato in questo percorso della mia vita.

1. Introduzione

L'Operating System (OS) fingerprinting permette l'identificazione remota del sistema operativo di un dispositivo tramite l'analisi del traffico di rete da esso generato; questo rende possibile l'individuazione di host che compromettono la sicurezza di una rete.

Il riconoscimento di un sistema operativo avviene tramite la costruzione di *firme* (o *fingerprints/signatures*), ovvero di pattern presenti nei pacchetti di rete generati da un determinato host. Gli strumenti di fingerprinting tradizionali fanno uso di basi di dati contenenti *firme* di vari sistemi operativi che, per rimanere efficienti facendo fronte alla continua evoluzione del mondo delle reti, devono essere aggiornate in modo costante. Le metodologie di *fingerprinting* utilizzate si dividono in due famiglie: attive e passive.

Il fingerprinting attivo consiste nell'inviare alla macchina bersaglio uno o più messaggi detti *sonda* per stimolare l'invio di messaggi di risposta da analizzare per ricavarne una firma. Questo metodo permette di ottenere risultati precisi data la possibilità di *forgiare* pacchetti sonda strutturati per indurre l'invio di risposte specifiche contenenti le informazioni necessarie per l'identificazione del sistema operativo direttamente dal bersaglio; purtroppo questa tecnica risulta molto invasiva e facilmente individuabile dalla macchina vittima o dal firewall di una rete i quali potranno bloccare ogni ulteriore tentativo d'inviare pacchetti *sonda*.

Il fingerprinting passivo, a differenza dell'attivo, è meno rilevabile da host e sistemi di sicurezza poiché si limita allo *sniffing* (ossia all'intercettazione) del traffico di rete del target, ma senza ottenere lo stesso livello di precisione nell'identificazione del sistema operativo delle tecniche più invasive, vista l'impossibilità di forgiare *messaggi sonda* personalizzati. Per identificare l'host del bersaglio è quindi necessario ispezionarne i pacchetti di rete alla ricerca di parametri utili alla costruzione di una firma.

La diffusione negli ultimi decenni di protocolli di rete crittografici che permettono di stabilire una connessione sicura, oscurando parte del traffico di rete in una comunicazione bidirezionale tra due host, ha reso obsolete le tecniche più affidabili di fingerprinting.

Di conseguenza è necessaria l'esplorazione di nuovi metodi d'identificazione remota nella prospettiva di un futuro privo di traffico in chiaro.

1.1 Obiettivo

Software come Mercury [8] utilizzano le informazioni del protocollo crittografico di comunicazione Transport Layer Security (***TLS***) per generare firme, le quali assieme a un contesto di destinazione permettono l'identificazione di processi, malware e sistemi operativi. Sebbene questa metodologia permetta di stabilire l'OS di un host aggirando le limitazioni delle connessioni sicure, la mancanza di un database di firme sempre aggiornato pregiudica l'intero processo d'identificazione.

Altri approcci più recenti [2, 3, 15, 20] fanno uso di tecniche di apprendimento automatico (in inglese *machine learing* o *ML*) supervisionato che, a differenza dei metodi tradizionali, dispongono della capacità di classificare firme non presenti nei database. La medesima, così come le tecniche *signature based*, soffre però di un limite; in quanto le firme di un sistema operativo si evolvono velocemente, tale che anche il modello di ML più performante deve essere sottoposto a un allenamento continuo per rimanere preciso e affidabile.

L'obiettivo dell'elaborato proposto difatti auspica al possibile sviluppo di un metodo di *fingerprinting passivo* in grado d'identificare il sistema operativo di un host integrando le informazioni veicolate dal protocollo **TLS** [24] con quelle del protocollo **TCP** [9], senza l'ausilio di tecniche di *machine learning supervisionato*.

2. Stato dell'arte

Le metodologie di *fingerprinting* proposte si basano su tool e tecniche di tipo *passivo* che al giorno d'oggi risultano obsolete poiché basate su protocolli superati, necessitano di database aggiornati o non sono facilmente riproducibili. Come tratteremo nei paragrafi successivi, questi strumenti offrono nozioni e strategie utili alla ricerca di nuovi metodi per l'identificazione dei sistemi operativi.

2.1 TCP/IP stack fingerprinting

Il **TCP/IP stack fingerprinting** è un metodo per inferire le caratteristiche di una implementazione dello **stack TCP/IP** tramite l'ispezione degli header IP [1] e TCP. Ciò risulta possibile grazie alla presenza di alcuni parametri dei protocolli in questione le cui definizioni sono lasciate all'implementazione.

2.1.1 p0f

P0f [19] è un tool che utilizza meccanismi di *fingerprinting passivo* per identificare host che comunicano tramite il protocollo *TCP/IP*. Per questo tipo di comunicazione p0f raccoglie i parametri degli *header IPv4* e *TCP* dai pacchetti di *SYN* e *SYN-ACK* in un *handshake TCP a tre vie*, rispettivamente il primo pacchetto generato da un host che vuole instaurare una connessione TCP e il primo pacchetto di risposta inviato dalla macchina che riceve la richiesta di comunicazione.

Le firme TCP presenti nel database di p0f sono composte dai seguenti parametri:

- ver: versione IP utilizzata, può assumere i valori 4, 6 o entrambi ('*')
- ittl: valore iniziale del campo *Time To Live* nell'header IP
- olen: lunghezza delle opzioni IPv4
- mss: valore del *maximum segment size* denotato nelle opzioni TCP, se presente

- wsize: valore della *recive window* usata per il controllo del flusso
- scale: valore che se presente nelle opzioni TCP denota il fattore moltiplicativo della *recive window*
- olayout: lista ordinata delle opzioni TCP[ ]
- quirks: parametri normalmente non settati o modificati negli header TCP e IP: *"don't fragment"* bit settato nell'header IPv4, valore del timestamp TCP uguale 0, etc...
- pclass: indica la presenza o meno di un payload nel pacchetto analizzato, normalmente i pacchetti *SYN* e *SYN-ACK* non hanno payload.

ver	ittl	olen	mss	wsize	scale	olayout	quirks	pclass
*	64	0	*	mss*10	4	mss,sok,ts,nop,ws	df,id+	0
*	64	0	*	mss*12	0	mss		0
*	64	0	16376	mss*2	2	mss,sok,ts,nop,ws	df,id+	0
*	128	0	*	65535	2	mss,nop,ws,nop,nop,sok	df,id+	0

Tabella 2.1: Firme TCP/IP utilizzate da p0f

In seguito una o più firme vengono associate a un sistema operativo attraverso una *label* composta da:

- type: 's' se le *signatures* non possono essere migliorate, 'g' se sono generiche
- class: distingue le firme dei sistemi operativi da quelle di applicazioni
- name: nome che identifica la firma, solitamente il nome del sistema operativo
- flavor: qualunque cosa aiuti a rendere ancora più specifica la firma (e.g. Windows 8/8.1)

Il processo d'identificazione utilizzato da p0f è semplice: dopo aver raccolto i parametri necessari alla costruzione di una firma da una sessione TCP viene cercata una corrispondenza nel database di *signatures*. Nel caso in cui vi sia riscontro p0f risponde comunicando

la *label* associata alla firma osservata contenente il nome del sistema operativo rilevato. Purtroppo la base di dati di p0f non è aggiornata ormai da anni e questo lo ha reso obsoleto.

2.2 Application Layer Fingerprinting

Un'altra tecnica di identificazione del sistema operativo di un host consiste nel **Application Layer Fingerprinting** la quale analizza i parametri presenti nel *payload* dei protocolli applicativi.

2.2.1 HTTP Fingerprinting

L'**HTTP Fingerprinting** è di gran lunga il metodo più semplice per rilevare l'OS di un client: ispezionando la **User-agent string** [12] di una richiesta HTTP è possibile identificare in modo istantaneo il sistema operativo.

User-agent string	OS
(Linux; Android 6.0.1; Nexus 5X Build/MMB29P)	Android
(Windows NT 10.0; Win64; x64)	Windows
(Macintosh; Intel Mac OS X 10_10_1)	Mac OS
(iPhone; CPU iPhone OS 16_0 like Mac OS X)	iOS
(X11; Ubuntu; Linux x86_64; rv:109.0)	Linux

Tabella 2.2: User-agent strings con OS corrispondente.

Sfortunatamente la diffusione capillare del protocollo applicativo **HTTPS** (estensione dell'HTTP combinato con protocolli crittografici come SSL o TLS) nel web ha reso questa tecnica di *fingerprinting* inadoperabile: le richieste e le risposte sono criptate, quindi visibili esclusivamente ai client e server HTTP comunicanti. Inoltre, la possibilità di modificare intenzionalmente la stringa dello *User-agent* prima d'inviare una richiesta HTTP è un fattore che influisce sull'attendibilità di questo parametro.

2.2.2 Mercury

Mercury [8] è uno strumento per monitorare il traffico di rete, la cui funzione principale è quella di produrre *firme* per vari protocolli di rete (TLS, DTLS, SSH, HTTP, TCP). Il tool è anche in grado d'identificare malware, processi indesiderati e sistemi operativi basandosi sulle *fingerprints* prodotte e un contesto di destinazione.

Le firme protocollo **TLS** sono di particolare interesse poiché permettono d'identificare comunicazione crittografate. Esse sono formate a partire da pacchetti contenenti un record **Client Hello**, ovvero il primo messaggio inviato da un client durante la fase di negoziazione detta **TLS Handshake**. Nel TLS Client Hello sono presenti informazioni come: la versione TLS utilizzata, un numero random, un ID di sessione nel caso di una di ri-connessione, una lista di cifrai e metodi di compressione da suggerire al server e infine una lista di estensioni TLS.

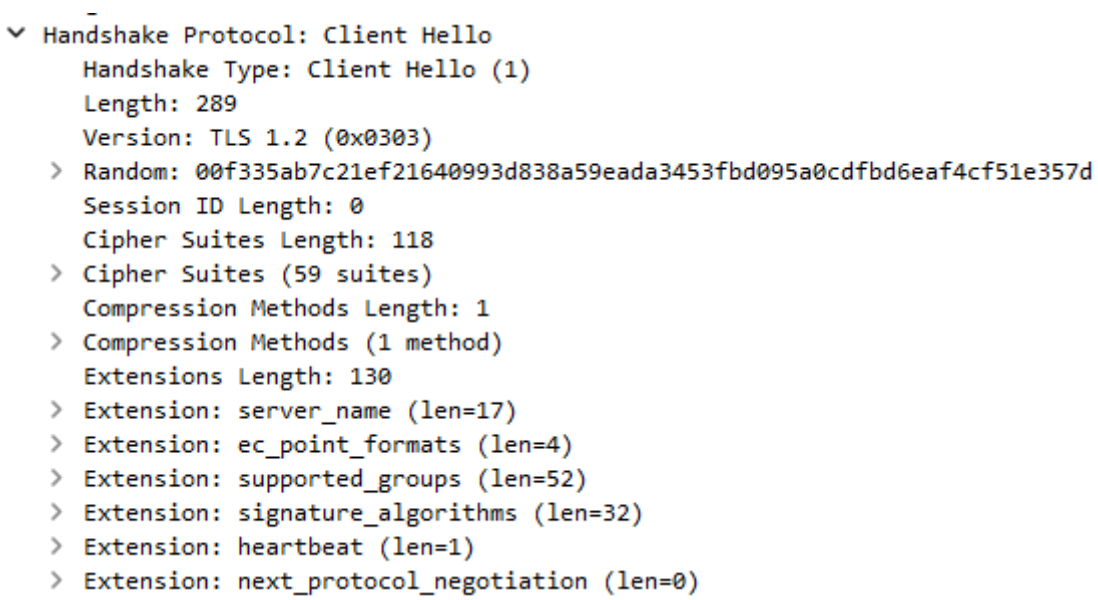

Figura 2.1: Struttura di un TLS Client Hello

Mercury utilizza: il numero della versione TLS supportata, la lista dei cifrari e la lista delle estensioni per la costruzione della firma TLS, rimuovendo da gli ultimi due elementi i valori appartenenti all'estensione TLS GREASE.

GREASE [5] (**G**enerate **R**andom **E**xtensions **A**nd **S**ustain **E**xtensibility) è un meccanismo svliuppato per la prevenzione di errori dovuti all'estensibilità dell'ecosistema TLS. Solitamente un client espone i cifrari e le estensioni che è in grado di supportare. D'altro

canto, il server dovrebbe ignorare le componenti non supportate e operare con quelle da lui conosciute. Poiché se il server non è in grado di gestire parametri ignoti si va incontro alla perdita d'*interoperabilità* tra sistemi. Per evitare questo tipo di problema *GREASE* introduce valori che una corretta implementazione TLS dovrebbe ignorare, quindi host che non dispongono di questa capacità non potranno operare, rilevando prontamente problemi d'interoperabilità.

Raccolti i parametri necessari alla costruzione di una firma Mercury utilizza un sistema di **Fingerprinting con Contensto di Destinazione** [4] (Fingerprinting with Destination Context o **FDC**) per identificare processi software che generano una determinata firma **TLS**. L'FDC prende in input una firma TLS e un contesto di destinazione, il quale consiste in: un IP di destinazione, una porta TCP di destinazione e il campo *server_name* dalle estensioni del TLS Client Hello. La funzione restituisce in output varie informazioni relative alla firma analizzata tra cui: lo stato, informazioni sul processo, un punteggio di probabilità che rappresenta la confidenza del classificatore sulla correttezza delle informazioni del processo, la probabilità che esso sia un malware e una lista dei possibili sistemi operativi a lui associati. Nello specifico lo stato di una firma è **labeled** se è presente nel database, **unlabeled** se è stata osservata, ma non sono disponibili informazioni su di essa, oppure **randomized** se non è stata mai osservata.

L'implementazione dell'FDC di Mercury utilizza un classificatore **Weighted Naive Bayes** [4] (WNB) per analizzare il contesto di destinazione e una firma per selezionare il classificatore WNB. Più semplicemente la firma è utilizzata per selezionare un classificatore da applicare sul contesto di destinazione per selezionare il processo più probabile dal database. Un Classificatore Bayesiano Naive stima le probabilità applicando il teorema di Bayes assumendo "ingenuamente" (*naive*) l'indipendenza condizionale delle caratteristiche (o *features*) dei dati. Inoltre, un classificatore WNB assegna dei pesi a ogni feature, i quali potranno essere aggiustati durante la fase di training per aumentare la precisione della classificazione.

Infine, prima dell'avvio del classificatore WNB, il contesto di destinazione è analizzato per trovarne la classe di equivalenza; questo permette di generalizzare gli indirizzi IP tramite gli **Autonomous System Numbers** [13] e i server_names tramite **domini DNS** per ridurre il numero di voci nel dizionario delle firme.

2.3 ML based

Nell'ultimo periodo varie tecniche di **Machine Learning** (ML) sono state impiegate per contrastare la necessità di creare e mantenere un database di *signatures*. Il Machine Learning permette anche di utilizzare e selezionare automaticamente un numero maggiore di parametri per l'identificazione dei sistemi operativi.

Muehlstein et al. [20] hanno proposto un metodo passivo per identificare tuple ⟨OS, Browser, Application⟩ con tecniche di learning supervisionato. Due set di features contenenti parametri dei protocolli TCP/IP e TLS sono impiegati per allenare un modello di **Support Vector Machine** (SVM) che si avvale di una **Radial Basis Function** (RBF) come ***kernel***.Questa tecnica raggiunge una grande accuratezza sia nell'identificazione della tupla che del sistema operativo in specifico.

Laštovička et al. [15] hanno realizzato due metodi in grado d'identificare il sistema operativo tramite i protocolli TCP/IP e TLS; entrambe le metodologie si avvalgono di un **Decision Tree** per classificare passivamente il traffico di rete. Nel primo caso, per ogni flusso HTTP i parametri TCP/IP e lo *User-Agent* vengono accoppiati per la creazione del dataset iniziale. Per il TLS, invece, vengono accoppiati lo User-Agent e i parametri del TLS Client Hello da richieste provenienti dallo stesso dispositivo. I risultati di questo esperimento hanno mostrato una maggiore precisione nel metodo TCP/IP.

Invece, Aksoy et al. [2] hanno impiegato un algoritmo genetico per la selezione automatica di features dall'header di pacchetti TCP/IP. Successivamente vengono allenati quattro classificatori utilizzando una gerarchia a due livelli. Il primo classificatore impara a identificare la famiglia del sistema operativo di un pacchetto (Linux, Mac, Windows), mentre i classificatori rimanenti imparano a specificare la versione del sistema operativo per ogni genere di OS. In breve, il primo livello si occupa di stabilire se il pacchetto appartiene a Linux, Mac o Windows, dopodiché i classificatori di secondo livello ricevono il pacchetto appartenente alla famiglia su cui sono stati allenati e ne specificato la versione (Windwos XP, 7, 10, etc...).

Anderson et al. [3] hanno introdotto una nozione di distribuzione di probabilità a priori e una regola di massima verosimiglianza per l'assegnazione di un sistema operativo a una

determinata firma. Per esempio, nella maggioranza delle reti la probabilità che un laptop abbia Mac OS è maggiore rispetto a quella di avere FreeBSD, quindi Mac OS verrà scelto dalla regola di massima verosimiglianza. In seguito, sono stati allenati tre classificatori impiegando separatamente parametri: TCP/IP, TLS e HTTP. I test sono stati eseguiti sia su sessioni singole di traffico, che in finestre di 60 minuti ottenendo discreti risultati.

3. Soluzione proposta

Come già anticipato nei capitoli precedenti, l'obiettivo ultimo dell'elaborato verte allo sviluppo di un metodo di *fingerprinting passivo* basato sui protocolli **TCP/IP** e **TLS**. In primo luogo sarà dunque necessario stabilire i parametri utilizzabili per la costruzione delle **firme** di sistemi operativi. Successivamente verranno raccolti i dati necessari per testare e validare possibili strategie di *fingerprinting*. Per ultimo saranno presentati i risultati ottenuti insieme all'implementazione del progetto.

Figura 3.1: Workflow dello sviluppo del progetto

3.1 Firma

La **firma** (o *signature*) è un insieme d'informazioni che permette d'identificare in modo univoco un sistema operativo. Con il passare del tempo, le firme devono rimanere sufficientemente stabili e diversificate per continuare a riconoscere gli OS. Nel mondo del networking questo è possibile impiegando le informazioni veicolate dai protocolli di rete.

Dunque, nel lavoro proposto, la scelta dei parametri da utilizzare per la costruzione di una firma è stata influenzata da tool e studi precedenti, ma anche dalla necessità di utilizzare elementi non soggetti alla crittografia dei protocolli di rete.

Con queste premesse i segmenti **TCP-SYN** e i record **TLS ClientHello** sono ottimi candidati per la ricerca di parametri utili al *fingerprinting* di un sistema operativo. Oltre a esporre informazioni sull'implementazione di un host, i messaggi scambiati durante l'instaurazione di una connessione sono trasmessi in chiaro, per consentire alle due parti di concordare i parametri indispensabili alla creazione di una connessione sicura. Inoltre il protocollo TLS incapsula il proprio *payload* in segmenti TCP, ciò permette di raccogliere le informazioni necessarie alla costruzione di una firma con solo i primi pacchetti di una conversazione instaurata dalla macchina di cui vogliamo identificare l'OS.

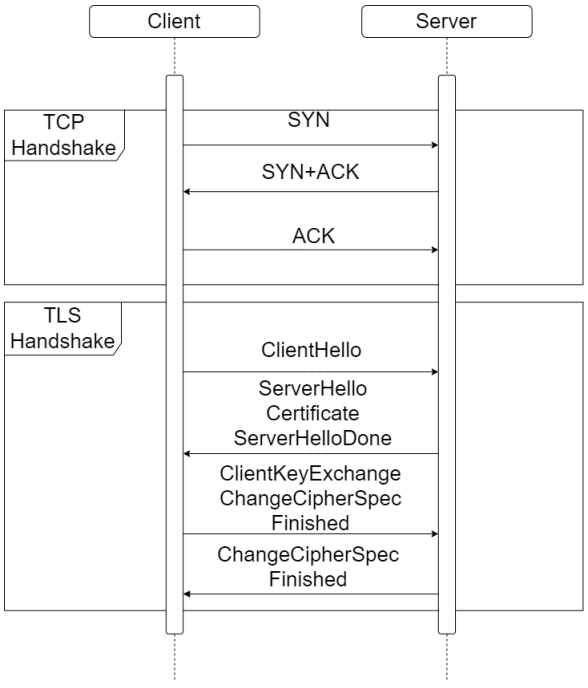

Figura 3.2: Processo di handshake completo tra client e server

3.1.1 Parametri TCP/IP

La prima parte della firma è composta da parametri presenti in un pacchetto in cui il segmento TPC ha il flag **SYN** è settato.

Parametro	**Valori possibili**
IP Time To Live	0-255
TCP Window Size	0-65535
TCP Maximum Segment Size	0-65535
TCP Window Scaling	0-14
Lista Opzioni TCP	

Tabella 3.1: Parametri TCP/IP considerati

La tabella 3.1 raffigura i parametri presi in considerazione per la costruzione della firma, in quanto questi campi sono utilizzati in molti sistemi di classificazione del traffico di rete [6, 19, 20]. Poiché gli RFC 791 [1] e 9293 [9] non ne specificano i valori da assegnare, le implementazioni dello stack di rete dei vari sistemi operativi presenteranno differenze in questi parametri, permettendone l'identificazione. Dunque macchine con il medesimo sistema operativo genereranno lo stesso set di parametri, viceversa host con OS differenti non presenteranno valori identici.

Va considerato che il valore del **TTL** presente nell'header IP potrebbe essere modificato durante la consegna del messaggio, poiché decresce ogni volta che un **router** (o un dispositivo di rete) inoltra il pacchetto verso un nuovo segmento di rete, ovvero esegue un **hop**. Comunemente al *TTL* sono assegnati valori come 64 o 128 ed è raro che un pacchetto di rete esegua più di 32-64 *hop*; per questo motivo il valore di questo parametro viene arrotondato alla potenza di due più vicina [16].

3.1.2 Parametri TLS

L'unica parte di traffico trasmesso in chiaro durante una connessione sicura tra due host, basata sul protocollo TLS, è la fase di negoziazione dei parametri crittografici detta **fase**

di handshake. In questo stadio il client inizia la conversazione inviando un messaggio di **ClientHello** in cui specifica: la versione TLS che supporta, un numero random, una lista di possibili cifrari in ordine di preferenza, metodi di compressione e una lista di estensioni supportate, può inoltre inviare un *Session ID* se viene negoziata una riconnessione.

TLS Client Hello		
TLS Version		
Random		Session ID
Ciphersuites Lenght		
Ciphersuites		
Compression Methods Lenght		
Compression Methods		
Extensions Length		
Extension Type	Length	Data
...		

Tabella 3.2: Formato record TLS ClientHello

Dal record del *ClientHello* è stato scelto di utilizzare le **ciphersuites** (lista dei cifrari) e i valori indicati dall'estensione **supported_groups**, in quanto forniscono una visione delle capacità crittografiche di un host [15]. Altri parametri invece sono stati scartati, poiché non rimangono stabili attraverso diverse sessioni dello stesso host, come la lista delle estensioni TLS, oppure non aggiungono abbastanza diversificazione, come `ec_point_format` che può contenere un solo valore [22].

Tuttavia, è importante sottolineare che le librerie che implementano il protocollo TLS operano nello spazio utente di un sistema operativo. Ciò significa che due host aventi lo stesso OS potrebbero disporre di librerie differenti evitando così di fornire informazioni che potrebbero essere utilizzate per identificare la piattaforma di un client. Nonostante ciò, i programmi tendono a essere ottimizzati per il sistema operativo su cui girano. Pertanto applicazioni multi piattaforma, come i *browser web*, possono presentare alcune differenze nella scelta di determinati parametri. Ad esempio, un browser web potrebbe utilizzare una

ciphersuite diversa su Windows rispetto a macOS o Linux per ottimizzare le prestazioni e la sicurezza [8].

3.1.3 Formalizzazione Firma

Avendo stabilito i parametri necessari alla formazione di una firma il prossimo passo sarà quello di formalizzare il processo di creazione di una firma.

Le firme sono estratte da **flussi**, ovvero sequenze unidirezionali di pacchetti di rete che condividono dei valori che ne definiscono l'unicità [14]. In questo contesto i valori che determineranno un flusso sono:

Indirizzo IP Sorgente
Porta TCP Sorgente
Indirizzo IP Destinazione
Porta TCP Destinazione

Per ogni flusso dai pacchetti **TPC/IP SYN** succeduti da un **TLS ClientHello** i parametri estratti permetteranno la costruzione della firma:

```
    signature = {
        ip_ttl,
        tcp_windows_size,
        tcp_mss,
        tcp_window_scaling,
        tcp_options_list,
        tls_ciphersuites,
        tls_supported_groups
        }

```

Listing 3.1: Struttura firma sistema operativo

3.2 Raccolta dei dati

La raccolta dei dati è stata la fase più complessa nello sviluppo di un metodo di fingerprinting. Per testare e la validare nuove strategie è necessario disporre di un **dataset** che contenga firme associate al sistema operativo di provenienza. Questo ci fornisce una *base empirica* (o **ground truth**) che ci aiuti a definire nuove regole per identificare un sistema operativo a partire da una *signature*.

Purtroppo, i dataset opensource per il fingerprinting sono datati o mancano dei parametri d'interesse. Altri invece, utilizzati da altri studi, non sono stati resi pubblici. Pertanto è stato necessario lo sviluppo di un programma di cattura in grado di elaborare pacchetti di rete per estrarne i parametri necessari alla costruzione di firme da utilizzare durante la fase di test.

3.2.1 Script di cattura

L'obiettivo dello script di cattura è quello di monitorare una macchina che riceve richieste di connessione da host remoti al fine d'intercettare pacchetti **TCP/IP SYN** o messaggi **TLS ClientHello**.

Quando viene rilevato un pacchetto TCP-SYN, lo script estrae diversi parametri, tra cui il campo TTL dall'header IP, la dimensione della finestra (Window Size), la lista delle opzioni TCP, recuperando anche il Maximum Segment Size e il Window Scaling, se presenti. D'altra parte, dai messaggi TLS ClientHello, vengono estratte le *ciphersuites* e i *supported_groups*. In entrambi i casi, i parametri vengono registrati in un file di log assieme all'indirizzo IP sorgente.

```
66.249.70.166: 122,65535,[(2, 1412), (4, 0), (8, 891833320311095296), (1, 0), (3, 8)]
66.249.70.166: [60138, 4865, 4866, 4867, 49195, 49199, 52393, 52392,49196, 49200, 49161, 49171,
 49162, 49172, 156, 157, 47, 53, 10],[35466, 29, 23, 24]
68.183.225.220: 48,64240,[(2, 1460), (4, 0), (8, 9016726484340965376),(1, 0), (3, 7)]
68.183.225.220: [49195, 49199, 49196, 49200, 52393, 52392, 49161, 49171, 49162, 49172, 156, 157,
 47, 53, 49170, 10, 4865, 4866, 4867],[29, 23, 24, 25]
```

Listing 3.2: Esempio file di log

I file di log risultanti possono quindi essere utilizzati per la formazione delle firme, ma ciò non basta alla costruzione del dataset poiché non siamo in grado di stabilire il sistema operativo del mittente dei messaggi. Pertanto, i file di log devono essere affiancati a informazioni aggiuntive per ottenere una visione completa della natura del client.

Gli endpoint **HTTPS** rappresentano un canale privilegiato per raccogliere questo tipo di dati. Avendo accesso alle richieste HTTPS in chiaro è possibile ricavare le informazioni sul sistema operativo in uso dal client tramite la stringa dello **User-Agent**. Questo

campo è spesso inserito all'intero delle richieste dai browser web per identificarsi e fornire una compatibilità adeguata. L'integrazione dei log delle richieste HTTPS con i file prodotti dallo script di cattura produce un dataset adeguato per lo studio di processi di *fingerprinting*.

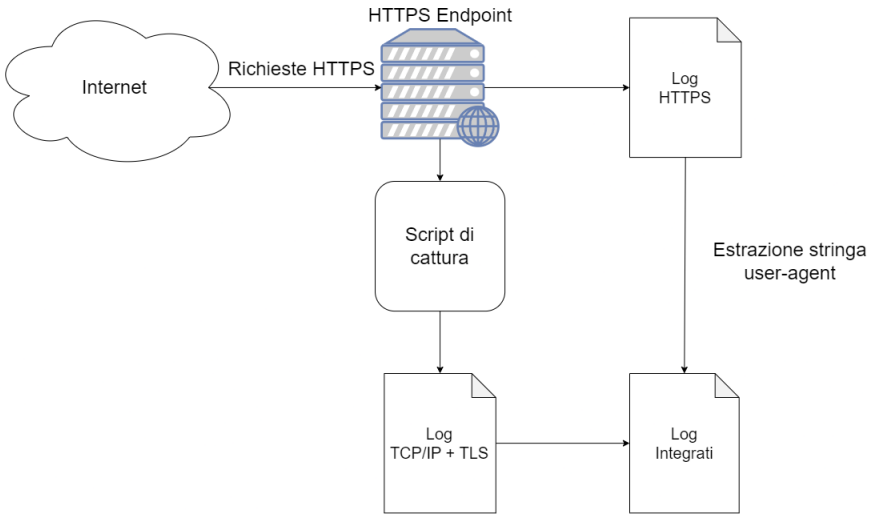

Figura 3.3: Processo di raccolta e integrazione dei file di log

3.2.2 Ambiente di cattura

Lo script di cattura è stato fatto girare sul *web server* di ntop.org così da avere accesso ai log delle richieste HTTPS ricevute dal server. Questo ha reso possibile l'unione dei dati ricavati da entrambi i tipi di log assicurando informazioni affidabili per il dataset.

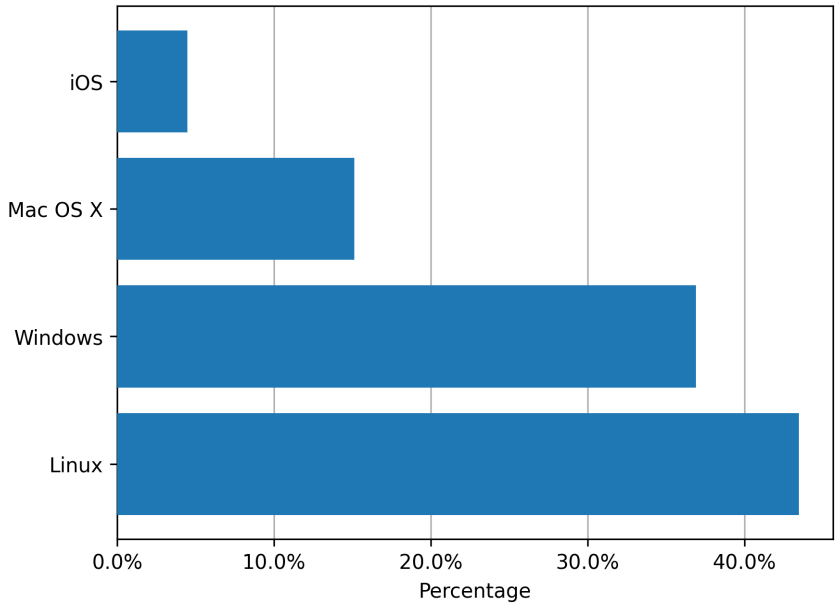

Figura 3.4: Frequenze percentuali dei sistemi operativi osservati

Dopo sedici ore di cattura sono stati raccolti più di 82000 frame e circa 63800 richieste HTTPS. Infine, dopo aver pulito i file di log da pacchetti malformati e aver scartato le *requests* inviate da indirizzi IPv6, il dataset risultante è arrivato a contenere più di 2400 firme.

Le richieste provenienti da hosts IPv6 no sono impiegate poiché questa versione del protocollo Internet introduce cambiamenti che possono influenzare i protocolli superiori: l'*Hop Limit*, a differenza del TTL, non costringe a rispettare il tempo massimo di vita di un pacchetto, quindi i livelli superiori devono essere modificati per gestire questo nuovo meccanismo. La dimensione dei pacchetti IPv6, maggiore rispetto al suo predecessore, comporta anche una modifica nel calcolo dei checksum e della riduzione della dimensione massima del payload dei protocolli di livello superiore. Queste considerazioni, combinate con l'elevata predominanza del traffico IPv4, rendono le richieste IPv6 trascurabili.

La figura 3.4 mostra la frequenza percentuale dei sistemi operativi dei client che hanno contattato il web server durante il periodo di cattura.

3.3 Strategia proposta

La metodologia di *fingerprinting* proposta vuole rispondere alla seguente domanda:

"Firme simili sono generate dal medesimo sistema operativo?"

Per rispondere a tale quesito in primis è necessario analizzare la struttura delle firme, stabilendo se insiemi composti da elementi omogenei coincidano con una determinata famiglia di sistemi operativi.

Tramite l'impiego di algoritmi di ***clustering*** [32] è possibile aggregare un set di oggetti in modo che elementi appartenenti allo stesso gruppo (chiamato **cluster**) siano più *simili* tra loro che a componenti di altri gruppi.

3.3.1 Considerazioni sulla struttura delle firme

La scelta di un appropriato algoritmo di clustering dipende dalla struttura dei data che andiamo ad analizzare.

Il formato delle firme proposto 3.1 consiste in quattro valori interi positivi e tre liste ordinate d'interi positivi. Il primo pensiero sarebbe quello di considerare i primi quattro parametri come dati di tipo numerico, ma in questo caso non rappresentano una quantità o una misura, bensì una qualità o una caratteristica della firma. Perciò tutti e sette i parametri (o ***features***) che compongono una firma sono dati di tipo categorico.

Una volta definita la natura dei parametri che costituiscono una firma, resta da scegliere una funzione di distanza o similarità. Questo criterio sarà fondamentale per valutare quanto due firme siano simili durante il processo di clustering.

Dovendo confrontare classi di dati categorici, la *distanza di Hamming* [30] è ideale per catturare le differenze tra due set di features, perché ne conta la porzione di componenti discordi. Per esempio la distanza tra la firma:

64|65535|1380|6|2-1-3-1-1-8-4-0-0|4865-4866-4867-49195-49199|29-23-24

e

128|65535|1380|9|2-1-3-4-8|4865-4866-4867-49195-49199|29-23-24

risulta pari a **3**, poiché le componenti del TTL, Window scaling e la lista delle opzioni TCP tra le due firme non combaciano.

3.3.2 Clustering

Il termine **cluster** non può essere definito in modo preciso, per cui esistono diversi algoritmi di *clustering*. Nonostante ciò, l'obiettivo di tutti i metodi di clustering è quello di raggruppare elementi per identificare pattern o strutture nascoste all'interno dei dati. Per questa ragione esistono diversi modelli di cluster e per ognuno di essi esiste un algoritmo differente. Comprendere le differenze tra i vari modelli di cluster è indispensabile per capire le diverse strategie impiegate dai vari algoritmi.

- **Clustering gerarchico** [31]: costruisce una gerarchia unendo o dividendo progressivamente gli elementi in base alla loro similarità o distanza. È di tipo agglomerativo se ciascuna componente iniziale viene considerata come un cluster, per poi unire i gruppi più simili/vicini a ogni passo finché non ne rimane solo uno.

  In contrapposizione, il clustering gerarchico divisivo inserisce tutti i dati in un unico cluster il quale viene diviso ricorsivamente finché ogni insieme è composto da un singolo elemento. La gerarchia prodotta dall'algoritmo viene chiamata dendrogramma, la quale è successivamente tagliata al livello desiderato per ottenere la ripartizione finale.

- **DBSCAN** [10] (Density-Based Spatial Clustering of Applications with Noise): raggruppa un set di punti in cluster in base alla loro densità in un determinato spazio, definendoli come regioni dense separate da aree più sparse. L'algoritmo seleziona un punto centrale ed espande il cluster aggiungendovi punti che si trovano entro una distanza specificata e che hanno un numero adeguato di vicini. I punti che non appartengono a nessun cluster vengono considerati outliers o rumore.

- **K-means** [17]: ha come obiettivo quello di partizionare n osservazioni in k cluster dove ogni osservazione appartiene al gruppo del punto medio (o **centroide**) più vicino. L'algoritmo, supponendo che i dati possano essere rappresentati come vettori, minimizza la varianza intra-cluster che equivale alla somma del quadrato della distanza Euclidea tra il centroide di un insieme e i membri di quest'ultimo:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}\in S_i} \|\mathbf{x} - \mu_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i \tag{3.1}$$

Data la natura categorica dei dati, il **clustering gerarchico agglomerativo** è la scelta ideale perché permette di utilizzare qualsiasi funzione di distanza [21]. Inoltre, questa tipologia di clustering aggrega gli elementi di un dataset solo in base alla loro somiglianza, mentre altri algoritmi considerano: la distanza da un "punto medio", la distribuzione probabilistica oppure la densità nello spazio; concetti non applicabili con la funzione di distanza scelta per esprimere la similarità tra due firme [7].

Infine dobbiamo stabilire un criterio di collegamento (o **linkage criterion**), il quale determina la distanza tra due **cluster**.

Alcuni criteri di collegamento comunemente usati includono:

Nome	**Formula**
Complete-linkage	$\max_{\mathbf{a}\in\mathbf{A},\,\mathbf{b}\in\mathbf{B}} \mathbf{d}(\mathbf{a},\mathbf{b})$
Single-linkage	$\min_{\mathbf{a}\in\mathbf{A},\,\mathbf{b}\in\mathbf{B}} \mathbf{d}(\mathbf{a},\mathbf{b})$
Average-linkage	$\frac{\mathbf{1}}{\lvert\mathbf{A}\rvert \cdot \lvert\mathbf{B}\rvert} \sum_{\mathbf{a}\in\mathbf{A}} \sum_{\mathbf{b}\in\mathbf{B}} \mathbf{d}(\mathbf{a},\mathbf{b}).$

Dove A e B sono due cluster e d è la funzione di distanza tra i singoli membri del cluster.

Il criterio **average-linkage** [26], il quale considera la media delle distanze di ogni elemento tra due cluster, si allinea bene con la *distanza di Hamming* 3.3.1, in quanto la funzione cattura le differenze tra due elementi. Dunque, attraverso questo criterio di collegamento cluster con piccole differenze vengono uniti, tenendo conto della struttura degli insiemi presi in considerazione [11].

4. Implementazione

La figura 4.1 mostra le fasi da implementare per il processo di *fingerprinting passivo* proposto.

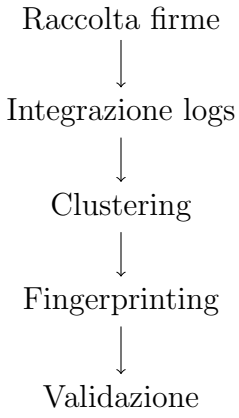

Figura 4.1: Fasi del processo di fingerprinting

A ognuna di queste fasi corrisponde uno script Python che, presi in input i risultati dello stadio precedente, produce i dati necessari al processo successivo.

- `capture_fp.py`: specificata un'interfaccia di rete (oppure un file `pcap` [28]) lo script estrae i parametri necessari alla costruzione delle firme dai pacchetti catturati. I risultati della cattura vengono scritti in file di log, i quali verranno utilizzati nella prossima fase.

- `process_logs.py`: dati in input i file di log prodotti dallo script precedente e i log delle richieste HTTPS analoghe al periodo di cattura, viene prodotto un file csv con le firme rilevate associate al sistema operativo estratto dalla stringa dello user-agent.

- `cluster_data.py`: le firme presenti nel file csv fornito vengono aggregate utilizzando l'algoritmo di clustering descritto nella sezione 3.3.2. I risulati vengono salvati in un file JSON e viene prodotto un grafico per visualizzare la composizione dei cluster.

- `fingerprint.py`: lo script legge pacchetti di rete da un'interfaccia o un file `pcap` e, utilizzando l'output dello script di clustering, tenta d'identificare il sistema operativo degli host osservati.

L'implementazione degli script sono disponibili sulla repository GitHub.

4.1 Estrazioni dei parametri TCP/IP e TLS

Ogni pacchetto ricevuto dallo script di cattura viene processato per estrarne i parametri d'interesse. L'analisi di un pacchetto avviene tramite il seguente codice:

```
def process_pkt(buf):
    eth_hdr = dpkt.ethernet.Ethernet(buf)
    if not isinstance(eth_hdr.data, dpkt.ip.IP):
        return None
    ip_hdr = eth_hdr.data
    if not isinstance(ip_hdr.data, dpkt.tcp.TCP):
        return None
    tcp_hdr = ip_hdr.data
    features = {}
    if tcp_hdr.flags == dpkt.tcp.TH_SYN:
        features = extract_tcp_features(ip_hdr, tcp_hdr)
    elif len(tcp_hdr.data) > 0 and tcp_hdr.data[0] == TLS_HANDSHAKE:
        features = extract_tls_featurs(tcp_hdr.data)
    else:
        return None
    flow = {
        "src_ip": ip_hdr.src,
        "dst_ip": ip_hdr.dst,
        "src_port": tcp_hdr.sport,
        "dst_port": tcp_hdr.dport,
        }
    flow = {**flow, **features}
    return flow
```

Il *parsing* dei pacchetti è gestito tramite l'ausilio della libreria `dpkt` [27]. Dopo essere sicuri di aver ricevuto un segmento TCP, viene stabilito se si tratta di un messaggio di **SYN** o un **TLS Handshake** per procedere all'estrazione delle features. Infine, i parametri estratti vengono aggiunti a un dizionario assieme alle informazioni del flusso del pacchetto.

L'estrazione dei parametri TCP/IP è gestita dal seguente frammento di codice:

```
def extract_tcp_features(ip_hdr, tcp_hdr):
    tcp_opts = []
    for opt in dpkt.tcp.parse_opts(tcp_hdr.opts):
        tcp_opts.append((opt[0], int.from_bytes(opt[1], "big")))
    features = {
```

```
            "ttl": ip_hdr.ttl,
            "tcp_window": tcp_hdr.win,
            "tcp_opts": tcp_opts
        }
        return features

```

Le opzioni TCP vengono salvate in una lista come coppia tipo-valore per poi essere aggiunte a un dizionario con il valore del TTL e della finestra TCP.

I messaggi TLS di tipo Handshake vengono invece analizzati nel seguente modo:

```
    def extract_tls_featurs(buf):
        record = dpkt.ssl.TLSRecord(buf)
        if record.data[0] != TLS_CLIENT_HELLO :
            return None
        handshake = dpkt.ssl.TLSHandshake(record.data)
        client_hello: dpkt.ssl.TLSClientHello = handshake.data
        supp_groups = []
        if not hasattr(client_hello, "extensions"):
            return None
        for ext_type, ext_data in client_hello.extensions:
            if ext_type == SUPPORTED_GROUPS_TYPE:
                supp_groups_len = struct.unpack("!H", ext_data[:2])[0]
                ptr = 2
                while ptr <= supp_groups_len:
                    supp_group = struct.unpack("!H", ext_data[ptr:ptr + 2])[0]
                    ptr += 2
                    supp_groups.append(supp_group)
        ciphersuites = [x.code for x in client_hello.ciphersuites]
        features = {
            "ciphersuites": ciphersuites,
            "supp_groups": supp_groups
        }
        return features
```

Nel caso in cui il messaggio di Handshake sia un **ClientHello** e contenga delle estensioni TLS, verranno estratte le liste dei cifrari e i *supported_groups* (entrambi rappresentati come numeri interi). I parametri estratti verrano infine scritti in un file di log seguendo il formato dell'esempio 3.2.1.

4.2 Integrazione logs

Come spiegato in 3.2.1, il passo successivo è quello d'integrare i file di log prodotti dallo strumento di cattura con i log delle richieste HTTPS del web server di ntop.org.

Per ciascun indirizzo IP osservato i parametri TCP/IP e TLS vengono associati allo user-agent della richiesta corrispondente. Non avendo a disposizione la porta TCP nei log delle richieste HTTPS, usata per stabilire l'unicità di un flusso come esposto in 3.1.3, l'associazione dei log avviene in ordine cronologico.

```
host2features: "dict[str, dict[str, str]]" = {}
for filename in os.listdir(user_agent_log_dir):
    if filename.endswith(".log"):
        parse_user_agent_file(os.path.join(
            user_agent_log_dir, filename), host2features)
for filename in os.listdir(features_log_dir):
    if filename.endswith(".log"):
        parse_features_file(host2features, os.path.join(
            features_log_dir, filename))
return host2features
```

Quindi richieste inviate da client con un indirizzo IP già osservato verranno scartate.

I log delle richieste HTTPS seguono il formato:

```
Indirizzo IP - - [Timestamp] "HTTPS Request" Status Code Content-Length "Referer" "User-Agent"
```

Da ogni voce nei file di log vengono estratti l'indirizzo IP e lo user-agent, per poi essere aggiunti al dizionario principale in cui verranno registrati tutti i client osservati durante il periodo di cattura.

```
with open(log_file, mode="r") as fp:
    for line in fp:
        ip_string, log_entry = line.split(' ', maxsplit=1)
        if not is_ipv4_address(ip_string):
            continue
        if ip_string in hosts:
            continue
        hosts[ip_string] = {
                feature: None for feature in FEATURES_SET}
        log_entry = log_entry.split('\" \"', maxsplit=1)
```

```
            user_agent = parse(log_entry[-1])
            os_name = f'{user_agent.os.family}'.strip()
            if os_name != "Other":
                if os_name in UNIX_UAS:
                    os_name = 'Linux'
                hosts[ip_string]['os_name'] = os_name
```

Per semplificare il processo di clustering, sistemi operativi Unix-like sono classificati sotto Linux.

```
UNIX_UAS = {'Debian', 'Ubuntu', 'CentOS', 'FreeBSD', 'Chrome OS', 'Android'}
```

Listing 4.1: Sistemi operativi Unix-like

Tuttavia i paramteri TPC/IP e TLS devono essere processati prima dell'integrazione con i log HTTPS. Secondo Lippmann et al. [16] il valore del TTL deve essere arrotondato alla potenza di due più vicina, questo per rimuovere l'influenza del percorso di rete tra client e web server. In aggiunta, dalla lista delle opzioni TCP verranno estratti i valori del MSS e del window scaling.

```
def extract_tcp_ip_features(feature: str) -> "tuple[str, str, str, str, str]":
    ttl_str, windowSize_str, tcp_opts_str = feature.split(',', maxsplit=2)
    exponent = math.ceil(math.log2(int(ttl_str)))
    ttl_int = 2 ** exponent
    if ttl_int > 255:
        ttl_int = 255
    tpl_lst = TPL_REGEX_PATTERN.findall(tcp_opts_str)
    tcp_opts_str = ''
    maximumSegmentSize_str = '0'
    windowScaling_str = '0'
    for (option, value) in tpl_lst:
        tcp_opts_str += f'{option}-'
        if option == TCP_OPT_MMS:
            maximumSegmentSize_str = value
            continue
        if option == TCP_OPT_WS:
            windowScaling_str = value
    return f'{ttl_int}', f'{windowSize_str}', f'{maximumSegmentSize_str}', f'{windowScaling_str}', f'{
 tcp_opts_str[:-1]}'
```

Dai parametri TLS devono essere semplicemente rimossi i parametri di GREASE [5] dalla lista delle *ciphersuites* e dei *supported_groups*.

```
GREASE = {
    0x0A0A, 0x1A1A, 0x2A2A, 0x3A3A, 0x4A4A, 0x5A5A, 0x6A6A, 0x7A7A,
```

```
        0x8A8A, 0x9A9A, 0xAAAA, 0xBABA, 0xCACA, 0xDADA, 0xEAEA, 0xFAFA
    }

    features_lst = []
    for feature in features_str.split('],['):
        features_lst.append('-'.join([element for element in feature.lstrip(
            '[').rstrip(']').split(', ') if int(element) not in GREASE]))
    return features_lst
```

Il risultato del processo d'integrazione dei log è serializzato in un file CSV del seguente tipo:

os_name	ttl	ws	mss	win_scale	tcp_opts	ciphersuites	supported_groups
Linux	128	65535	1412	8	2-4-8-1-3	4865-4866-4867-49195	29-23-24
Linux	64	65535	1250	6	2-1-3-1-1-8	4865-4866-4867-49196	29-23-24-25
Linux	64	29200	1460	7	2-4-8-1-3	49200-49196-49192-49188	23-25-28-27-24
Linux	64	64240	1460	8	2-4-8-1-3	4866-4867-4865-49196	29-23-30-25-24
Windows	128	64240	1460	8	2-1-3-1-1-4	4865-4866-4867-49195	29-23-24
Mac OS X	255	26883	1460	7	2-4-8-1-3	49196-49195-49199-491712	23-24-25-9-10
Linux	64	64240	1440	7	2-4-8-1-3	4865-4866-4867-49195	29-23-24

Tabella 4.1: Tabella delle features da utilizzare nel processo di clustering

Il dataset prodotto verrà utilizzato nella fase di clustering e di fingerprinting, esso dovrà essere randomizzato e separato al fine di creare due dataset: uno di training e uno di test. Questa procedura permette di ridurre il bias dovuto a un possibile ordinamento dei dati. I log sono raccolti e analizzati in ordine cronologico, lasciandone inalterata la struttura vi è il rischio d'isolare una classe di firme nella porzione dei dati di test. Questo comporterebbe la creazione di un modello che non generalizza in modo appropriato il campione di dati a disposizione, poiché le possibili categorie di firme non sono equamente distribuite.

4.3 Clustering

Prima di poter applicare l'algoritmo di clustering è necessaria una fase di **preprocessing** di riduzione e trasformazione sui nostri dati. In quanto nel nostro dataset potrebbero essere presenti elementi con valori mancanti o duplicati. In aggiunta è opportuno codificare i dati in un formato semplice da manipolare. Le librerie Python `pandas` [18] e `scikit-learn` [23] sono state impiegate a questo fine.

```
# Viene esclusa la prima colonna del dataset, che include i nomi dei sistemi operativi
data = dataframe.dropna().iloc[:, 1:].drop_duplicates()
X = data.to_numpy()
X_enc = preprocessing.OrdinalEncoder().fit_transform(X)
```

Listing 4.2: Preprocessing dei dati

La classe `OrdinalEncoder` permette di codificare features categoriche come un vettore di numeri interi. Dunque, ogni colonna viene trasformata in un array ordinale d'interi da **0** a **n_classi − 1**, dove il numero di classi equivale ai valori unici presenti in una colonna. A questo punto i dati sono pronti per essere sottoposti al processo di clustering.

```
def _cluster_data(self, X: np.ndarray):
    X_enc = preprocessing.OrdinalEncoder().fit_transform(X)

    distance_matrix = pairwise_distances(
        X_enc, metric='hamming', n_jobs=-1, w=WEIGHTS)

    model = AgglomerativeClustering(
            metric="precomputed", linkage='average', compute_full_tree=True)
    with tempfile.TemporaryDirectory() as temp_dir:
        memory = joblib.Memory(temp_dir, verbose=0)
        model.set_params(memory=memory)

        sample_range = range(2, len(X_enc))
        silhouette_avgs = np.zeros(len(sample_range))
        for n in sample_range:
            model.set_params(n_clusters=n)
            labels = model.fit_predict(distance_matrix)
            silhouette_avgs[n - 2] = silhouette_score(
                distance_matrix, labels, metric='precomputed', random_state=0)

        best_idx = np.argmax(silhouette_avgs)
        model.set_params(n_clusters=best_idx + 2)
        model = model.fit(distance_matrix)

    return model.labels_, model.n_clusters_
```

Listing 4.3: Algoritmo del clustering delle firme

Una matrice di distanze è calcolata per ridurre il costo computazionale dell'algoritmo. Le distanze vengono calcolate con la distanza di Hamming descritta nella sottosezione 3.3.1 assieme a un vettore di **pesi**. È stato osservato che una stessa macchina può inviare pacchetti **TCP-SYN** con valori del: TTL, Window Size, MSS e Window Scaling differenti,

mentre la lista delle opzioni TCP, le ciphersuites e i supported_groups tendono a rimanere invariati. Perciò è stata necessaria una fase di **tuning**, nella quale sono state analizzate le firme all'interno dei cluster ottenuti, modificando i valori dei pesi di ogni feature in modo da ottenere insiemi che rappresentassero al meglio le varie tipologie di firma, mantenendo un compromesso tra numero di cluster e *silhouette score*.

Alla fine di questo processo sono stati determinati i seguenti pesi per le features considerate:

```
# ttl: 0, ws: 1, mss: 2, win scale: 3, tcp opts: 4, ciphersuites: 5, supported groups: 6
WEIGHTS = np.array([1, 1, 1, 1, 5, 5, 5])
```

Listing 4.4: Vettore dei pesi

Per l'algoritmo di clustering gerarchico agglomerativo è stata scelta l'implementazione di `scikit-learn` [23] `AgglomerativeClustering`. Non conoscendo a priori il numero ottimale di cluster, ovvero il valore che permette un'aggregazione naturale dei dati, l'impiego di metriche per la valutazione del clustering è imprescindibile. Secondo Rousseeuw [25] l'impiego del **silhouette score** permette di stabilire il numero appropriato di cluster da cercare. Perciò vengono valutati $\mathbf{n-2}$ possibili risultati di clustering dove $\boldsymbol{n}$ corrisponde al numero delle firme da aggregare. Per ogni esito ne viene calcolato il silhoute score e infine viene scelto il numero di cluster che ottiene il punteggio maggiore.

```
cluster2data = {k: [] for k in range(n_clusters)}
data2cluster = {}
for cluster_id, elem in zip(cluster_labels, data_arr):
    cluster2data[cluster_id.item()].append(elem)
    data2cluster[elem] = cluster_id.item()

cluster2occurance = {k: {label: 0 for label in dataframe[labels_column].unique()}
                        for k in range(n_clusters)}

rows = dataframe.apply(lambda row: ','.join(row[1:].astype(str)), axis=1)
cluster_ids = np.array([data2cluster[row]for row in rows])

occurance_counts = np.zeros(n_clusters)
for cluster_id, label in zip(cluster_ids, dataframe[labels_column]):
    cluster2occurance[cluster_id][label] += 1
    occurance_counts[cluster_id] += 1
```

Listing 4.5: Algoritmo di costruzione dizionari risulati del clustering

Il frammento di codice sovrastante si occupa di costruire due dizionari: `cluster2data` associa le firme al proprio cluster di appartenenza, invece `cluster2occurance` è utilizzato per contare le occorrenze di un sistema operativo all'interno di ogni cluster.

```
K_PERCENTILE = 85
threshold = np.percentile(occurance_counts, K_PERCENTILE)
print(f"Threshold at {K_PERCENTILE}th perctile: {threshold:.1f}")

indices = np.where(occurance_counts < threshold)[0]
cluster2data = {k: v for k, v in cluster2data.items()
                if k not in indices}
cluster2occurance = {k: {label: count for label, count in cluster2occurance[k].items(
) if count > 0} for k in cluster2occurance if k not in indices}
n_clusters -= len(indices)
```

Dopodiché è necessario scartare i cluster con un basso di numero di occorrenze totali; essi potrebbero contenere firme viste raramente o possibili **outliers**, quindi non risultano utili al processo di fingerprinting. Al seguito di varie sperimentazioni, è stato osservato che eliminare i cluster con una quantità di osservazioni complessive al di sotto del valore del 85° percentile permette di conservare abbastanza informazioni per comprendere la struttura dei dati.

```
for k, occurances in cluster2occurance.copy().items():
    labels = list(occurances.keys())
    values = np.array(list(occurances.values()), dtype=np.float_)
    predominat_class_idx = np.argmax(values)
    if values[predominat_class_idx] >= 55:
        cluster2occurance[k] = labels[predominat_class_idx]
        continue
    del cluster2occurance[k]
    del cluster2data[k]
    n_clusters -= 1

clusters = {cluster_id: {'Fingerprints': cluster2data[cluster_id],
    'Os': cluster2occurance[cluster_id]} for cluster_id in cluster2data}
return clusters
```

Per stabilire l'identità di un cluster viene scelto il sistema operativo con il maggior numero di occorrenze, assumendo che la soglia di percentuale minima per considerare una classe di OS predominante si attesti sopra al 55% delle occorrenze totali. I cluster privi di un sistema operativo con una maggioranza netta non vengono presi in considerazione, in quanto non forniscono informazioni rilevanti sulle firme contenutevi.

Da ultimo, i dizionari `cluster2occurance` e `cluster2data` vengono uniti e serializzati in un file JSON che segue lo schema:

```
"cluster_id": {
    "type": "object",
    "properties": {
        "Fingerprints": {
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "Os_percetange": {
            "type": "object",
            "properties": {
                "Os_name": {
                    "type": "string"
                }
            }
        }
    },
    "required": [
        "Fingerprints",
        "Os_percetange"
    ]
}
```

```
"68": {
    "Fingerprints": [
        "64,29200,1460,7,2-4-8-1-3,49199-49200-49195-49196-52392-52393-49171-49161-49172,29-23-24-25",
        "64,14600,1460,7,2-4-8-1-3,49199-49200-49195-49196-52392-52393-49171-49161-49172,29-23-24-25",
        "64,64240,1460,7,2-4-8-1-3,49199-49200-49195-49196-52392-52393-49171-49161-49172,29-23-24-25",
        "64,29200,1424,7,2-4-8-1-3,49199-49200-49195-49196-52392-52393-49171-49161-49172,29-23-24-25",
        "64,28400,1420,7,2-4-8-1-3,49199-49200-49195-49196-52392-52393-49171-49161-49172,29-23-24-25",
        "64,14600,1460,9,2-4-8-1-3,49199-49200-49195-49196-52392-52393-49171-49161-49172,29-23-24-25",
    ],
    "Os_percetange": {
        "Linux": "100.0%"
    }
}
```

Listing 4.6: Esempio di un possibile output

4.4 Fingerprinting

Lo script di fingerprinting deserializza il file JSON ottenuto da `cluster_data.py` e lo salva in un dizionario. Successivamente viene costruito un secondo dizionario in cui, per ogni cluster, vengono salvati i valori unici di ogni feature. Quest'ultimo verrà utilizzato nel caso in cui non venga trovato un match esatto nel primo dizionario per ottenere un riscontro parziale.

```
{
    'ttl': {'128', '64', '255'},
    'tcp_window': {'64800', '64240', '8192'},
    'mss': {'1386', '1440', '1380', '1452', '1412', '1460', '1220', '1448'},
    'win_scale': {'8'}, 'tcp_opts': {'2-1-3-1-1-4'},
    'ciphersuites': {'
 4865-4867-4866-49195-49199-52393-52392-49196-49200-49162-49161-49171-49172-156-157-47-53'},
    'supp_groups': {'29-23-24-25-256-257'}
}
```

Listing 4.7: Esempio di voce del dizionario per riscontri parziali

La classificazione delle firme avviene tramite una ricerca nel database. Nel caso in cui non venga trovato un match esatto verrà allora calcolato un punteggio di similarità per cercare una corrispondenza parziale. Dunque, per ogni feature della firma priva di corrispondenza ne viene controllata la presenza all'interno dei cluster. In caso di riscontro il punteggio viene incrementato, tenendo conto dei pesi utilizzati per il calcolo della distanza di Hamming 4.4. Da ultimo, alla signature verrà assegnato il sistema operativo del cluster che ottiene lo score di similitudine più alto.

```
def _search_fingerprint_in_db(self, fp_str):
    fingerprint = "No match"
    for os_name, signatures in self.database.items():
        if fp_str in signatures:
            fingerprint = os_name
            break

    if fingerprint == "No match":
        max_score = 0
        most_similar_cluster = -1

        for cluster_id, cluster_data in self.mod_database.items():
            curr_score = 0

```

```
                for feature, value in zip(FEATURES_SET, fp_str.split(",")):
                    if value in cluster_data[feature]:
                        if feature in MOST_WEIGHT:
                            curr_score += 5
                            continue
                        curr_score += 1

                if curr_score > max_score:
                    max_score = curr_score
                    most_similar_cluster = cluster_id

            fingerprint = self.mod_database[most_similar_cluster]["Os"]

        return fingerprint
```

Listing 4.8: Ricerca di una firma all'interno del database

Per visualizzare le performance del nostro processo di fingerprinting viene impiegata una matrice di confusione.

		Classi Predette		
		Classe 1	**Classe 2**	**Class 3**
Classi Reali	**Classe 1**	A_{11}	A_{12}	A_{13}
	Classe 2	A_{21}	A_{22}	A_{23}
	Classe 3	A_{31}	A_{32}	A_{33}

Tabella 4.2: Matrice di confusione per classificazioni multi classe.

Le colonne della matrice rappresentano i valori predetti, mentre le righe i valori reali. Gli elementi della riga *i-esima* e colonna *j-esima* esprimono i casi in cui il tool di fingerprinting identifica la classe i come classe j. Pertanto, i valori sulla diagonale della matrice indicano il numero di firme correttamente classificate, mentre per ogni riga l'elemento *j-esimo* corrisponde ai falsi positivi. Invece, per ogni colonna, l'elemento *i-esimo* rappresenta il numero di falsi negativi.

Tramite questa matrice è possibile calcolare: la precisione (o **precision**), il recupero (o **recall**) e la media armonica (F_1 **score**) del processo di fingerprinting.

La precisione di una classe è definita come il rapporto tra il numero di elementi correttamente identificati alla classe (veri positivi) e il numero totale di elementi classificati come appartenenti alla classe (somma dei veri positivi e falsi positivi).

$$\text{Precisione} = \frac{veripositivi}{veripositivi + falsipositivi}$$

Il recupero di una classe stabilisce il rapporto tra il numero di elementi correttamente identificati alla classe (veri positivi) e il numero totale di elementi effettivamente appartenenti alla classe (somma dei veri positivi e falsi negativi).

$$\text{Recupero} = \frac{veropositivi}{veropositivi + falsinegativi}$$

Infine, la media armonica è espressa come:

$$F_1 = 2 \cdot \frac{\text{precisione} \cdot \text{recupero}}{\text{precisione} + \text{recupero}} = \frac{2 \cdot \text{veripositivi}}{2 \cdot \text{veripositivi} + \text{falsipositivi} + \text{falsinegativi}}$$

5. Validazione

In questo capitolo sono presentati i risultati del processo di clustering e di fingerprinting. Come accennato nella sottosezione 3.2.2 il dataset copre un periodo di sedici ore, in cui sono stati raccolti: segmenti **TCP-SYN**, **TLS ClientHello** e richieste HTTPS. Dopo il processo di pulizia e integrazione dei log sono state identificate più di 2400 firme, di cui 600 uniche. Di queste l'80% sono state utilizzate nel processo di clustering, mentre le restanti come dati di test per la valutazione delle performance del processo di fingerprinting.

5.1 Risultati clustering

Avviando lo script `cluster_data.py` e fornendo in input il dataset di test viene determinato che il numero ottimale di cluster si attesta a 200 con un silhouette score medio di 0.5079, che indica come sia stata trovata una buona ripartizione dei dati.

Il numero di cluster ottenuto è di gran lunga superiore a quello dei sistemi operativi presenti nel dataset. Un risultato del genere può essere attribuito al fatto che il clustering è eseguito sul kernel e lo user space dei client. Come spiegato in 3.1.2, macchine con il medesimo sistema operativo posso utilizzare librerie o implementazioni del protocollo TLS differenti, questo comporta che il **ClientHello** di due client con lo stesso OS non coincida. Di conseguenza le firme generate presenteranno caratteristiche TCP/IP identiche, ma le differenze nelle features TLS porterà alla creazione di cluster diversi.

Calcolando l' 85-esimo percentile risulta che il 15% delle aggregazioni conta con più di 11 occorrenze. Vengono quindi rimossi gli insiemi con un numero di osservazioni minore o uguale al valore di soglia calcolato, portando il numero di cluster a 30; la figura 5.1 ne mostra la composizione. Di questi, quattro sono scartati poiché privi di una famiglia di sistemi operativi predominante.

Dunque, sono individuati: 12 cluster per Linux, 10 per Windows, due per iOS e Mac OS X. Il numero esiguo di gruppi identificati per i sistemi operativi Apple deriva dal fatto che la somma delle loro firme compone meno del 30% del dataset.

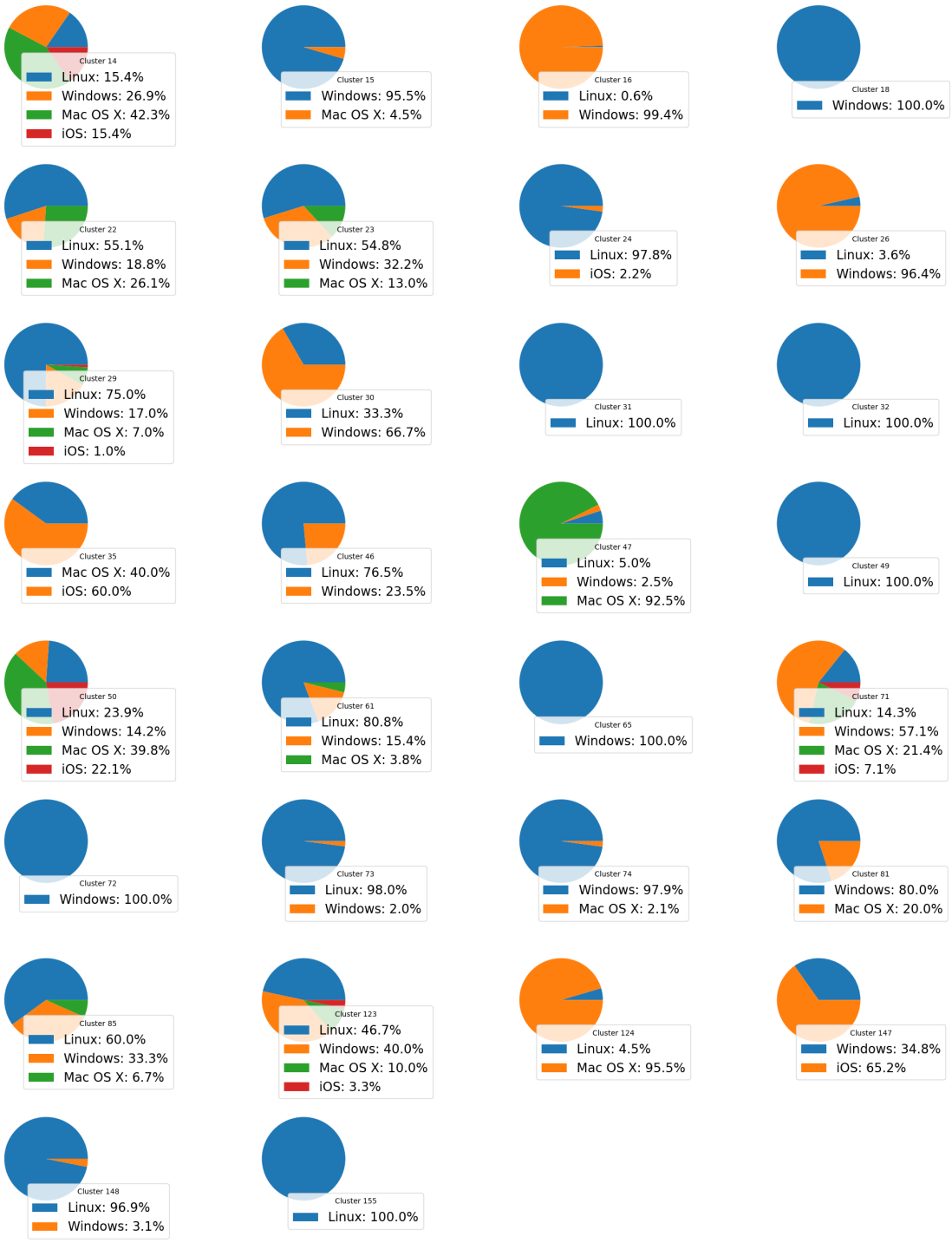

Figura 5.1: Distribuzione dei sistemi operativi all'interno di ogni cluster

5.2 Risultati fingerprinting

Tramite i risultati del processo di clustering sono stati costruiti due dizionari impiegati nella classificazione delle firme di test. Come spiegato in 4.4, il primo dizionari permette

l'identificazione istantanea di firme già osservate. Il secondo è utilizzato per il matching approssimato che restituisce il sistema operativo del cluster più simile alla signature analizzata.

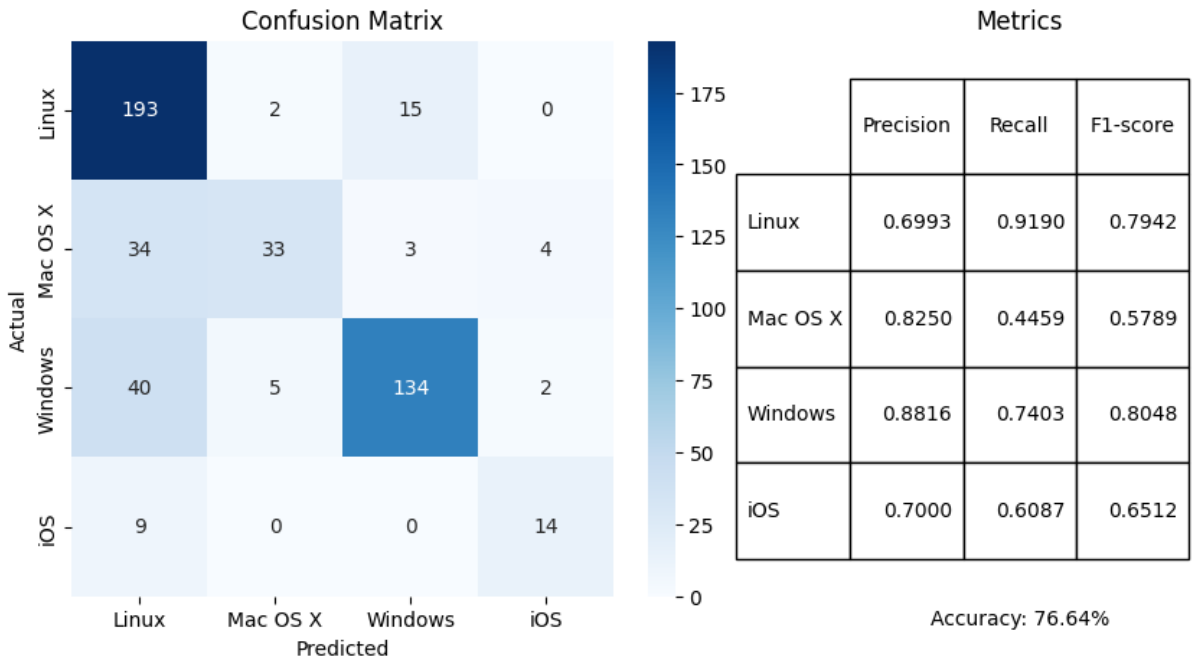

Figura 5.2: Matrice di confusione sui dati di test

La figura 5.2 mostra la matrice di confusione e le metriche: precision, recall, F_1 score e accuratezza che permettono di visualizzare le performance del processo di classificazione.

Linux e Windows riescono a ottenere dei buoni risultati, mentre l'identificazione di Mac OS X e iOS riscontra alcune difficoltà. La natura sbilanciata del dataset, ovvero l'irregolare distribuzione delle osservazioni (e.g. le firme provenienti da sistemi Linux sono di gran lunga superiori a quelle di Mac OS X), e il basso numero di dati raccolti per i sistemi operativi Apple 3.4 hanno compromesso la capacità del modello di aggregare alcune classi di firme. Questo ha poi causato performance ridotte nella classificazione degli OS.

I risultati degli studi sui modelli a sessione singola, sviluppati da Anderson et al. [3], che utilizzando features TCP/IP e TLS ottengono rispettivamente un'accuratezza del 55,54% e del 76,39%. Il primo impiega le medesima caratteristiche esposte in 3.1.1, il secondo invece impiega la lista delle estensioni TLS e i dati associati alle estensioni *ec_point_format* e *application_layer_protocol_negotiation*, oltre ai parametri introdotti in 3.1.2.

I classificatori TCP/IP e TLS proposti da Laštovička et al. [15] raggiungono risultati ancora migliori con un'accuratezza nell'identificazione dei sistemi operativi al di sopra del 90% e 80%. Il modello basato sullo stack TCP/IP utilizzano: la dimensione del pacchetto TCP-SYN, la dimensione della finestra TCP e il valore del TTL. Quello TLS impiega i parametri proposti in questa tesi integrando anche: la versione TLS utilizzata dal client, i dati associati alle estensioni *server_name_indication* e *ec_point_format*, il vettore delle estensioni TLS e la sua dimensione.

Infine il modello basato su SVM di Muehlstein et al. [20] ottiene un'accuratezza oltre il 96% utilizzando più di 40 features tra: parametri TCP/IP, TLS e metriche sul comportamento di un flusso.

Oltre al maggior numero di caratteristiche utilizzate, questi metodi d'identificazione, sono stati allenati su un numero esponenzialmente maggiore di dati, che hanno permesso di ottenere questi ottimi risultati.

6. Conclusioni

In questa tesi è stato proposto un modello di clustering gerarchico agglomerativo per aggregare firme di sistemi operativi. La successiva analisi delle similitudini tra signatures generate dalla medesima famiglia di OS ha permesso di definire un metodo di *fingerprinting passivo*, per identificare il sistema operativo di macchine remote avendo a disposizione solo sessioni di traffico criptato. Questo grazie alle informazioni veicolate dai protocolli TCP/IP e TLS, facilmente estraibili dai pacchetti di rete di un host.

I risultati precedentemente discussi 5 mostrano come la strategia proposta nel capitolo 3 ottenga una discreta performance raggiungendo un'accuratezza del 76,64%, migliorabile aumentando la dimensione e la diversificazione dei dati di training.

A differenza di metodi basati sull'apprendimento supervisionato [2, 3, 15, 20], viene usato un minor numero di features che ne riduce la complessità, facilitandone l'uso in ambienti in cui la raccolta esaustiva di diversi parametri non è realizzabile.

6.1 Lavori futuri

Con ulteriore tempo, sarebbe stato auspicabile:

- Raccogliere e organizzare in modo più efficiente i dati di prova. Durante la lettura e ricerca di strategie di fingerprinting, la mancanza di un dataset adeguato per testare nuovi metodi d'identificazione ha generato difficoltà non indifferenti. Per questo motivo, la creazione di una raccolta open source contenente sessioni di traffico di rete reali sarebbe di grande vantaggio per progetti futuri nell'ambito dell'OS fingerprinting.

- Usare tecniche di riduzione della dimensionalità, come t-SNE [29], per visualizzare la dispersione nello spazio delle firme.Comprenderne al meglio la struttura permetterebbe di selezionare parametri più appropriati e funzioni di distanza che meglio descrivono la similarità degli elementi.

- Osservare come il modello si comporti in altri contesti di rete. Per esempio, in ambienti in cui la distribuzione dei sistemi operativi risulta più uniforme la classificazione dovrebbe ottenere performance migliori.

Bibliografia

[1] Internet Protocol. RFC 791, September 1981.

[2] Ahmet Aksoy, Sushil Louis, and Mehmet Hadi Gunes. Operating system fingerprinting via automated network traffic analysis. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 2502–2509, 2017.

[3] Blake Anderson and David McGrew. Os fingerprinting: New techniques and a study of information gain and obfuscation. In *2017 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2017.

[4] Blake Anderson and David McGrew. Accurate tls fingerprinting using destination context and knowledge bases, 2020.

[5] David Benjamin. Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility. RFC 8701, January 2020.

[6] Robert Beverly. A robust classifier for passive tcp/ip fingerprinting. In Chadi Barakat and Ian Pratt, editors, *Passive and Active Network Measurement*, pages 158–167, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[7] Shyam Boriah, Varun Chandola, and Vipin Kumar. Similarity measures for categorical data: A comparative evaluation. In *Proceedings of the 2008 SIAM international conference on data mining*, pages 243–254. SIAM, 2008.

[8] McGrew David, Enright Brandon, and Anderson Blake. Mercury: Fast tls, tcp, and ip fingerprinting. https://github.com/cisco/mercury, 2020.

[9] Wesley Eddy. Transmission Control Protocol (TCP). RFC 9293, August 2022.

[10] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[11] BS Everitt. Cluster analysis: a brief discussion of some of the problems. *The British Journal of Psychiatry*, 120(555):143–145, 1972.

[12] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Semantics. RFC 9110, June 2022.

[13] John A. Hawkinson and Tony J. Bates. Guidelines for creation, selection, and registration of an Autonomous System (AS). RFC 1930, March 1996.

[14] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064, 2014.

[15] Martin Laštovička, Stanislav Špaček, Petr Velan, and Pavel Čeleda. Using tls fingerprints for os identification in encrypted traffic. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6, 2020.

[16] Richard Lippmann, David Fried, Keith Piwowarski, and William Streilein. Passive operating system identification from tcp/ip packet headers. In *Workshop on Data Mining for Computer Security*, volume 40, 2003.

[17] J MacQueen. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, pages 281–297. University of California Los Angeles LA USA, 1967.

[18] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[19] Zalewski Michal. p0f v3: passive fingerprinter. https://lcamtuf.coredump.cx/p0f3/, 2012.

[20] Jonathan Muehlstein, Yehonatan Zion, Maor Bahumi, Itay Kirshenboim, Ran Dubin, Amit Dvir, and Ofir Pele. Analyzing https encrypted traffic to identify user's operating system, browser and application. In *2017 14th IEEE Annual Consumer Communications I& Networking Conference (CCNC)*, pages 1–6, 2017.

[21] Frank Nielsen. *Hierarchical Clustering*, pages 195–211. Springer, 02 2016.

[22] Yoav Nir, Simon Josefsson, and Manuel Pégourié-Gonnard. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier. RFC 8422, August 2018.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[25] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[26] Robert R Sokal and Charles D Michener. A statistical method for evaluating systematic relationships. *Multivariate statistical methods, among-groups covariation*, page 269, 1975.

[27] Dug Song. dpkt: Fast, simple packet creation/parsing, with definitions for the basic TCP/IP protocols. https://github.com/kbandla/dpkt, 2007–2021.

[28] The Tcpdump Group. libpcap - a portable framework for packet capture. Online, 2011.

[29] Laurens van der Maaten and Geoffrey E. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[30] B. Waggener and W.N. Waggener. *Pulse Code Modulation Techniques*. A Solomon press book. Springer US, 1995.

[31] Joe Ward Jr. Hierarchical grouping to maximize payoff. Technical report, Air Force Wright Air Development Division, Personnel Laboratory LACKLAND AFB, 1961.

[32] Wikipedia contributors. Cluster analysis — Wikipedia, the free encyclopedia, 2023. [Online; accessed 20-June-2023].