

UNIVERSITÀ DEGLI STUDI DI BRESCIA

---

Relazione progetto Sicurezza Informatica

## **Adversarial machine learning and malware**



**Studenti**

Lorenzo Papa MAT 715472  
Yanez D. Parolin MAT 715432  
Lorenzo Serina MAT 715995

---

A.A. 2020/2021

# Indice

<b>1</b>	<b>Introduzione</b>	1
<b>2</b>	<b>Elementi di partenza: dataset, modello e librerie</b>	2
2.1	Formato PE . . . . .	2
2.2	Struttura EMBER . . . . .	3
2.3	LightGBM . . . . .	5
2.4	SHAP Library . . . . .	5
2.4.1	Spiegazione di una singola previsione . . . . .	6
2.4.2	Spiegazione di una singola feature . . . . .	6
2.4.3	Spiegazione dell'intero dataset . . . . .	7
<b>3</b>	<b>Come abbiamo realizzato l'attacco</b>	9
3.1	Ricerca delle vulnerabilità . . . . .	9
3.2	Fase di attacco al modello: modifica del malware . . . . .	13
3.3	Test ulteriori . . . . .	18
<b>4</b>	<b>Possibili contromisure</b>	20
4.1	Contromisure per la tecnica di <i>overlay</i> . . . . .	20
4.2	Contromisure per la tecnica di <i>packing</i> . . . . .	21
4.3	Contromisure per le vulnerabilità del dataset . . . . .	21
<b>5</b>	<b>Conclusioni</b>	22
	<b>Riferimenti bibliografici</b>	22

# Capitolo 1

## Introduzione

Il progetto realizzato consiste nella ricerca e modifica del codice eseguibile di un malware, con la finalità di bypassare il modello di classificazione LigthGBM allenato sul dataset di Ember ma mantenendo intatta la funzionalità del codice stesso.

Nel capitolo 2 vengono presentati gli elementi da cui siamo partiti, ovvero il dataset EMBER, il modello LigthGBM e la libreria SHAP per identificare le feature più importanti e le vulnerabilità del modello selezionato.

Successivamente nel capitolo 3 viene descritto il lavoro svolto per realizzare l'identificazione delle vulnerabilità del modello e la conseguente modifica del malware.

Infine, nel capitolo 4 vengono presentate possibili contromisure volte a rafforzare il modello di classificazione rispetto alle vulnerabilità da noi identificate.

# Capitolo 2

## Elementi di partenza: dataset, modello e librerie

Il dataset EMBER [1] è una collezione di feature ricavate da file PE, *Portable Executable*. Abbiamo utilizzato la versione del dataset creata nel 2018, contenente feature provenienti da 1 milione di PE analizzati in quell'anno o precedentemente. Di questi file, 800000 sono stati utilizzati per il training, mentre 200000 sono stati utilizzati per il testing. Il training set è composto da 300000 PE maligni e altrettanti innocui, a cui vengono aggiunti 200000 PE non classificati. Il test set è diviso in egual modo tra PE benigni e maligni.

### 2.1 Formato PE

Il PE è il principale tipo di eseguibile Windows. Il formato del file è composto da un numero standard di header seguiti da una o più sezioni. Gli header includono:

1. Il *Common Object File Format* (COFF) che contiene informazioni riguardanti la macchina su cui l'eseguibile è compatibile, la natura del file, il numero di sezioni e il numero di simboli;
2. Gli header opzionali, che identificano linker version, dimensione del codice, dimensione di dati inizializzati e non, l'indirizzo dell'entry point e altro;
3. Le Data Directories che includono tabelle per export, import, risorse, eccezioni, informazioni per debug e certificato e tabelle di riallocazione;
4. la tabella di sezione indica il nome, l'offset e la dimensione di ogni sezione.

Le sezioni contengono codice e dati inizializzati e in aggiunta un header con tutte le informazioni riguardanti.

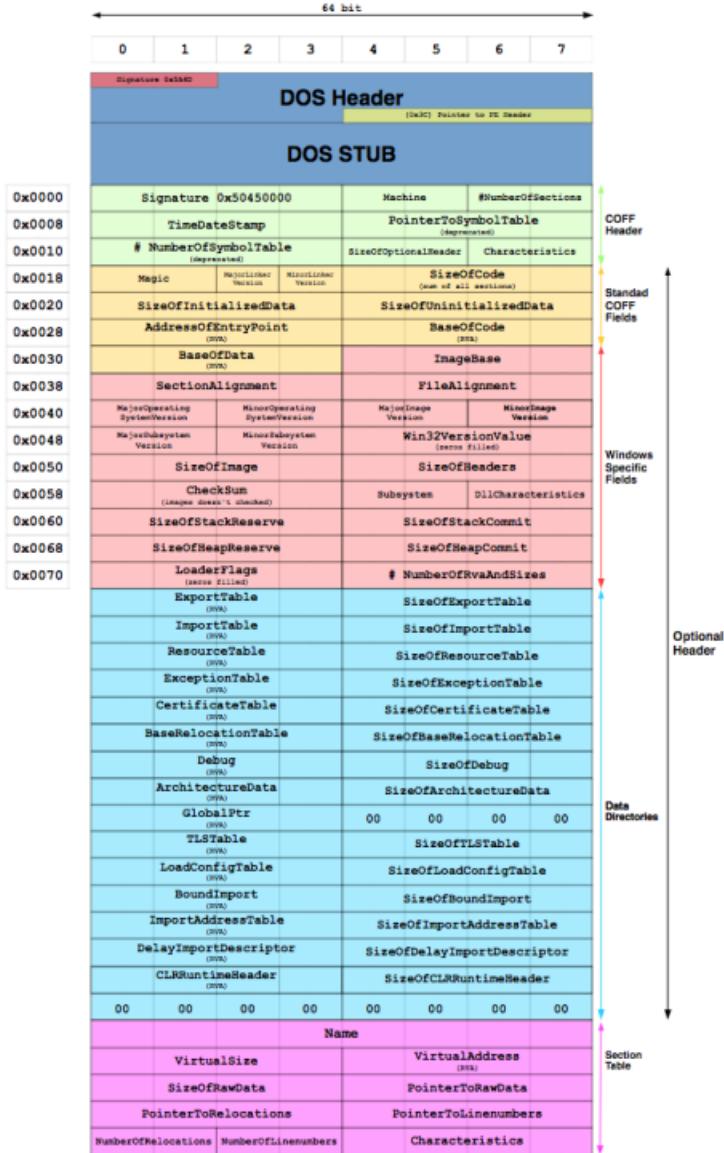


Figura 2.1: Schema formato file PE

## 2.2 Struttura EMBER

Il dataset EMBER [1] consiste in una collezione di file di linee JSON, ognuno contenente un oggetto JSON . Ogni oggetto contiene:

1. L'hash del file originale come identificatore univoco tramite SHA256 ('sha256');
2. informazioni riguardanti a quando il file è stato visto per la prima volta ('appeared');
3. un'etichetta per indicare se è un file benigno o maligno ('label');

4. otto gruppi di features non trattate per essere leggibili, che includono sia valori parsati che sotto forma di histogrammi indipendenti dal formato ('histogram', 'byteentropy', 'strings', 'general', 'header', 'section', 'imports', 'exports').

```

"sha256": "000185977be72c8b007ac347b73ceb1ba3e5e4dae4fe98d4f2ea92250f7f580e",
"appeared": "2017-01",
"label": -1,
"general": {
    "file_size": 33334,
    "vsize": 45056,
    "has_debug": 0,
    "exports": 0,
    "imports": 41,
    "has_relocations": 1,
    "has_resources": 0,
    "has_signature": 0,
    "has_tls": 0,
    "symbols": 0
},
"header": {
    "coff": {
        "timestamp": 1365446976,
        "machine": "I386",
        "characteristics": [ "LARGE_ADDRESS_AWARE", ..., "EXECUTABLE_IMAGE" ]
    },
    "optional": {
        "Subsystem": "WINDOWS.CUI",
        "DLL_characteristics": [ "DYNAMIC_BASE", ..., "TERMINAL_SERVER_AWARE" ],
        "magic": "PE32",
        "major_image_version": 1,
        "minor_image_version": 2,
        "major_linker_version": 11,
        "minor_linker_version": 0,
        "major_operating_system_version": 6,
        "minor_operating_system_version": 0,
        "major_subsystem_version": 6,
        "minor_subsystem_version": 0,
        "sizeof_code": 3584,
        "sizeof_headers": 1024,
        "sizeof_heap_commit": 4096
    }
},
"imports": {
    "KERNEL32.dll": [ "GetTickCount" ],
    ...
},
"exports": [],
"section": {
    "entry": ".text",
    "sections": [
        {
            "name": ".text",
            "size": 3584,
            "entropy": 6.368472139761825,
            "vsize": 3270,
            "props": [ "CNT_CODE", "MEM_EXECUTE", "MEM_READ" ]
        },
        ...
    ]
},
"histogram": [ 3818, 155, ..., 377 ],
"byteentropy": [ 0, 0, ..., 2943 ],
"strings": {
    "numstrings": 170,
    "avlength": 8.170588235294117,
    "printabledist": [ 15, ..., 6 ],
    "printables": 1389,
    "entropy": 6.259255409240723,
    "paths": 0,
    "urls": 0,
    "registry": 0,
    "MZ": 1
}
}

```

Figura 2.2: Schema formato file PE

Le feature sono divise in due tipi:

1. Feature parsate: comprendono 5 gruppi degli 8 gruppi di feature, sono estratte dopo aver fatto il parsing del PE. Esistono vari tipi di feature parsate:
  - (a) General file information: include la dimensione del file e altre informazioni ottenute dagli header;
  - (b) Header information: comprendono i campi presenti nel COFF, come il timestamp, la macchina obiettivo e le caratteristiche. Provvedono anche il subsystem obiettivo, le caratteristiche DLL, i file magic, il major e minor file version, linker version, system version e subsystem version e le dimensioni di codice header e commit;
  - (c) Imported functions: le funzioni importate sono inserite in una tabella;
  - (d) Exported Functions: lista di funzioni esportate;
  - (e) Section information: proprietà di ogni sezione.
2. Feature indipendenti dal formato: 3 gruppi di feature che non richiedono il parsing del PE per essere estratte. Ne esistono di vari tipi:
  - (a) Byte histogram: contiene 256 valori interi, rappresentanti il numero del valore di ogni byte nel file, viene poi normalizzato;
  - (b) Byte-entropy histogram: approssima la distribuzione congiunta dell'entropia e del valore di ogni byte;
  - (c) String information: statistiche riguardanti stringhe stampabili del codice.

## 2.3 LightGBM

Per il nostro esperimento abbiamo usato il modello consigliato dalla documentazione di EMBER, il modello LightGBM, Light Gradient Boosting Machine [4]. Questo è un framework si basa su Gradient Boosting, una tecnica che permette di generare un predittore forte unendo più predittori deboli, in questo caso alberi decisionali, e algoritmi di apprendimento che si sviluppano, per questo motivo, su alberi. È pensato per essere distribuito ed efficiente.

## 2.4 SHAP Library

SHAP (SHapley Additive exPlanations) è un approccio basato sulla teoria dei giochi utilizzato per spiegare l'output di qualsiasi modello di apprendimento automatico. Esso collega l'allocazione ottimale del credito con spiegazioni locali utilizzando i classici valori di Shapley della teoria dei giochi e le relative estensioni [6].

I valori Shapely sono misure dei contributi che ogni predittore (o caratteristica) ha in un modello di apprendimento automatico [2].

SHAP può essere installato sia tramite PyPI che conda-forge tramite i rispettivi comandi:

```
pip install shap
conda install -c conda-forge shap
```

Successivamente è necessario importare SHAP, creare una spiegazione basata sul modello LigthGBM e infine calcolare i valori SHAP.

```
import shap
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)
```

### 2.4.1 Spiegazione di una singola previsione

Con SHAP è possibile generare spiegazioni per una singola previsione. Il grafico in figura 2.3 mostra le caratteristiche che contribuiscono a spingere l'output dal valore di base (output medio del modello) al valore previsto effettivo. Il colore rosso indica le caratteristiche che stanno spingendo la previsione più in alto (malware) e il colore blu indica il meccanismo opposto (sw benigno). La dimensione dei riquadri invece indica la potenza di tale spinta [2][6].

In particolare, l'esempio mostrato è stato classificato come *malware* e si può notare come i valori delle feature *MajorLinkVersion*, *MajorSubsystemVersion* e *TimeDateStamp* siano quelli che contribuiscono maggiormente a questa sua valutazione.



Figura 2.3: SHAP values for a single sample.

Tale grafico è stato ottenuto mediante il comando:

```
shap.force_plot(    explainer.expected_value[1],    shap_values[1][19,:],    feature_names = X_test.columns)
```

Il valore 19 si riferisce al diciannovesimo sample, e il valore 1 serve per indicare come *malware* un campione con valore predetto superiore alla media (se avessimo scelto 0 il ragionamento e il grafico sarebbe stato l'opposto).

### 2.4.2 Spiegazione di una singola feature

Per comprendere l'effetto che una singola caratteristica ha sull'output del modello, è possibile tracciare un valore SHAP di quest'ultima rispetto al valore della caratteristica per tutte le istanze nel dataset.

Nel grafico in Figura 2.4 ogni punto è una singola predizione (riga) del dataset. Esso mostra come varia il valore di SHAP per *MajorLinkerVersion* (asse *y*) al variare del valore della feature stessa (asse *x*). Tale valore di SHAP permette di capire quanto tale feature influisce sull'output del modello per la previsione di quel campione. In questo caso un valore di zero indica che non è presente nessuna influenza sul risultato finale, un valore positivo indica che in quel caso si verifica una spinta verso una classificazione come *sw benigno* e un valore negativo indica la presenza di una spinta verso una classificazione come *malware*.

Le dispersioni verticali a un singolo valore mostrano gli eventuali effetti di interazione con altre features nel dataset. In particolare, nel grafico mostrato il colore tendente al rosso indica una forte interazione tra *MajorLinkerVersion* e *Characteristic* (viene scelta in automatico dalla libreria SHAP), mentre il colore tendente al blu indica una bassa interazione tra le due. In questo caso la quasi totalità dei punti presenta il pattern blu quindi si deduce che è presente una ridotta interazione tra le due features [2][6].

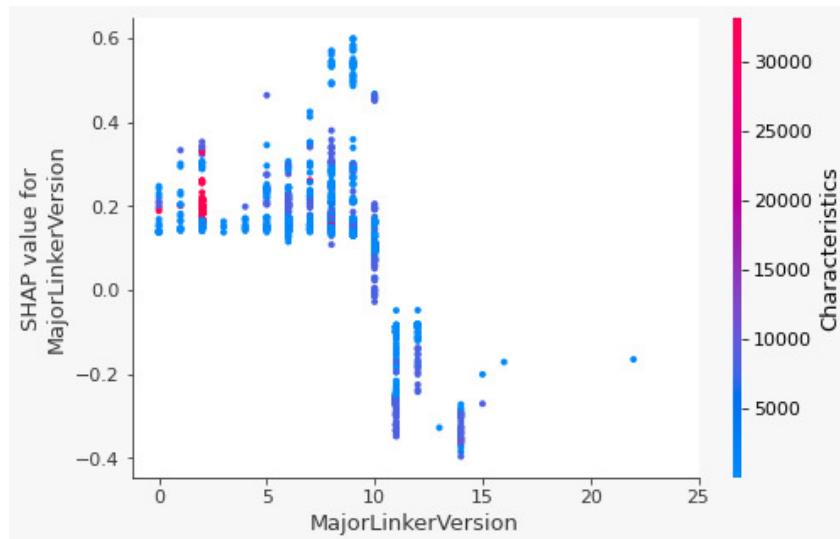


Figura 2.4: SHAP values for a single feature.

Tale grafico è stato ottenuto mediante il comando:

```
shap.dependence_plot('MajorLinkerVersion', shap_values[1], X_test, xmax = 25)
```

### 2.4.3 Spiegazione dell'intero dataset

Tramite SHAP è possibile visualizzare l'importanza delle feature e il loro impatto sulla previsione del modello.

Nel grafico in Figura 2.5 ogni punto è un valore di Shapley per una caratteristica e un'istanza. In esso le varie feature sono ordinate sulla base della somma del relativo valore di SHAP per tutti i campioni del dataset.

La posizione sull'asse  $y$  viene determinata come detto dall'importanza della caratteristica, mentre la posizione sull'asse  $x$  è determinata dal valore di Shapley. Il colore dei punti rappresenta il valore originale della caratteristica da basso (blu) ad alto (rosso) per il dato campione. I punti che si sovrappongono sono jitterati nella direzione dell'asse  $y$ , in modo da avere un senso della distribuzione dei valori di Shapley per ogni caratteristica.

Infine, esso mostra l'eventuale presenza di relazioni positive o negative tra i predittori rispetto all'output del modello [2][6].

Ad esempio, nel grafico mostrato, alti valori (rosso) della feature *Subsystem* spingono l'output verso una classificazione di tipo *sw benigno*, mentre alti valori della feature *SizeOfHeaders* compiono un effetto opposto e spingono l'output verso una classificazione di tipo *malware*.

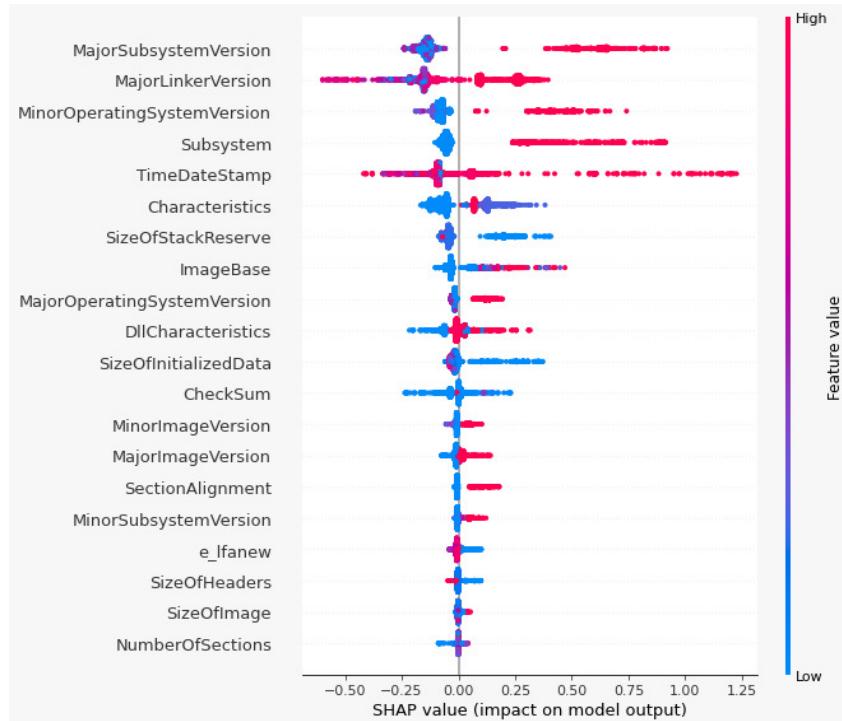


Figura 2.5: SHAP values for the entire model.

Tale grafico è stato ottenuto mediante il comando:

```
shap.summary_plot(shap_values[0], X_test)
```

Il valore zero fa in modo che vengano presentati sul lato destro i valori che spingono l'output del modello verso *sw benigno*.

# Capitolo 3

## Come abbiamo realizzato l'attacco

Il nostro obiettivo consisteva nell'identificare delle vulnerabilità all'interno di un classificatore in analisi statica. A tal fine la scelta è ricaduta sull'utilizzo del modello *LightGBM* addestrato sul dataset *EMBER* (con i parametri consigliati dalla documentazione di quest'ultimo). Ulteriore scopo era il fatto di eludere la classificazione di un *malware* e fare in modo che questo venisse classificato come *software benigno*. Secondo la documentazione di Ember, si ricorda, un malware viene classificato come tale se in seguito alla predizione si ottiene un valore maggiore di 83.3%. Dunque non è necessario scendere al di sotto di un valore pari a 50%.

### 3.1 Ricerca delle vulnerabilità

Al fine di trovare i punti deboli, che ci avrebbero permesso di modificare il codice eseguibile del malware (o il codice sorgente), abbiamo scelto di utilizzare una libreria nota come *SHAP*. Essa, come ampiamente già descritto nella sezione precedente, non consiste in un vero e proprio attacco, come sarebbe potuto essere per esempio con un *Fast Gradient Method*, bensì permette di interpretare e di spiegare come un modello lavora e come esso produce la sua classificazione. Da ciò deriva che, una volta eseguito questo "attacco interpretativo", sia necessario eseguire un'ulteriore analisi e filtraggio dei risultati ottenuti per scovare le vulnerabilità.

Il primo passo è stato il recupero di un insieme di campioni di test destinati ad essere analizzati tramite SHAP insieme al modello addestrato. Questo con il fine di calcolare l'output e l'importanza di ogni singola feature per tali esempi. Per questo motivo abbiamo prelevato dal dataset EMBER duemila campioni casuali, tra quelli all'interno del test set e su di essi abbiamo eseguito il calcolo dei cosiddetti *shap values*, ovvero valori che descrivono l'output del nostro modello per gli esempi testati.

In seguito alla produzione di tali valori abbiamo scelto di stampare il grafico riassuntivo (Figura 3.1) che descrive l'importanza di ogni feature all'interno di questo sottoinsieme di duemila campioni. Dove sull'asse *y* sono collocate le feature, ordinate in base alla maggior importanza "media", del nostro modello e dataset, e sull'asse *x* è collocato il relativo valore

di SHAP. Esso spiega quanto una determinata caratteristica può influire nella classificazione. La scala di colore blu-rosso corrisponde invece al valore che assume nel dataset di test quella caratteristica, rosso è pari a un valore alto e blu basso.

Nel nostro caso abbiamo stampato il grafico impostando come riferimento la classificazione benigna, dunque un valore di SHAP positivo va ad indicare che quel parametro influisce positivamente sull'output finale.

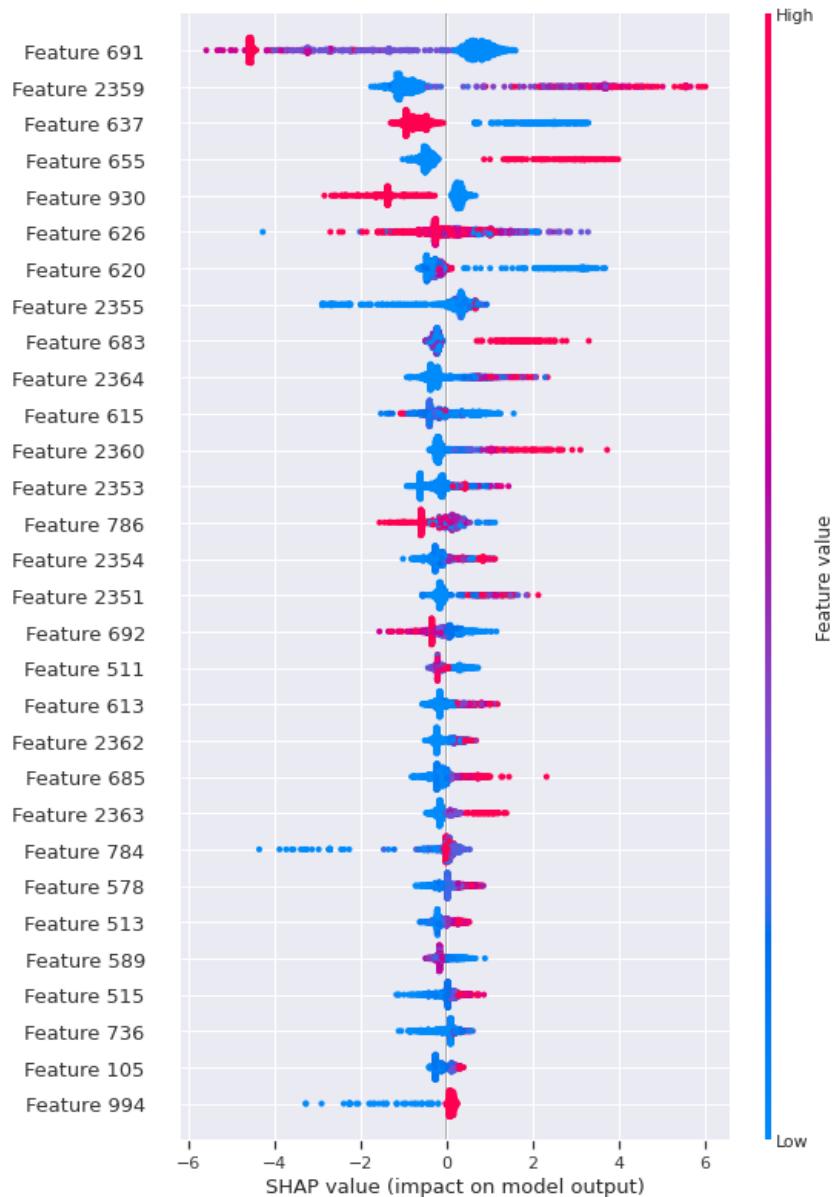


Figura 3.1: SHAP values for the test set

Nel dataset EMBER, le feature non hanno una diretta corrispondenza tra il numero

e il significato, questo a causa del *FeatureHashing* e di una costruzione particolare, rendendone dunque più complessa la lettura. Per questo abbiamo costruito una tabella [7], interpretando il codice dell'estrazione delle feature, che permette una facile traduzione.

Da una prima analisi si nota come ci siano una decina di caratteristiche che possono influire molto nella classificazione, ma che allo stesso tempo hanno un range di valori molto ampio: per esempio il *numero di letture in memoria* (feature 691) influisce molto e in maniera "negativa" in presenza di valori alti, spingendo in fase di predizione verso una classificazione come malware. Viceversa la dimensione della *Exception Table* (feature 2359), più cresce e più un software tendenzialmente è valutato come benigno dal modello. Discorso simile si può fare per la feature 637 che si attiva quando l'eseguibile in realtà è un *DLL* e ha la rispettiva flag nelle caratteristiche segnata. La feature relativa al *numero di import* (feature 620) è molto importante, meno se ne verificano e più si tratta di un file non sospetto. Al contrario, per esempio, l'*entropia* dei nomi delle sezioni o la loro *dimensione* (feature 784 e 736) influiscono molto ma solo per quanto riguarda i malware.

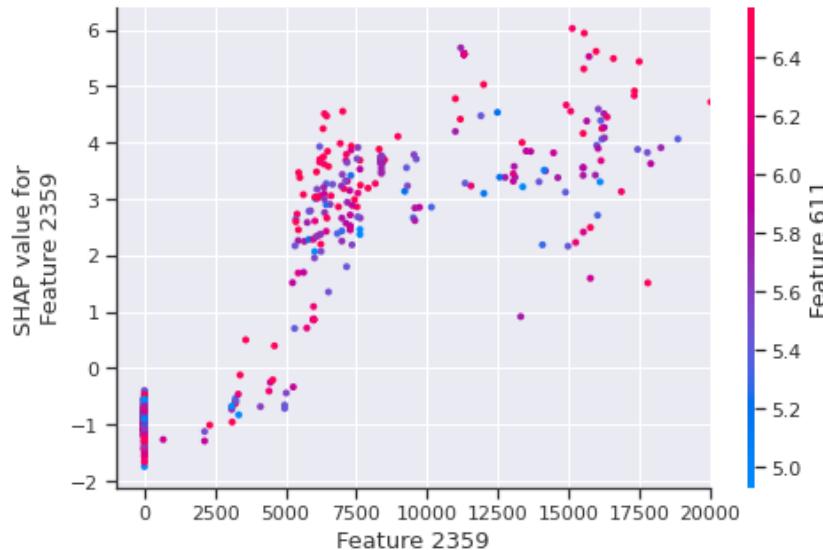


Figura 3.2: SHAP values for Exception Table

In Figura 3.2, si può notare come una dimensione della Exception Table inferiore a 5000 unità influisca negativamente nella valutazione, viceversa un valore maggiore influisce positivamente.

Purtroppo però tutte queste feature risultavano, di primo impatto, complesse da "attaccare" modificando il malware, poiché molte di esse richiedevano una modifica sostanziale del codice e della sua logica. Per esempio, ricompilare il malware come DLL e lanciarlo attraverso un altro eseguibile. Discorso a parte per quanto riguarda l'exception table, ma che richiedeva comunque un lavoro meticoloso perché alcuni malware si basano sul lancio stesso di eccezioni.

Allo stesso tempo risaltava la presenza di alcune caratteristiche che influivano in modo minore, relativamente poco se prese singolarmente, ma sulle quali si poteva eseguire una manipolazione più facilmente. In tal senso, si fa riferimento ai blocchi di feature 0 – 255

che corrisponde, per ogni colonna, alla distribuzione di un determinato byte all'interno del file, 256 – 511 per l'entropia dei byte, 514 – 609 per la quantità di caratteri stampabili nel file ed infine l'elemento 513 per la lunghezza media.

	shap_importance
123	1.751396
128	1.193490
515	1.157254
198	1.122617
255	0.953075
570	0.908505
589	0.898241
578	0.862525
511	0.742040
530	0.700377
32	0.699608
604	0.697117
265	0.696025
256	0.673400
529	0.665869

Figura 3.3: Max SHAP Value 0-609

	mean shap_importance
511	0.226100
578	0.199969
513	0.184760
589	0.182388
515	0.179439
...	...
262	0.002522
281	0.002133
267	0.001980
285	0.001908
270	0.001634

Figura 3.4: Mean SHAP Value 0-609

Sebbene la media dei valori di SHAP (Figura 3.4) di queste caratteristiche sia di 0.017, alcune feature presentano picchi vicini o superiori a 1, come il byte 0x7B.

Anche la dimensione del file (Figura 3.5) gioca un ruolo importante, infatti per valori superiori a 2 megabyte si verifica una crescita lineare dell'importanza "positiva" di questa caratteristica.

Dopo aver analizzato tutte le caratteristiche sopra citate, siamo giunti alla conclusione che probabilmente la soluzione vincente sarebbe stata quella di trovare un modo per influire sulle ultime presentate, piuttosto che concentrarci sulla modifica di alcune feature più delicate come il tipo di subsystem (655 - W32).

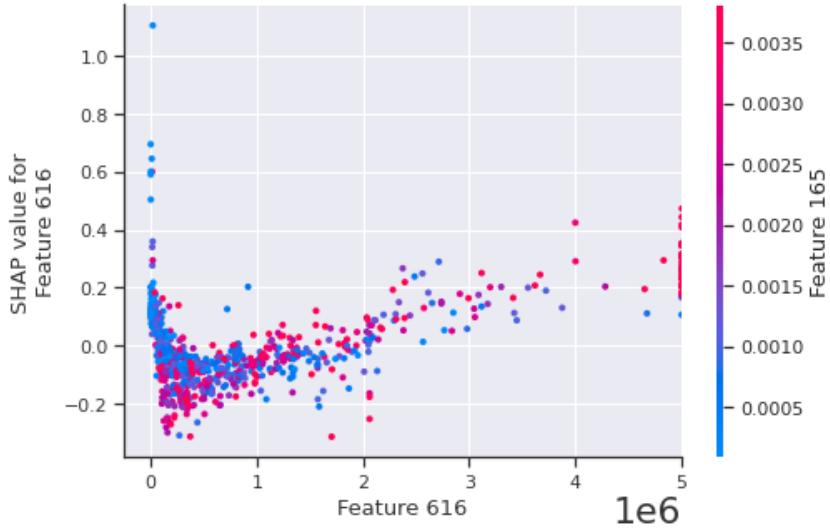


Figura 3.5: Shap Value size

La soluzione che abbiamo pensato è stata quella di applicare il cosiddetto *overlay*, cioè di aggiungere alla fine del file binario, una serie di byte e/o stringhe, prese da un altro binario al fine di influenzare il risultato.

Abbiamo quindi scelto di prendere un eseguibile di grandi dimensioni per mascherare i byte e l’entropia reale del malware. Per tale motivo abbiamo scaricato un eseguibile benigno dalla repository di Microsoft su GitHub, noto come *wingetcreate.exe* dal peso di 31 Mib, dal quale abbiamo estratto le stringhe dei caratteri stampabili e la sequenza di byte che ne compongono il binario.

### 3.2 Fase di attacco al modello: modifica del malware

Come malware di riferimento abbiamo scelto di utilizzare il ransomware noto come *Jigsaw*. Tale scelta è stata effettuata anche per merito della semplicità nel verificare che un’eventuale modifica non andasse ad influire sul suo funzionamento. L’eseguibile è stato scaricato tramite la repository Github di TheZoo [12].

In particolare, tramite il modello LightGBM, Jigsaw veniva classificato come malware con una probabilità del 99.993%. In Figura 3.6 è presentato secondo l’output della libreria SHAP per tale ransomware, dove la linea rossa corrisponde alle feature che spingono per valutazione *malware* e quelle blu per *sw benigno*. Dato che il risultato finale è maggiore di zero esso è stato classificato come *malware*.

Tramite SHAP abbiamo notato che per la classificazione di questo campione veniva prodotto uno valore di SHAP complessivo pari a 9.55. Tale valore, molto elevato, risultava molto difficile da abbassare unicamente con delle modifiche molto semplici, quale la modifica del *timestamp* dell’eseguibile (626) che al massimo poteva incrementare la linea blu di 3 punti (scegliendo un timestamp < 2006).

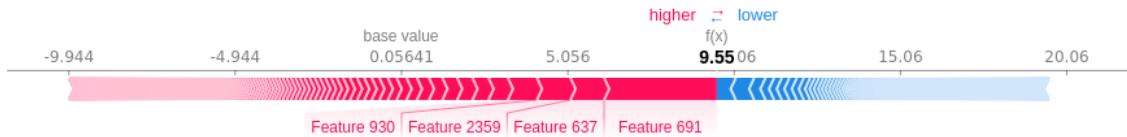


Figura 3.6: Jigsaw Classificazione

shap_importance	shap_importance
<b>691</b>	3.389009
<b>637</b>	1.011091
<b>2359</b>	1.008044
<b>930</b>	0.874718
<b>655</b>	0.581109
<b>2353</b>	0.484496
<b>2364</b>	0.458751
<b>1559</b>	0.443208
<b>620</b>	0.402469
<b>736</b>	0.362906
<b>95</b>	-0.138040
<b>951</b>	-0.187173
<b>2362</b>	-0.209566
<b>2380</b>	-0.224400
<b>1060</b>	-0.227531
<b>128</b>	-0.233034
<b>105</b>	-0.258425
<b>2355</b>	-0.339358
<b>626</b>	-0.551746
<b>578</b>	-0.574214

Figura 3.7: Valori di SHAP per Jigsaw

Andando ad analizzare l’importanza di ogni features (Figura 3.7), si nota come il *numero di letture in memoria* (691), il fatto che l’eseguibile non sia un *DLL* (637) e che abbia una *Exception Table* vuota (2359) vanno a influire per più del 50%. Mentre nella valutazione benigna sono presenti contributi importanti da elementi che si riferiscono alla distribuzione dei byte (features 105, 128, 95) o dei caratteri stampabili (feature 578).

Come primo tentativo abbiamo provato ad eseguire il già citato *overlay*, aggiungendo alla fine dell’eseguibile il binario del file benigno precedentemente scaricato.

Il risultato della predizione era pari a 95%, ancora sopra alla soglia di 83.3% che separa un *malware* da un *non malware*. Però come si nota dal grafico, questa diminuzione di 4% è risultata essere di impatto nell’importanza delle feature, passando da un valore di SHAP di 9.55 a 2.95, un risultato notevole (Figura 3.8).

Come si può facilmente evincere dal grafico SHAP, non ci sono più caratteristiche benigne che spiccano, ma si è verificata l’aggiunta di molte feature che influiscono poco, prese singolarmente, ma che insieme hanno permesso di ridurre di quasi sette punti la classificazione secondo SHAP. Queste caratteristiche, come avevamo previsto, riguardano appunto la distribuzione di byte, l’entropia e i caratteri stampati, oltre alla dimensione del file.

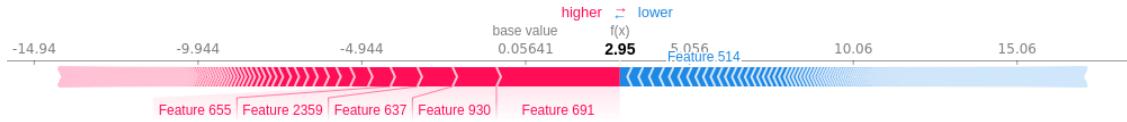


Figura 3.8: Valori di SHAP per Jigsaw con overlay

Infatti, durante la classificazione del ransomware, le prime 609 colonne con l'aggiunta della colonna *size*, influivano a favore di una classificazione benigna per meno di 0.7 come valore di SHAP, dunque erano assolutamente irrilevanti. Mentre ora, grazie all'*overlay* di un file benigno sono andate ad influire di più di 7 punti. Ciò ha confermato che l'applicazione dell'*overlay di un file binario benigno* risulta essere una tecnica efficace; mancavano però ancora dei punti prima di poter arrivare a rendere il malware *sw begnigno* secondo EMBER.

Abbiamo optato dunque per effettuare anche l'*overlay delle stringhe*, ma questo ha influito poco, abbassando la classificazione ad un valore pari a 94.2% e a uno SHAP Value di 2.79.

Il prossimo passo consisteva dunque nel trovare, tra quelle più importanti, una feature con cui potessimo massimizzare di circa 3 punti questa classificazione. Abbiamo perciò scelto di lavorare sul *timestamp* (626) perché una sua modifica avrebbe potuto influenzare la classificazione di quasi 3 punti.

Con la libreria LIEF [10] abbiamo modificato il valore del *timestamp* forzandolo a 0 e grazie a questa modifica siamo riusciti a scendere sotto la soglia del 83.3%, giungendo a un eccellente 57.28%. Ciò significava che il "nostro" malware non veniva più classificato come tale.

Anche tramite SHAP (Figura 3.9) abbiamo avuto conferma di questo effetto, infatti grazie a questa modifica siamo passati a un valore di SHAP di 0.29 (SHAP considera come soglia il 50%).



Figura 3.9: Valori di SHAP per Jigsaw con overlay, string overlay e modifica del tmst

Il nostro obiettivo però rimaneva quello di riuscire a scendere sotto al cinquanta per cento, cercando di andare il più possibile vicino allo zero.

Guardando dunque alle caratteristiche rimanenti, quella che poteva garantirci una forte riduzione del valore della classificazione era la 691, ovvero quella riferita alle *lettture in memoria*. Abbiamo dunque pensato di applicare una compressione al nostro eseguibile, attraverso i numerosi *packer* disponibili, che appunto, vanno a comprimere il codice, oltre a rinominare le sezioni e ridurne dati. Jigsaw, essendo sviluppato con il framework .NET richiedeva l'utilizzo del packer .netshrink [8]. Ma in generale si possono usare anche altri packer generali, come UPX [9]. Oppure anche degli offuscatori di codice binario, come *Babel* o *Obfuscator*.

Applicando unicamente la compressione, il peso del malware passa da 283.5Kb a 262.3Kb, ma la principale modifica ricade nel numero di sezioni che effettuano una lettura RX (MEM\_READ + MEM\_EXECUTE), che passa da 3 a 1. Oltre a modificare il numero di sezioni (da 5 a 3) e modificandone il contenuto. Ciò è osservabile in Figura 3.10.

```
yanez@yanez ~/ember/scripts % master ± python editpe.py jigsaw.exe
!mmUPp 34260 2000 34400 400 0 7.99911 CNT_INITIALIZED_DATA - MEM_EXECUTE - MEM_READ - MEM_WRITE
.text 11878 38000 11a00 34800 0 5.40365 CNT_CODE - MEM_EXECUTE - MEM_READ
.rsrc 650 4a000 800 46200 0 4.25424 CNT_INITIALIZED_DATA - MEM_READ
.reloc c 4c000 200 46a00 0 1.78067 CNT_INITIALIZED_DATA - MEM_DISCARDABLE - MEM_READ
10 4e000 200 46c00 0 2.09192 CNT_CODE - MEM_EXECUTE - MEM_READ
yanez@yanez ~/ember/scripts % master ± python editpe.py jigsaw_compressed.exe
.text 40a44 2000 40c00 200 0 7.93749 CNT_CODE - MEM_EXECUTE - MEM_READ
.rsrc 650 44000 800 40e00 0 4.25667 CNT_INITIALIZED_DATA - MEM_READ
.reloc c 46000 200 41600 0 1.94734 CNT_INITIALIZED_DATA - MEM_DISCARDABLE - MEM_READ
```

Figura 3.10: Sinistra: sezioni Jigsaw.exe Destra: sezioni Jigsaw compresso con .netshrink

Nonostante ciò il risultato della classificazione non cambia di molto, passando da 99.99% a 99.95%. Anche perché, come avevamo già visto, gran parte della valutazione è data da più di duemila caratteristiche, dunque modificandone solamente qualcuna non si ottiene un grandissimo risultato.

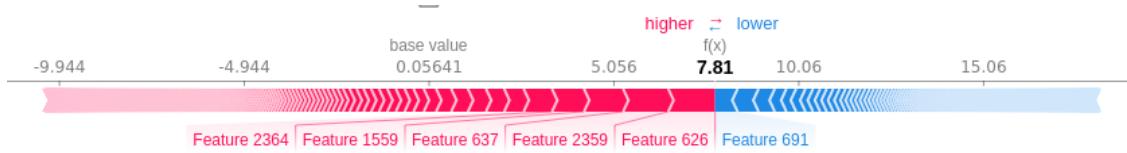


Figura 3.11: Valori di SHAP per Jigsaw compresso con .netshrink

Analizzando le modifiche tramite SHAP (Figura 3.11), si nota che in realtà si è verificata un’importante variazione. Quasi tutte le feature sono rimaste invariate, ma una in particolare ha invertito il suo contributo, ovvero la *feature 691*, che passa da valore circa 3 punti a favore della valutazione come *malware*, a valere circa 0.6 punti ma a favore di un *software benigno*.

A questo punto, dopo aver portato dal lato della classificazione benigna anche questa caratteristica, non ci rimaneva altro che provare ad applicare anche l’*overlay* per cercare di avvicinarci il più possibile verso lo 0.

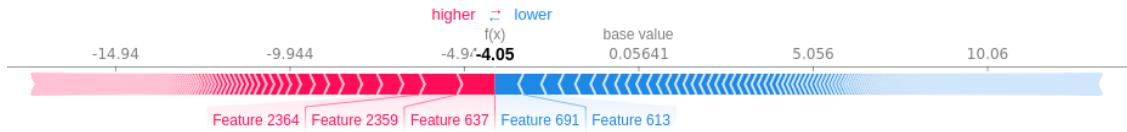


Figura 3.12: Valori di SHAP per Jigsaw compresso con .netshrink + overlay

Applicando unicamente l’*overlay* siamo riusciti ad abbassare di oltre 13 punti, arrivando a classificare il file con una percentuale di 0.017% come *malware*. Questa forte diminuzione è data dal fatto che, al contrario di come accadeva in precedenza, l’importanza della *feature 691* non solo non influiva più in positivo nella valutazione di un software maligno, ma partecipava in forza per la valutazione come *software benigno* (Figura 3.12).

Per abbassare ulteriormente la valutazione abbiamo scelto di applicare anche l’*overlay dei caratteri stampabili* e la modifica del *timestamp*, riuscendo ad arrivare a una classificazione del 0.00019%. In Figura 3.13 è possibile osservare il risultato di tutte le modifiche sui valori di SHAP.

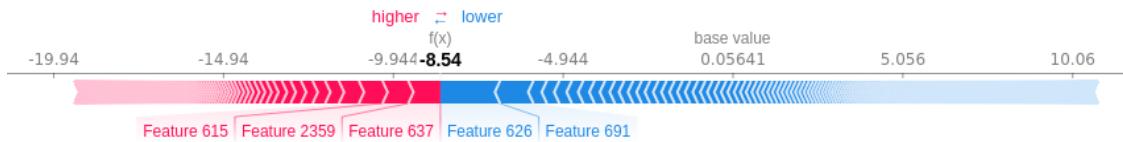


Figura 3.13: Valori di SHAP per Jigsaw compresso con .netshrink + overlay binario, string e modifica timestamp

A questo punto rimaneva unicamente da verificare il funzionamento del codice. Ciò non era scontato, dato che, sebbene la compressione, l’overlay e il timestamp generalmente non influiscano sul funzionamento, in alcuni casi la modifica del binario potrebbe danneggiare il codice, rendendo per esempio invalidi determinati certificati o controlli.

Com’è possibile notare dalle Figure 3.14 e 3.15, sia la versione compressa che quella non compressa, una volta applicati l’overlay e la modifica del timestamp continuano a funzionare (sulla sinistra è possibile vedere *Jigsaw* in esecuzione su VM Win 10). Non a caso lo stesso *VirusTotal* continua a riconoscere il campione come malware (sebbene con meno antivirus) e anche Windows con la sua protezione real-time lo individua come file dannoso.

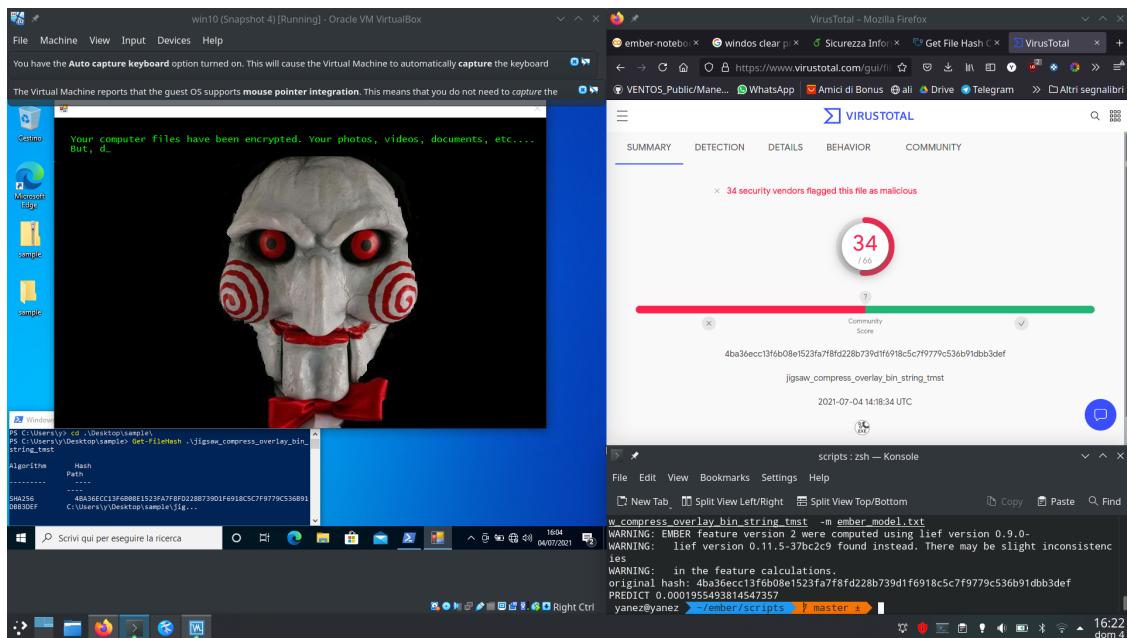


Figura 3.14: Test Jigsaw compresso con .netshrink + overlay

### 3 – Come abbiamo realizzato l'attacco

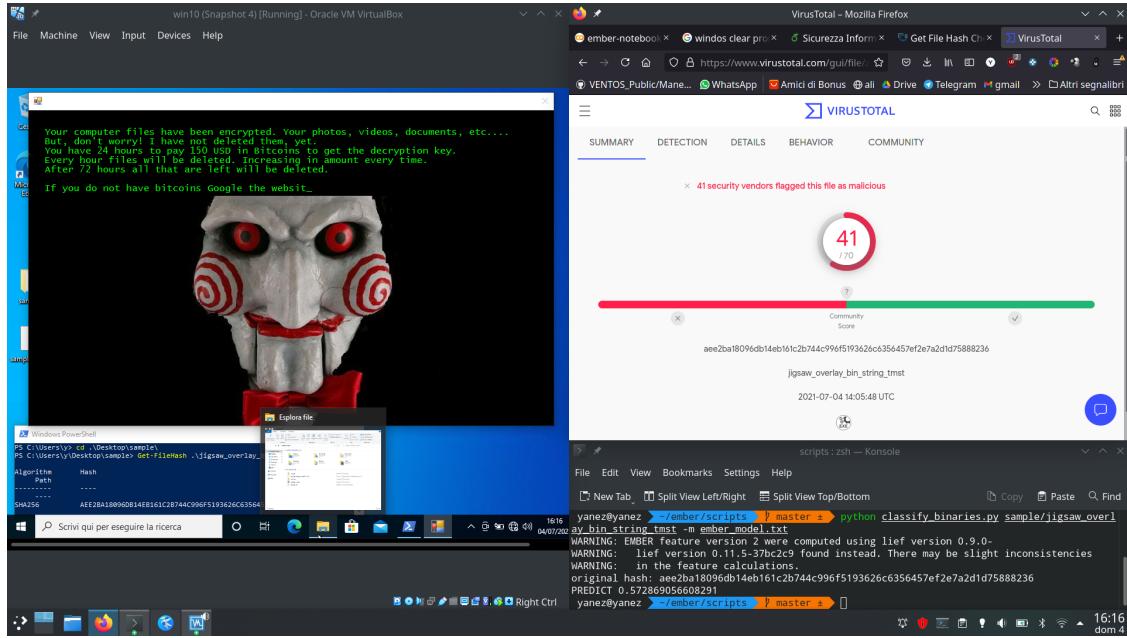


Figura 3.15: Test Jigsaw + overlay

### 3.3 Test ulteriori

Questi test sono stati ripetuti con altri malware, anche più recenti, come il ransomware *Cerber*, il virus *Windows Update* e il Trojan *Dark Tequila*. Sui primi due, per raggiungere una classificazione vicina allo zero è bastato applicare l'*overlay*. Mentre sul Trojan, che risultava essere già compresso con un altro software, è stato prima decompresso e ricompresso con UPX, per poi subire anche questo un attacco con l'*overlay*. In tutti in tre casi, l'attacco ha avuto successo.

Di seguito, nelle Figure 3.16 e 3.17, vengono presentati i risultati della classificazione prima e dopo l'applicazione dell'*overlay*.

```
yanez@yanez ~ /ember/scripts ✘ master ± ➤ python classify_binaries.py -m ember_model.txt power.exe
WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.11.5-37bc2c9 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
original hash: 9c3f8df80193c085912c9950c58051ae77c321975784cc069ceacd4f57d5861d
PREDICT 0.996095869222039
yanez@yanez ~ /ember/scripts ✘ master ± ➤ python classify_binaries.py -m ember_model.txt cerber.exe
WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.11.5-37bc2c9 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
original hash: b3e1e9d97d74c416c2a30dd11858789af5554cf2de62f577c13944a19623777d
PREDICT 0.9986601241390374
yanez@yanez ~ /ember/scripts ✘ master ± ➤ python classify_binaries.py -m ember_model.txt tequila.exe
WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.11.5-37bc2c9 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
original hash: a87b0369f72c2bad163dc7de3afee23c7277cc226429c7e78b97def8a60aa400
PREDICT 0.9989760399982125
```

Figura 3.16: Classificazione malware non modificati

```
yanez@yanez ~ /ember/scripts $ master ± python classify_binaries.py -m ember_model.txt tequila_attacked.exe
WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.11.5-37bc2c9 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
original hash: cf2e532ce02708c0505122535b5caf24926d5442198618192dfc51e367a19dbe
PREDICT 0.03826262653855719
yanez@yanez ~ /ember/scripts $ master ± python classify_binaries.py -m ember_model.txt power_attacked.exe
WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.11.5-37bc2c9 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
original hash: 47591bfbbb872e96b0875a33326f690ec6b89d8d7fe56cde0d6c0c6828910dc3
PREDICT 0.09134049659039142
yanez@yanez ~ /ember/scripts $ master ± python classify_binaries.py -m ember_model.txt cerber_attacked.exe
WARNING: EMBER feature version 2 were computed using lief version 0.9.0-
WARNING:   lief version 0.11.5-37bc2c9 found instead. There may be slight inconsistencies
WARNING:   in the feature calculations.
original hash: 809e7554e61256cf2b69de5a32c79cffb7fc1e0af763df7ce1630e43488ff556
PREDICT 0.0002409734604466563
```

Figura 3.17: Classificazione malware modificati

In generale, partendo da malware classificati con una percentuale media superiore a 99.75%, si è passati a un risultati nell’ordine del  $10^{-2}$ .

# Capitolo 4

## Possibili contromisure

Come descritto in precedenza, il lavoro che abbiamo svolto, per ottenere una modifica del codice del malware che ne mantenesse il funzionamento sovvertendo l'output del modello, prevede principalmente le tecniche di *overlay* e *packing*. Nelle sezione seguenti vengono descritte possibili contromisure per queste tecniche e per gestire lo sbilanciamento del dataset verso campioni relativi a versioni obsolete di Windows.

### 4.1 Contromisure per la tecnica di *overlay*

L'*overlay* permette di modificare le feature relativa alla distribuzione di un determinato byte all'interno del file e la sua entropia, che pur essendo singolarmente poco rilevanti, se ne si esegue la modifica in blocco si verificano effetti molto importanti sui risultati della classificazione. Inoltre permette di modificare la dimensione del file, feature altrettanto importante nella classificazione di quest'ultimo.

Per evitare questo comportamento si potrebbe applicare la tecnica nota come **feature squeezing** [11], che permette di ridurre il numero di queste feature. Questa tecnica è utilizzata per individuare gli attacchi tramite *adversarial examples*, che spesso mettono in difficoltà i modelli di deep learning. Per evitare questo si utilizza un secondo modello, a cui vengono rimosse alcune feature e che abbia delle prestazioni di classificazione su esempi legittimi equiparabili a quelle del modello originale. Ogni esempio in input viene fornito in ingresso a entrambi i modelli e se la classificazione di questi è diversa allora probabilmente l'input è un adversarial sample. Nel nostro caso potremmo eliminare le feature riferite alla presenza di byte e alla loro entropia all'interno del file, nonché quelle riferite a flag come "HAS\_DEBUG" e "EXCEPTION\_TABLE", che da uno studio dei PE presenti in EMBER non ci sembravano feature fondamentali, ma la loro modifica pesa sulle prestazioni di classificazione.

La scelta di ulteriori feature da conservare può essere guidata dalla tecnica di **Principal Component Analysis** (PCA), che permette di determinare le  $k$  principali feature del dataset (mantenendo una percentuale di varianza scelta), poi utilizzabili per la costruzione del secondo modello.

Un'ulteriore tecnica potrebbe essere quella del feature hashing[5], o **hashing trick**, per la quale si possono comprimere queste feature, trovate in gran numero e la cui somma

dei pesi non è ininfluente, in modo da mapparle in un vettore indicizzato attraverso una funzione di hash, permettendo così di ridurre l'impatto del modello nel caso della modifica di queste.

## 4.2 Contromisure per la tecnica di *packing*

La procedura di **packing** fa sì che la struttura in byte del file venga stravolta tramite operazioni di compressione, cifratura e/o modifica del formato, permettendo al malware di evadere più facilmente il modello.

Una possibile contromisura a questa tecnica è l'**adversarial training**, ovvero arricchire il dataset del modello introducendo i PE non solo in formato originale, ma anche dopo varie procedure di *packing*, così da renderlo più robusto a eventuali modifiche della struttura del PE.

## 4.3 Contromisure per le vulnerabilità del dataset

Inoltre, durante l'analisi delle feature più vulnerabili del dataset abbiamo notato che tra queste la *major system version* e il *Timestamp* sono particolarmente critiche. Questo è dovuto al fatto che la stragrande maggioranza dei PE con cui è stato allenato EMBER si compone di file eseguibili relativi a sistemi più obsoleti come Windows XP e Vista e che i malware utilizzati provengano da un dato intervallo temporale. Per tale motivo il modello potrebbe risultare più debole contro malware più moderni, o con timestamp molto vecchi. Pertanto sarebbe opportuno eseguire una procedura di *data augmentation* sui dati attuali e/o aggiungere nuovi campioni più recenti al dataset originale.

# Capitolo 5

## Conclusioni

Nel complesso abbiamo visto che, nel caso di utilizzo di EMBER come dataset, per rendere vulnerabile un modello è sufficiente applicare un attacco che preveda l'aggiunta di un file benigno di grandi dimensioni alla fine dell'eseguibile stesso (overlay). Con molti malware, soprattutto più recenti, non è necessario applicare ulteriori attacchi, come la modifica del timestamp o la compressione. Non escludiamo che in alcuni casi sia necessario effettuare ulteriori modifiche, operando direttamente sul codice sorgente, ma da come si evince tramite SHAP, ci sono molte caratteristiche facilmente modificabili senza il rischio di danneggiare l'eseguibile.

Il dataset di EMBER purtroppo risulta abbastanza datato e ricco di vulnerabilità, abbiamo evidenziato delle possibili contromisure per renderlo più solido, ma che diventerebbero vane se non avvenisse un aggiornamento del dataset con nuovi malware.

In conclusione il nostro approccio non permette comunque di bypassare un analisi di tipo dinamico, infatti sia *VirusTotal* che *l'antivirus di Windows 10* riescono, non solo a determinare che siamo di fronte a un malware, ma anche al tipo e addirittura in alcuni casi al nome stesso. Sicuramente una sviluppo futuro potrebbe essere quello di implementare un attacco sfruttando software come *Veil* [3], che vanno a modificare il payload, dunque rendendo l'attacco robusto anche ad un'analisi dinamica.

# Bibliografia

- [1] H. S. Anderson e P. Roth. «EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models». In: *ArXiv e-prints* (apr. 2018). arXiv: [1804.04637 \[cs.CR\]](https://arxiv.org/abs/1804.04637).
- [2] BetterDataScience. *SHAP: come interpretare i modelli di machine learning con Python*. URL: <https://ichi.pro/it/shap-come-interpretare-i-modelli-di-machine-learning-con-python-38637352733673>.
- [3] ChrisTruncer. *Veil: a tool designed to generate metasploit payloads that bypass common anti-virus solutions*. URL: <https://github.com/Veil-Framework/Veil>.
- [4] Microsoft Corporation. *LightGBM: a gradient boosting framework*. URL: <https://lightgbm.readthedocs.io/en/latest/>.
- [5] scikit-learn developers. *FeatureHasher*. URL: [https://scikit-learn.org/stable/modules/generated/sklearn.feature\\_extraction.FeatureHasher.html](https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.FeatureHasher.html).
- [6] Scott Lundberg. *SHAP documentation*. URL: <https://shap.readthedocs.io/en/latest/index.html>.
- [7] Serina Papa Parolin. *EMBER features table*. URL: [https://docs.google.com/spreadsheets/d/1ARw3CSJznwa\\_0WQC\\_Vh9meTKP1qmt\\_rvLqoM8bwppZY/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1ARw3CSJznwa_0WQC_Vh9meTKP1qmt_rvLqoM8bwppZY/edit?usp=sharing).
- [8] PELOCK. *.netshrink.exe packer*. URL: <https://www.pelock.com/products/netshrink>.
- [9] UPX Team. *UPX: the Ultimate Packer for eXecutables*. URL: <https://upx.github.io/>.
- [10] Romain Thomas. *LIEF - Library to Instrument Executable Formats*. <https://lief.quarkslab.com/>. Apr. 2017.
- [11] Evans David Weilin Xu e Yanjun Qi. «Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks». In: *ArXiv e-prints* (feb. 2018). arXiv: [1704.01155 \[cs.CR\]](https://arxiv.org/abs/1704.01155).
- [12] 5fingers Yuval Nativ Lahad Ladar. *theZoo - A Live Malware Repository*. URL: <https://github.com/ytisf/theZoo>.