Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

STUDIO E SVILUPPO DI TECNICHE DI PASSWORD CRACKING TARGETTIZZATO

Presentata da:
Parrinello Angelo

Relatore:
Lumini Alessandra

Sessione II
Anno Accademico (2020-2021)

"L'uomo rimane il più straordinario dei computer."
John Fitzgerald Kennedy

Introduzione

L'autenticazione basata su password rimane il metodo più comune per accedere a servizi e dati di qualunque tipo grazie alla sua semplicità: presenta però molti difetti ben documentati. I progressi tecnologici hanno fatto sì che l'accesso via password diventasse più robusto; l'anello più debole resta il fattore umano. L'avvento di app, che permettono di salvare e generare automaticamente le password all'interno dei propri dispositivi, non ha comunque fermato le persone dal crearle in maniera da essere facili da ricordare e da scrivere. Questo comporta l'uso di termini comuni alla persona come parole generiche (amore, gelato, ecc.) ma anche *informazioni personali*. Questa caratteristica risulta una potenziale vulnerabilità in caso di attacco informatico.

Lo scopo della mia tesi è quello di studiare, analizzare e sviluppare tecniche di *password cracking targettizzato*. Questa tecnica consiste nell'attaccare una password profilando un target, arrivando quindi a dati personali più o meno sensibili, da sfruttare in tale offensiva. Le password e i dati utilizzati in questa sede sono frutto di ricerche su internet riguardanti possibili data leak, ovvero set di informazioni arrivate online in seguito ad un attacco informatico. Oltre ai vari test, su strumenti all'attuale stato dell'arte, ho sviluppato un tool per generare password a partire da dati personali. Per la fase sperimentale sono stati raccolti molti dataset ed è stato definito un protocollo che ha consentito di valutare e confrontare numerosi strumenti, in modo da selezionare quello più adatto ad un attacco alle password targettizzato.

Questo documento è organizzato in cinque capitoli. Nei primi due capitoli si affronta un'analisi dell'attuale stato dell'arte nel contesto del password cracking: prima approfondendo alcune tecniche tradizionali poi studiando attacchi basati su idee e modelli ben più moderni. Nel primo capitolo l'analisi ha un'impronta di cracking molto più generica mentre nel secondo mi focalizzo sull'identificazione di un target. Il terzo capitolo esamina nel dettaglio le tecnologie utilizzate in questa tesi, dividendole in due macro categorie. Il quarto capitolo descrive la parte sperimentale, includendo lo studio dei dataset, la definizione dei protocolli di testing, la descrizione dei risultati degli esperimenti ed infine i commenti. Il quinto capitolo conclude la tesi trattando gli obiettivi raggiunti e i possibili sviluppi futuri.

Indice

Capitolo 1

Analisi dello stato dell'arte

Gli scopi di questa tesi sono lo studio e il confronto dei principali mezzi per il password cracking con il supporto di informazioni personali relative ad un soggetto target dell'attacco; svilupperò anche un tool in Python per la generazione di possibili password sulla base di parole chiave legate all'utente identificato. E' importante per i miei scopi partire con lo studio dello stato dell'arte all'atto della stesura della tesi, per riuscire a contestualizzare al meglio il mio progetto.

In questa prima fase spiegherò meglio le principali strategie di *password cracking generico*: esse si applicano largamente anche a quello *targettizzato*. Il password cracking viene definito generico quando non utilizza informazioni di nessun tipo legate alla persona che possiede la password obiettivo; mentre si parla di password cracking targettizzato quando un attacco può sfruttare informazioni personali legate all'utente (nome, cognome, città, ecc.).

1.1 Introduzione al Password Cracking

L'autenticazione ad un sistema attraverso una password é uno dei piú comuni mezzi di accesso. La fase di login con password ad un sistema esegue una comparazione tra la password inserita e la password salvata; al fine di evitare accessi inattesi le password salvate nel sistema vengono quindi criptate attraverso una qualche funzione hash (MD5, SHA-1, ecc.): il *Password Cracking* ha lo scopo di riuscire a recuperare una password in chiaro partendo da una posizione (online o offline) che contiene la password criptata . Un attaccante alla password cercherá di indovinare la password corretta finché non riuscirá nel suo intento oppure verrà fermato in qualche maniera (arresa spontanea o numero limitato di tentativi): il difensore dovrá cercare di costruire una password piú robusta possibile. Se le persone peró fossero in grado di memorizzare, ogni volta che si crea una password, informazioni casuali e uniche allora recuperare una password sarebbe un compito assai arduo. Ma per (s)fortuna cosí non

é. La maggior parte delle persone nel momento in cui creano una nuova password tendono ad usare pattern ben definiti come:

- **Uso di password comuni**. Numerosi studi (ad esempio [3]) hanno dimostrato come l'utente tenda ad utilizzare password estramemente comuni oppure scelga password che soddisfano i requisiti minimi di sicurezza di un determinato sistema (es:1234567a).

- **Riciclo di password**. Recenti ricerche hanno dimostrato come le password di una persona tendano ad essere sempre le stesse: è stato stimato che in media ogni utente abbia 26 iscrizioni online ma solo 5 password diverse [17]. Con l'aumentare degli account online è sempre più difficile per una persona mantenere il controllo di tutte le password: quindi si riutilizzano dando la possibilità ad un malintenzionato di sfruttare questa debolezza.

- **Inserimento di dati personali**. Castelluccia et al. [6] e altre ricerche ancora hanno notato come l'utenza media del web utilizzi informazioni personali come data di nascita, nickname, ecc. per creare una password: questo fatto risulta essere estremamente rilevante per un persona che se ne vuole impossessare.

Il *riciclo di password*, il *riciclo di vocaboli (personali e non)* o altro ancora sono in generale accompagnati ad una certa strategia fissa nella composizione di una password (es: parola + anno + numero casuale) [34]; mentre le persone rimangono ancorate ai loro metodi per tale creazione, le tecniche di cracking migliorano costantemente. Gli attacchi contro le password possono assumere molteplici forme ma in generale si possono raggruppare questi metodi in due macro insiemi: *Attacco online* e *Attacco offline*.

1.2 Attacco Online e Offline

Gli *Attacchi online* sono quella categoria di attacchi alle password in cui l'attaccante non ha accesso diretto alle credenziali. Il classico esempio é un'offensiva ad un sistema connesso ad internet attraverso un banale form di login. In questo caso il difensore ha il controllo sul numero di tentativi di accesso al sistema, limitando quindi i tentativi da parte dell'attaccante. Questo ovviamente non fermerá i maleintenzionati o i penetration tester: dovranno però saper gestire l'attacco utilizzando tipicamente solo qualche milione se non migliaia o addirittua centinaia di tentativi totali. Questo metodo risulta lento, inefficiente (perché richiede la risposta online del servizio interrogato): diverse volte non si conosce nemmeno il nome utente di chi si sta prendendo di mira (la complessitá di un attacco simile puó essere davvero elevata [1]).

Invece gli *Attacchi offline* vengono effettuati una volta che si é recuperata la password hashata. Ció puó avvenire in molteplici modi:

- Compromissione del sistema
- Estrazione del DB
- Intercettazione della password attraverso una connessione (tipicamente da rete Wi-Fi).

Questo attacco, come si intuisce facilmente, é il piú utilizzato dato che non pone limiti di tentativi o tempo. La principale differenza tra i due attacchi risiede quindi nel possesso o meno di informazioni relative alla password.

1.3 Tecniche Tradizionali di Password Cracking

Queste tecniche includono il furto, la frode e lo studio della persona.

1.3.1 Ingegneria Sociale

L'ingegneria sociale é l'arte di manipolare, usufruire e frodare persone per ottenere informazioni utili (nel nostro caso password). Si parte con il raccoglimento di informazioni personali sulla vittima in molteplici modi come l'uso di Social Media, tecniche OSINT in generale. Le tecniche OSINT vengono utilizzate per ricercare, raccogliere ed analizzare dati e notizie d'interesse attraverso fonti aperte pubbliche [8].
In un secondo momento si passa alla verifica della correttezza dei dati (arrivando anche a chiamare il target) per poi preparare l'attacco effettivo. Tra le piú famose tecniche di Social Engineering troviamo lo Shoulder Surfing, il Baiting (offrire qualcosa per riuscire a far scaricare un file dannoso), il Phishing (truffa che si concretizza per mezzo di messaggi, e-mail, ecc.), il Pretexting (il target viene contattato telefonicamente).

1.3.2 Shoulder Surfing

Il furto della password puó avvenire in molteplici modi, anche banali, come guardare nella scrivania della persona o inserendosi nella connessione di una rete, ma in primis attraverso lo *Shoulder Surfing*. Questa tecnica, che non richiede tipicamente nessuna competenza informatica, consiste nello spiare un utente al fine di ottenere una password o informazione segreta. Questa abilità può essere usata ad esempio durante l'inserimento di un PIN al supermercato, in autobus mentre viene digitata la password per Facebook o in altri posti ancora: all'attaccante basterà spiare la persona target, senza farsi notare, per riuscire ad ottenere ciò che desidera. Lo shoulder surfing avviene tipicamente in luoghi affollati per la

semplicitá di osservazione sul target senza peró attirare l'attenzione [12], ma puó avvenire anche a stretto contatto con la persona. Se é vero che questa particolare attivitá puó non richiedere conoscenze informatiche, é altresí vero che molto spesso vengono usate tecnologie adatte a far ció (microspie, nano-micorfoni, ecc.).

1.3.3 Algorithm Analysis

Gli attacchi dove viene effettuata un *Analisi Algoritmica* si concentrano sugli algoritmi di crittografia utilizzati in una trasmissione: sfrutta le caratteristiche degli algoritmi per tentare di dedurre uno specifico testo in chiaro o le chiavi. In generale l'analisi sfrutta una conoscenza di base della natura dell'algoritmo, una certa conoscenza delle caratteristiche del testo e qualche tupla testo cifrato-testo in chiaro per confrontare ció che si é acquisito.

1.3.4 Fully Guesting

Il tipo di metodo peró piú utilizzato per attaccare una password (molte volte utilizzato assieme o dopo le tecniche sopra citate) é il *Fully Guesting*. Questo metodo in realtá contiene molti tipi di attacchi possibili:

Brute Force Attack

L'attacco di tipo *Brute force* coinvolge l'attaccante nel tentare ogni possibile combinazione per poi scoprire la password in plain-text. Viene usato anche in combinazione ad altri attacchi fully guesting nella ricerca della soluzione a password brevi. In generale il metodo consiste nel calcolo dell'hash della password che si tenta: poi si confronta il risultato con l'hash di destinazione. In caso di successo l'attacco si ferma altrimenti il processo continua tentando una nuova password. Se da un lato risulta essere un tipo di attacco molto banale dall'altro porta ad un consumo importante di risorse, soprattutto nel caso di password molto lunghe. Questo tipo di attacco utilizza solitamente un charset composto da lettere tutte maiuscole, tutte minuscole e tutte le cifre: se pensiamo ad una password la cui lunghezza é pari a 9, questo tipo di attacco tenterá 62^9 password. Ipotizzando un velocitá di generazione di hash pari a 100MH/s, impiegheremo piú di 4 anni per completare l'attacco [25].

Dictionary Attack

Nel *Dictionary attack* al posto di provare tutte le possibili parole che si possono costruire con una data grammatica, si sfrutta un elenco di possibili password provando ripetutamente a determinare la chiave di decrittazione di una password target: il termine tecnico adatto è sfruttare un *dizionario*. Ci sono poi molteplici liste di password molto comuni: la

più popolare è Rockyou, una lista di milioni di password più utilizzate al mondo. Allo stato attuale si è arrivati ad avere un file con 82 miliardi di password con il peso di circa 100GB. Questo tipo di attacco quindi è molto più efficiente rispetto ad un semplice brute-force ma non assicura il successo dell'operazione. E' quindi il giusto trade-off tra efficienza ed efficacia.

Hybrid Attack

L'*Hybrid Attack* è una combinazione di entrambi gli attacchi appena visti: l'attacco a dizionario e quello a forza bruta. Per effettuare questo tipo di offensiva a dati sensibili si utilizza un dizionario di parole/password e in seguito si applicano le tecniche di brute forcing a ciascuna possibile password nell'elenco, creando così qualche variazione della parola (come l'aggiunta di un prefisso o suffisso di numeri). Questo metodo di password cracking è in sostanza una variazione del più tradizionale *Combinator Attack*; quest'ultimo attacco consiste nell'unire due dizionari parola per parola, aggiungendo quindi alle parole del primo altre del secondo, arrivando poi ad un terzo dizionario contenente le parole combinate.

Rule-Based Attack

L'attacco *Rule-based* é una delle modalitá di attacco piú complicate in assoluto. Come dice la parola stessa questa procedura permette, partendo da un dizionario di parole e modificandole secondo certe regole, di generare possibili password. Tale difficoltá si incontra nel momento della creazione di *rules* o *regole* per uno specifico attacco: bisogna saper utilizzare questo particolare "linguaggio di programmazione" per portare un attacco soddisfacente. Infatti esiste un linguaggio specifico per impartire degli ordini al programma di cracking. Le rules scelte andranno poi inserite in un file di testo, tendezialmente scrivendo una regola per riga di testo. Esempio: si prende un dizionario molto semplice che contiene solo la parola "Cesena". Se si scrive nel file di testo, contenente le regole, la seguente riga (seguendo le istruzioni di scrittura delle rules per Hashcat [32]):

$1 $2 $3

otterremmo in uscita la parola "Cesena" e "Cesena123". Il risultato si ottiene combinando assieme la parola "Cesena" e la regola scritta sopra, il cui signficato si può tradurre in "aggiungi alla fine delle parole nel dizionario i numeri 123".

Mask Attack

Il *Mask attack* è uno strumento estremamente importante ed utile quando si conosce una parte della password o si conoscono dettagli molto specifici su di essa. Esempio: si consideri il caso di una password composta

da 12 caratteri e terminante con "qwerty". Un attacco che si potrebbe provare a portare contro questa parola è il brute force ma è ovvio che una strategia così non conviene affatto su una parola lunga 12 caratteri. Tutto ciò che serve in questo caso è indovinare i primi 6 caratteri della password cercata. Ecco a cosa serve il Mask attack. Continuando l'esempio, un possibile attacco alla parola potrebbe avere in un tool di password cracking come Hashcat [31], una sintassi simile

?c?c?c?c?c?cqwerty

Questo comando creerà così tutte le possibili password che hanno come prime sei lettere una lettera minuscola compresa dalla a alla z e come ultime 6 lettere la parola qwerty (come aaaaaaqwerty, aaaaabqwerty ecc.). La ragione principale quindi per usare un attacco del genere e non un semplice brute-force è l'enorme riduzione del *keyspace* ovvero il numero totale di password possibili da generare. In più in un attacco simile si possono utilizzare alcuni pattern che le persone tendono ad usare in fase di creazione delle password.

Capitolo 2

Targetting Password Cracking

Il *Targetting Password Cracking* è una branca del Password Cracking che si occupa di recuperare una determinata password col sostegno di informazioni personali legate alla vittima. Questo capitolo si pone lo scopo di spiegare le caratteristiche di un attacco ad una password con l'aggiunta di dati sul target partendo dall'analisi di un attacco di questo tipo per arrivare poi alle più recenti tecniche per attuare ciò.

Questo documento analizzerà attacchi di tipo *offline*, ma va sottolineato come questo tipo di attacco targettizzato sia applicabile anche contro sistemi collegati in rete.

2.1 Analisi degli attacchi Targettizzati

Un cyber attacco targettizzato viene definito tale dal momento in cui si sferra con l'ausilio di informazioni extra strettamente legate al target (persona, società, ecc.), a differenza quindi dei virus informatici che attaccano simultaneamente più sistemi, molte volte senza un vero e proprio destinatario. Questi attacchi, specificatamente progettati per colpire un bersaglio, sono denominati *Advanced Persistent Threat* o *APT* (Minaccia Persistente Avanzata) [24]. L'acronimo APT rappresenta una minaccia portata avanti con costanza nel tempo da un soggetto in possesso di numerose skills ed, eventualmente, anche un numero sostanzioso di risorse [9]. E' proprio il tempo una delle risorse più delicate di questo tipo di attacco: l'attaccante mira ad estendere questo attacco nel tempo, cercando di recuperare più informazioni possibili (proprio per tale ragione molte volte la P di Persistent viene sostituita dalla P di Patience o Pazienza, a causa della pazienza dell'attaccante nel persistere nel suo gesto). C'è un certo accordo nella comunità sul fatto che questi attacchi non siano necessariamente più avanzati di altri, tranne quando l'attacco prende di mira un target di alto valore: ciò richiede molte più abilità.

Si possono individuare tre fasi in un attacco informatico targettizzato:

- **Intelligence Gathering/Raccolta di informazioni**. La raccolta iniziale di informazioni utili è il punto di partenza. Senza dati grezzi è impossibile elaborare informazioni sul target per prendere decisioni future: la probabilità di successo è direttamente proporzionale alla quantità di dati raccolti. Le risorse disponibili pubblicamente attraverso i metodi OSINT sono una buona base, hanno però il difetto di essere sensibili al tempo (quindi che variano col passare del tempo) e non essere molto accurate in certe occasioni. Risulta necessario il controllo di questi dati fino ad attacco completato. Più in generale l'ingegneria sociale gioca un ruolo fondamentale in questa prima fase. La profilazione del target può contenere anche interrogazione di archivi pubblici o la ricerca di vulnerabilità su siti web/database pubblici.

- **Threat Modeling/Modellazione della minaccia**. Una volta raccolte le informazioni necessarie, si passa alla modellazione della minaccia. Si crea quindi un profilo del target e dell'ambiente da colpire (se le informazioni raccolte e le risorse a disposizione lo consentono, si può addirittura ri-creare il sistema da colpire così da testare diverse penetrazioni). In tale fase vengono anche ricercate le relazioni tra gli oggetti che compongono l'ambiente al fine poi di scovare eventuali debolezze (o punti di forza) in un eventuale attacco. Questa fase si conclude quindi dopo aver valutato attentamente i rischi e aver selezionato i possibili attacchi che hanno più possibilità di successo.

- **Attacking and Exploiting Targets/Attaccare e Sfruttare il bersaglio**. L'ultima fase, facilmente comprensibile già dal titolo, è quella in cui l'attaccante lancia l'attacco basandosi sul modello creato in precedenza. In linea di massima questa fase si concretizza con il caricamento di un malware, ovvero un software creato per arrecare danno a dispositivi e sistemi informatici, agendo contro l'interesse degli utenti [33], in un calcolatore delle rete target per poi estrarre le ricchezze a cui si punta.

Si puó intuire quindi che questi attacchi, coi giusti requisiti, possono portare grossi risultati ed é per tale motivo che sono molto usati soprattutto per scopi di lucro o per rubare segreti industriali.

L'attacco informatico targettizzato si svolge quindi attraverso tre fasi, composte a loro volta da diversi momenti e processi da eseguire. L'atto del password cracking fa parte di queste azioni: si analizza ora in quali fasi dell'attacco si puó incontrare.

- **Durante la prima fase di Ricognizione dei dati**. Nel momento in cui ci si adopera per scovare dati ci si può imbattere in debolezze di alcuni sistemi della rete target. Se ció avvenisse, sarebbe

possibile a volte entrare in possesso di grosse quantitá di dati come password hashate e/o informazioni personali. Quindi nel caso in cui ci si imbattesse in password hashate si procederebbe con il cracking di esse.

- **Durante la terza fase di Attacco al bersaglio**. Il malware, che ha infettato un computer, estrae da esso informazioni segrete. Se tra queste fossero contenute password non in chiaro allora andrebbero riportate alla versione originale.

Entrando in possesso di alcune informazioni personali é facile imbattersi nelle relative password associate: il *password cracking targettizzato* parte da qui. Con l'utilizzo dei corretti strumenti uniti alle informazioni personali é possibile ridurre il consumo di risorse nel recupero di queste password.

2.2 Tecniche Moderne di Password Cracking

Con l'aumentare della consapevolezza dell'importanza delle password (e di conseguenza anche delle tecniche e degli gli strumenti con cui si creano/mantengono), é stato necessario progettare nuove tecniche di password cracking, che si possono affiancare alle piú classiche giá viste. Alcuni algoritmi o modelli sono importati da altre letterature per poi essere combinati con le metodologie di cracking delle password esistenti: come il modello matematico di Markov e la grammatica linguistica context-free. Altri ancora utilizzano il Machine Learning e addestrano un modello su password/keywords esistenti. A differenze delle tecniche classiche, che comunque molto spesso vengono affiancate a questi metodi moderni, queste sono in grado di utilizzare in maniera intelligente le parole che prendono in ingresso: ciò vuol dire che delle ipotetiche parole specifiche per il target possono aumentare la probabilità di successo.

2.2.1 Markov Chains Attack

I Modelli di Markov si trovano in numerosi applicazioni legate alla computer security ed in particolare alla password security. Esempi del loro impiego sono tool per craccare le password come **OMEN** [6] o **NEMO** [22] ma anche uno stimatore della robustezza di una password come **APSM** [7].

Definisco ora una serie di concetti chiave (senza voler entrare nello specifico e affrontare un certo livello di rigore matematico in quanto non è lo scopo di questo documento), per capire al meglio il concetto della *Catena di Markov*:

- **Successione**. Una successione può essere definita intuitivamente come un elenco ordinato costituito da un'infinità numerabile di oggetti, detti termini della successione.

- **Spazio di probabilità**. Uno spazio di probabilità è una terna (Ω, A, P), dove Ω è un insieme, A è una famiglia coerente di eventi su Ω e P una probabilità su A.

- **Variabile aleatoria**. Sia (Ω, A, P) uno spazio di probabilità. Una funzione $X : \Omega \to R$ é una variabile aleatoria se

  $$\omega \in \Omega : X(\omega) \leq a$$

  è un evento per ogni a $\in$ R.
  Una variabile aleatoria è dunque un numero che viene assegnato, mediante una determinata regola, a ciascun punto dello spazio di probabilitá.

- **Matrice di Transizione di Markov**. Una Matrice di Transizione è la matrice generata dalla probabilitá di transizione da uno stato all'altro del nostro sistema. Come matrice contenente delle probabilitá ogni elemento di essa sarà compreso tra 0 e 1. E in secondo luogo la somma di ogni riga è 1. Prendiamo ad esempio una matrice cosí definita:

  $$\begin{bmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0 & 0.7 \\ 0.5 & 0 & 0.5 \end{bmatrix}$$

  Dove la prima riga e la prima colonna si riferiscono al primo stato del sistema, la seconda riga/colonna al secondo e cosí via. Quindi se prendiamo l'elemento presente nella prima riga e seconda colonna, esso rappresenta la probabilitá di passare dal primo stato al secondo stato.

La *Catena di Markov* é una successione di variabili aleatorie definite su un insieme finito di stati e, ad ogni passo della successione, mi dovrò trovare in uno di questi stati. La probabilitá di passare da uno stato all'altro é data da una *Matrice di Transizione*. Questo fenomeno viene anche detto *Processo stocastico senza memoria* in quanto il "passato" non conta ovvero lo stato successivo dipende solo da quello presente.

Questo concetto viene sfruttato per il password cracking basandosi sul fatto che le lettere adiacenti nelle password generate dall'uomo non sono scelte indipendentemente, ma seguono determinate regole: ad esempio, il 2-grammo *un* è molto più probabile di *uk* e la lettera *a* è molto probabile che segua *un*). In un modello di Markov a n-grammi (sequenze di n parole presenti in un testo), si modella la probabilità del carattere successivo in una stringa in base a un prefisso di lunghezza n-1. Quindi, per una data stringa c_1,..., c_m, un modello di Markov stima la sua probabilità come:

$$P(c_1, ..., c_m) = P(c_1, ..., c_{n-1}) \cdot \prod_{i=n}^{m} P(c_i | c_{i-n+1}, ..., c_{i-1})$$

Nel Password Cracking il funzionamento è il seguente:

- Le probabilità iniziali $P(c_1, ..., c_{n-1})$ e la matrice di transizione $P(c_n | c_1, ..., c_{n-1})$ sono ricavate da dati reali. Il dizionario di password/dati con cui si addestra il modello Markoviano deve essere simile sia a livello di struttura della sintassi delle parole sia a livello di distribuzione di queste parole (più occorrenze ha una parola più importanza avrà).
- Si generano le possibili passwords in ordine di probabilitá decrescente come stimato dal modello di Markov.

Per rendere il più efficiente possibile un attacco simile è importante utilizzare un dataset grande ed avere un algoritmo di ordinamento delle password nell'ordine corretto. L'assenza di un database significativo comporterebbe infatti la bassa precisione nel calcolo delle informazioni vitali dell'algoritmo, come la corretta frequenza delle parole. La mancanza di un buon algoritmo di ordinamento rende invece la ricerca molto lunga. Tale problema è stato risolto in alcuni tool di Password Cracking come John The Ripper [21] ma anche nel Password Guesser OMEN [6], come vedremo nei capitoli successivi.

2.2.2 PCFG Attack

Il Probablistic Context Free Grammar Attack (PCFG Attack, in italiano Attacco con Grammatica Probabilistica Libera dal Contesto) ha come assunzione di base che le strutture delle password abbiano probabilitá diverse o, in altre parole, non tutte le password generate hanno la stessa probabilità di recuperare una password. Come per MCA (Markov Chains Attack), anche nella seguente modalità, alla base dell'attacco troviamo la generazione di possibili password in ordine decrescente di probabilità per massimizzare la verosimiglianza con la password di destinazione.

Il Context-Free Grammar (CFG) é un concetto ampiamente studiato nello studio dei linguaggi naturali dove viene usato per generare stringhe con una certa struttura. Questo approccio è facilmente impiegabile nell' ambiente del Password Generation. In generale definiamo una CFG come

$$G = (V, \Sigma, S, P)$$

dove V è un set di variabili o insieme finito di simboli non terminali, Σ è un insieme finito di simboli terminali, S (è un elemento di V) determina il simbolo di partenza non terminale e P è un insieme finito di regole di produzione di una parola del tipo:

$$\alpha \to \beta$$

dove α è una singola variabile (o simbolo non terminale) mentre β è una stringa composta da variabili o simboli terminali [40]. In pratica un CFG è una grammatica dove ogni regola sintattica è espressa sotto forma di derivazione di un simbolo a sinistra a partire da uno o più simboli a destra. La Grammatica Libera dal Contesto si dice che generi un *Context-Free Language* (CFL): questo linguaggio è l'insieme di stringhe costituito da tutti i simboli terminali derivabili dal simbolo di inizio. Un esempio rapido: vogliamo generare un linguaggio che contenga un egual numero di a e b della forma $a^n b^n$, il CFG allora sarà definito come

$$G = ((S, A), (a, b), S, (S \to aAb, A \to aAb|\emptyset))$$

con (S,A) come set di variabili o non terminali, (a,b) set di valori terminali, S simbolo di partenza e infine $(S \to aAb, A \to aAb|\emptyset)$ l'insieme di regole di produzione (notare come A possa generare sia aAb che l'insieme vuoto). Quindi, ipotizzando n pari a 3:

$$\begin{aligned} S &\to aAb \\ &\to aaAbb \\ &\to aaaAbbb \\ &\to aaabbb \\ &\to a^3b^3 \end{aligned}$$

I *PCFG (Probabilistic CFG)* sono dei semplici CFG ma con associato ad ogni produzione una probabilità in modo tale che per una specifica variabile del lato sinistro tutte le produzioni associate si sommano a 1. Per capire meglio questo concetto ecco alcuni termini che usati, come definito di consueto nei paper che trattano questo argomento:

- Con **L** si indicano le **Alphabet String** ovvero una sequenza di simboli alfabetici.
- Con **D** si indicano le **Digit String** ovvero una sequenza di cifre.

- Con **S** si indicano le **Special String** ovvero una sequenza nè di simboli alfabetici nè cifre.

Ad esempio la parola "ce$ena123" sará identificata dalla *struttura semplice* LSLD. Inoltre viene assegnata ad ogni sequenza della struttura la propria lunghezza (diventerebbe $L_2S_1L_3D_3$ chiamata *struttura base*).

Tipo di Dato	Simbolo	Esempio
Alpha String o L	abcgdefghilmnopqrstuvzw...	informatica
Digit String o D	0123456789	123
Special String o S	!@*− = +...	!!!

Tabella 2.1: Lista delle diverse sequenze con esempio. Notare che i simboli relativi a L variano a seconda della lingua

Definiti questi termini, ecco il funzionamento del PCFG implementato in un attacco ad una password. La fase iniziale, quella di training, verrà effettuata su un set di dati simile il piú possibile in struttura e distribuzione alla/e password target. Gli obbiettivi in questa fase sono derivare un set di regole di produzione che generi la struttura base delle parole e un set di produzione che produca simboli terminali costituiti da cifre e caratteri speciali. Discorso a parte va fatto per le Alpha String. La riscrittura di queste variabili può avvenire usando un dizionario di input simile a quello utilizzato in un attacco di dizionario tradizionale. Un altro approccio è quello di ricavare la probabilità dei caratteri alfabetici in base alla ricorrenza di questi, suddivisi per lunghezza della parola stessa: in pratica si ricava un set di regole anche per le variabili L. Una stringa derivata dal simbolo di partenza è definito come *sentential form/forma sentenziale* (nell'esempio sopra dove si spiegava il funzionamento dei CFG, aaAbb é una sentential form). La probabilità di una forma sentenziale è il prodotto delle probabilitá delle produzioni usate nella sua derivazione. La stima di massima della forma sentenziale derivata prima di associarle una stringa alfabetica viene chiamata *pre-terminal structure/strutture pre-terminali*. Molti framework che utilizzano PCFG (o comunque la struttura delle password simile ad esso) oltre alle variabili L, S e D modellano altri aspetti del linguaggio (date, lettere maiuscole, ecc.) utilizzando ulteriori variabili [35].

Tipo di Struttura	Esempio
Semplice	LSLD
Base	$L_2S_1L_3D_3$
Pre-Terminale	$L_2\$L_3123$
Password Generata	ce\$ena123

Tabella 2.2: Diverse strutture in fasi diverse del processo di Password Generation con esempio.

Nella fase di training si ottiene direttamente un PCFG dal training set. Un esempio di un Probabilistic Context-Free Grammar è di seguito:

Lato Sinistro	Lato Destro	Probabilità
S $\rightarrow$	$D_1L_3S_2D_1$	0.75
S $\rightarrow$	$L_3D_1S_1$	0.25
$D_1 \rightarrow$	4	0.60
$D_1 \rightarrow$	5	0.20
$D_1 \rightarrow$	6	0.20
$S_1 \rightarrow$	!	0.65
$S_1 \rightarrow$	%	0.30
$S_1 \rightarrow$	#	0.05
$S_2 \rightarrow$	\$\$	0.70
$S_2 \rightarrow$	**	0.30

Tabella 2.3: Esempio di un semplice PCFG. Si noti come la somma delle probabilità delle trasformazioni per ogni variabile sia 1.

Con una grammatica simile a quella presentata poc'anzi, è possibile generare,ad esempio, una struttura pre-terminale del tipo:

$$S \rightarrow L_3D_1S_1 \rightarrow L_34S_1 \rightarrow L_34!$$

Con una probabilità di $0.25 \rightarrow 0.25 * 0.60 \rightarrow 0.25 * 0.60 * 0.65 \rightarrow 0.0975$. Una volta generato il nostro PCFG si passa alla fase di generazione password, che saranno utilizzate per attaccare il set di password interessato.

2.2.3 Deep Learning Attack

Con il termine *Machine Learning* ci si riferisce a quel processo tramite il quale un compilatore sviluppa il riconoscimento di modelli/pattern, ovvero la capacità di apprendere dall'esperienza effettuando poi previsioni utilizzando i dati che possiede. Il concetto che si cela dietro a questo tipo di attacco, è una branca del Machine Learing, il *Deep Learning* o *Apprendimento Profondo*. In poche parole il Deep Learing è un insieme di tecniche basate su *Reti Neurali*; queste reti, che si ispirano alle rete neurali biologiche, sono organizzate in diversi strati, dove ognuno di essi si addestra su un dataset affinchè l'informazione venga elaborata in maniera sempre più completa, migliorandone l'accuratezza.

Fondamentalmente l'idea di base è addestrare una rete neurale per determinare in modo autonomo le caratteristiche e le strutture delle password e sfruttare questa conoscenza per generare nuovi campioni che seguano la stessa distribuzione. Le reti neurali sono in grado di catturare una grande varietà di proprietà e strutture che descrivono la maggior parte delle password scelte dall'utente; allo stesso tempo, le reti neurali possono essere addestrate senza alcuna conoscenza a priori o un'assunzione di tali proprietà e strutture. Questo concetto è in netto contrasto con gli approcci attuali studiati fino ad ora. Si pensi ad un attacco alle password basato sui modelli di Markov. Questa tecnica presuppone che tutte le caratteristiche rilevanti delle password possano essere definite in termini di n-grammi; o si considerano i rules attack, in grado di generare password solo in base a delle regole. Gli esempi potrebbero continuare ancora. Le password create utilizzando una rete neurale non sono limitate ad un solo aspetto particolare dello spazio delle password.

La principale rete utilizzata in un attacco di questo tipo è la *GAN*: **GAN**, una recente innovazione nel campo del ML. Una *Generative Adversarial Network* o *Rete Generativa Avversaria* è composta a sua volta da due reti neurali: una generativa, o rete generatrice G, e una discriminatoria, o rete discriminatrice D. Lo scopo della prima rete è quello di produrre nuovi dati, mentre la seconda rete apprende come distinguere i dati reali da quelli generati artificialmente. L'intenzione è quindi quella di creare un modello che riproduca più fedelmente possibile i dati di addestramento. Una volta raggiunto questo scopo, il discriminatore D, non riuscirà più a distinguere i dati reali da quelli generati. Questo stato viene raggiunto per mezzo dell'addestramento alternato tramite retropropagazione dell'errore: prima D viene addestrata in modo da massimizzare la probabilità di classificare in maniere esatta i campioni (sample) provenienti dai dati di addestramento e i campioni generati (i parametri del modello generativo rimangono costanti). Poi G viene allenata per massimizzare la probabilità di D di considerare i campioni prodotti dalla rete generativa (costanti, a sua volta, i parametri della rete discriminativa).

In termini matematici diciamo che la rete generatrice cerca di minimizzare la seguente funzione [11] mentre la rete discriminatoria cerca di massimizzarla:

$$E_x[log(D(x))] + E_z[log(1 - D(G(z))]$$

- dove D(x) è la funzione che rappresenta il discriminatore che prende in ingresso l'istanze x e restituisce la probabilità che tale istanza sia reale.
- dove E_x è il valore atteso su tutte le istanze di dati reali.
- dove G(z) è l'output del generatore quando viene dato in input istanze random.
- dove D(G(z)) stima del discriminatore su un istanza falsa. Restituisce la probabilità che un'istanza artificiale sia reale.
- dove E_Z è il valore atteso su tutte le istanze di dati artificiali.

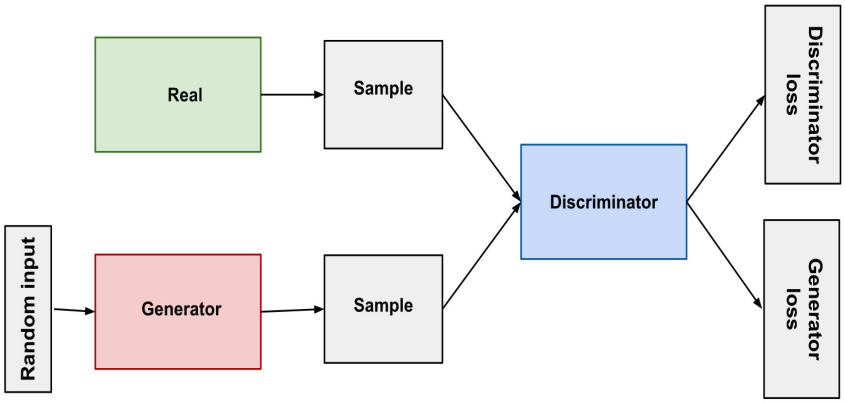


Figura 2.1: In figura l'intero sistema della rete GAN. Attraverso la retropropagazione il discriminatore fornisce poi un feedback al generatore che utilizzerà per aggiornare i suoi pesi [11].

Il principale tool nell'ambito del Password Cracking, basato su Deep Learning, è **PassGAN** [16]. PassGAN sfrutta il concetto appena descritto per generare nuove possibili password. Il discriminante D è addestrato utilizzando un elenco di password reali (campioni reali). Ad ogni iterazione, l'output di PassGAN (campioni falsi) si avvicinerà sempre di più alla distribuzione delle password del dataset originale, di conseguenza ad ogni iterazione aumenterà la probabilità che corrisponda alle password degli utenti reali.

2.2.4 PRINCE Attack

Una delle tecniche di Password Cracking più utilizzate è il **PRINCE Attack**. Questo tipo di attacco è basato sull'algoritmo PRINCE sviluppato da Jens Steube [29].

PRINCE sta per PRobability INfinite Chained Elements e utilizza un dizionario come base per le ipotesi di password e combina le parole del dizionario in vari modi. *L'"Infinito"*, nel suo nome, deriva dal fatto che funzionerà fino ad esaurire lo spazio delle chiavi, il che richiederà davvero molto tempo [30]. Internamente, PRINCE legge ogni parola del dizionario in una tabella di *elementi* (parole), ordinata per lunghezza. Gli elementi verranno poi combinati per creare *catene* di una certa lunghezza: ad esempio una catena di lunghezza quattro può essere creata da due elementi di lunghezza due (2 + 2) o quattro elementi di lunghezza uno (1 + 1 + 1 + 1), e così via. A seconda della lunghezza della catena e del numero di elementi nella catena, il numero di modi per materializzare la catena può essere diverso. Questo numero è indicato come lo *spazio delle chiavi* della catena. Una volta che PRINCE ha prodotto le password della lunghezza x, ordina le diverse catene (di lunghezza complessiva x) in maniera decrescente in base alla dimensione dello spazio delle chiavi e le esaurisce applicando diversi attacchi:

- Attacco a dizionario
- Ibrido
- Keyboard walks (attacco basato su pattern specifici nella tastiera, es:qwertyu)
- Brute Force + Markov

Ecco un esempio per comprendere meglio il concetto. Si supponga che PRINCE stia cercando di indovinare delle password di lunghezza 3: le catene che quindi creerà saranno in totale:

$$2^{lunghezza-1}$$

in particolare, 1 + 1 + 1, 2 + 1, 1 + 2 e 3. Si ipotizzi nuovamente che il dizionario fornito a PRINCE contenga 20 parole di lunghezza 3, 5 parole di lunghezza 2 e 3 parole di lunghezza 1. Quindi PRINCE esaurirebbe le catene nell'ordine indicato dalla tabella 2.4, iniziando con le parole di lunghezza 2 concatenate alle parole di lunghezza 1, e finendo con tutte le combinazioni di 3 parole di lunghezza 1.

Catena	Elemento	Spazio delle chiavi
1 + 2	3 * 5	15
2 + 1	5 * 3	15
3	20	20
1 + 1 + 1	3 * 3 * 3	27

Tabella 2.4: Un esempio di una tabella per password di lunghezza 3 usata da PRINCE.

I pro di un attacco simile sono:

- Semplicità d'uso
- Implementazione di diverse strategie di attacco
- Runtime pressochè infinito

Mentre i contro sono:

- Generazione di parole duplicate
- Creazione di parole di eccessiva lunghezza
- Il dizionario di input dovrebbe avere parole di tante lunghezze diverse

Capitolo 3

Analisi delle Tecnologie Utilizzate

Prima di procedere con la fase di testing è importante analizzare alcuni strumenti che si utilizzeranno ed effettuare una chiara analisi dello scopo di essi. Si esporrano i vari componenti e strumenti che faranno parte del progetto finale.

Le strutture che compongono il sistema di testing si dividono in due parti:

1. **Sezione Generativa**. In una prima fase le password andranno create da un dizionario di input. Le modalità con cui verranno generate varia da strumento a strumento e da dizionario a dizionario. Questa prima parte del progetto è definita di *Password Generation* o *Password Guessing* e comprende anche la fase di training del modello che il tool utilizza: il core di questo documento risiede in questi metodi. Le tecnologie impiegate sono **PCFG-Cracker** [41] (basato sui PCFG), **PassGAN** [18] (impiega il Deep Learning), **OMEN** [5] (utilizza i modelli di Markov), **CPG** (modifica e trasforma con regole definite le informazioni in input) e **PRINCEProcessor** (genera parole attraverso l'algoritmo PRINCE).

2. **Sezione Recuperativa**. Le password in chiaro generate andranno poi convertite tramite una funzione di hashing (definirò questa parola a breve) e confrontate con le password già criptate che si vogliono recuperare. Il *Password Cracking* si concentra in questa seconda fase, la sezione in cui realmente si rilevano le password sconosciute. Gli strumenti utilizzati in questa fase si fermano ad **Hashcat** [31].

Sono stati utilizzati poi strumenti non strettamente inerenti al password cracking, ma che sono stati utili in alcune fasi del percorso di testing e di cui parlerò brevemente: **phpMyAdmin**, **XAMPP**, **PACK** e **Google Colab**.

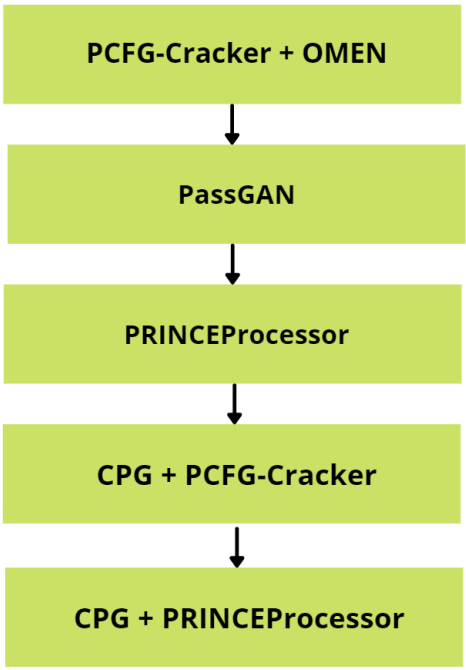


Figura 3.1: In grafica l'ordine di test degli strumenti di Password Guessing: tutti questi verranno in seguito affiancati ad Hashcat.

In figura 3.1 viene mostrato l'ordine con cui testerò i tool che sto per mostrare. Tutte le password che verranno generate saranno sempre poste in ingresso ad Hashcat. Si noti come alcuni tool vengano testati uniti ad altri: il PCFG-Cracker con OMEN, a causa delle implementazioni di essi, mentre il CPG con il PCFG-Cracker per via di una mia scelta tecnica.

3.1 Strumenti per il Password Guessing

3.1.1 PCFG-Cracker

PCFG-Cracker. è un tool sviluppato da Matt Weir durante il suo dottorato alla Florida University [36] e, come si intuisce dal titolo, pone le sue basi sul PCFG.

Il progetto utilizza il Machine Learning per identificare le abitudini di creazione di una password dell'utente target. Il tipo di Machine Learing utilizzato è il più classico della sua classe, non utilizza quindi Reti Neurali o altro tipo di Intelligenza Artificiale. Il modello PCFG che viene creato, viene addestrato su una lista di password sorelle (ovvero password

che l'utente target ha utilizzato in altre circostanze), password comuni o parole chiave del target in modo tale da seguire la stessa struttura e distribuzione. Durante la fase di training, l'algoritmo estrae tutte le informazioni possibili sulla struttura delle password come visto nel capitolo precedente. Il PCFG-Cracker, a differenza della versiona standard che utilizzava solo tre tipi di dato, sradicherà dalla password molte più informazioni. Tra i possibili simboli troviamo: lettere alfabetiche (A), cifre (D), lettere maiuscole (U), lettere minuscole (L), keyboard patterns (K), caratteri speciali (O), alcune informazioni personali come anni (Y) o email (E) e altro ancora. Tutte le informazioni necessarie saranno poi associate ad una probabilità e conservate in tabelle divise per tipologia di dato. Successivamente alla creazione di queste tabelle si passerà alla generazione delle password. La generazione avverà a partire dalle parole con probabilità maggiore fino a quelle meno probabili. Il processo di creazione andrà avanti finchè non sarà scoperta la password o l'attaccante si arrenderà. Altra parte da non sottovalutare di questo tool è la possibilità di riutilizzare queste tabelle/informazioni in altri strumenti di cracking come PRINCE e/o parti del set di regole possono essere incorporate direttamente in attacchi più tradizionali basati su dizionario.

PCFG-Cracker nasce dalle ceneri di *UnLock* il primissimo tool basato su PCFG che Matt Weir scrisse in Java e C++ [37]. Questo più recente è scritto interamente in Python e si divide in due sotto-programmi:

- **PCFG-Trainer**. Crea il modello basato su PCFG allenato sul dataset di input. Estrea quindi tutte le informazioni dalle password di training, generando delle regole che le identificano. Tra le opzioni più interessanti troviamo la possbilità di far riconoscere al trainer alcune informazioni sensibili (–save_sensitive), come la mail, e salvarle in memoria.

Figura 3.2: Schermata di Help del PCFG Trainer.

- **PCFG-Guesser**. Genera possibili password utilizzando il set di regole (*ruleset*), creato in precedenza, in ordine di probabilità decrescente. Le password candidate sono inviate su STDOUT e quindi molto facilmente si possono convogliare come input su un altro programma o su un file. Non manca anche la possibilità di salvare una sessione di cracking per usi futuri o per altri motivi in cui ci ritroviamo a bloccare la sessione in corso.

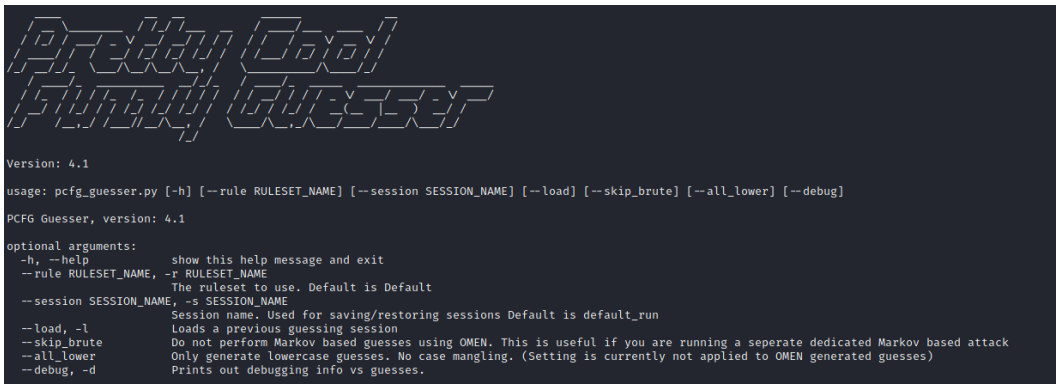


Figura 3.3: Schermata di Help del PCFG Guesser.

I requisiti per iniziare ad utilizzare il PCFG-Cracker sono Python3 e il modulo*Chardet* che può aiutare nella fase di Training per scoprire con quale codifica dei caratteri è stato scritto il train set (ASCII, UTF-8, ecc.): tale modulo fra l'altro viene installato spesso di default con *pip*. Il PCFG-Guesser è Single-Threaded CPU Buond quindi utilizza principalmente un Thread durante l'utilizzo massivo della CPU: infatti genera password abbastanza lentamente (i nostri test avranno durata di solo un'ora ma con alcuni tool, come questo, dovrebbero durare molto di più). Va però detto che esiste anche una versione scritta puramente in C [39], 20 volte più veloce della versione Python ma con grandi limitazioni nelle opzioni possibili (es: non si può salvare/ricominciare una sessione o non esiste un output status). E' inoltre estramemente raccomandato dedicargli almeno 16GB di memoria, soprattutto con sessione di cracking reali ed estremamente lunghe come 1/2 settimane. Durante questo tempo il PCFG creerà delle enormi tabelle di ruleset che durante l'esecuzione salverà in memoria. Il ruleset di default è però alquanto ridotto, è stato costruito su un set di 1 milione di password comuni. La pratica comune è poi passare in input, ad un tool di password cracking come Hashcat, le password attraverso una semplice pipe (|).

Concludo questa analisi del PCFG-Cracker spiegando una grande integrazione dalla versione 4.0: **OMEN**, un accurato guesser per il brute-force basato sulle catene di Markov: OMEN è un generatore di password molto interessante che sfrutta i modelli di Markov. Matt Weir ha riscritto in Python [38] l'originale implementazione in C di OMEN [5] per poi aggiungere questa logica ad entrambi i programmi .py. Come si nota dalle figure 3.2 e 3.3, sono state aggiunte delle opzioni per personalizzare un attacco utilizzando anche l'idea di OMEN. Nel *Trainer* è possibile modificare alcune funzionalità avanzate per avere una fase di addestramento del tipo Markov Model Based. Queste nuove informazioni verrano salvate in una cartella a parte ("Omen"). Nel guesser invece viene fornita la possibilità di disabilitare la OMEN password generation attraverso *–skip_ brute*. Questa scelta viene data in quanto una persona potrebbe volere portare un attacco brute-force separato.

3.1.2 OMEN

OMEN è un cracker di password basato su un modello di Markov migliorato rispetto al modello utilizzato in JtR (John-the-Ripper) [21]. L'algoritmo su cui si basa il tool *Ordered Markov ENumerator* enumera le password con probabilità decrescente. In pratica questo algoritmo discretizza tutte le probabilità in un certo numero di insiemi: il numero che viene dato all'insieme rispecchia la probabilità delle password che verranno inserite in esso. Una volta identificati gli insiemi vengono loro associate le corrette password; si itera poi su questi set in ordine decrescente di probabilità ed infine vengono trasmessi in output.

L'algoritmo comincia con il calcolo del logaritmo di tutte le probabilità degli n-grammi e discretizza queste probabilità in *livelli* che denotiamo con η. I livelli sono discretizzati tendenzialmente in 10/11, da 0 a -9/-10. Il numero di livelli può variare: aumentando i livelli si aumenta l'accuratezza ma si aumenta anche il runtime. Di seguito la formula che descrive questo:

$$\eta_i = lvl_i = round(log(c_1 \cdot prob_i + c_2))$$

dove c_1 e c_2 sono parametri scelti dall'algoritmo in modo tale che gli n-grammi più frequenti abbiano un livello uguale a 0 e che agli n-grammi, che non sono comparsi nell'addestramento, sia assegnata una probabilità ancora più piccola.

L'algoritmo OMEN(η, l) prende in ingresso un livello η e una lunghezza l e avanza come segue:

1. Istanzia tutti i vettori $a = (a_2,..,a_l)$ di lunghezza pari a l - 1 affinchè ogni istanza a_i sia un intero compreso tra 0 e -9 e la somma di questi elementi sia η. Si noti che la lunghezza delle istanze di a deve essere impostata a l - 1 in quanto avremo sempre una probabilità iniziale da cui partire.

2. Per ciascuno di questi a_i , seleziona tutti i 2-grammi x_1x_2 le cui probabilità corrispondono al livello a_2. Per questi 2-grammi, si itera su tutti gli x_3 in modo tale che il 3-grammo $x_1x_2x_3$ abbia livello a_3. Successivamente, per ciascuno di questi 3-grammi, cicla su tutti gli x_4 al fine di avere il 3-grammo $x_2x_3x_4$ abbia il livello a_4, e così via, fino a raggiungere la lunghezza desiderata. Alla fine, questo processo restituisce un insieme di password candidate di lunghezza l e livello η.

Di seguito l'algoritmo in maniera formale:

Algorithm 1 Enumerazione passwords per livello η e lunghezza l. Esempio con $l = 4$.

function OMENPWD(η,l)
for each vector $(a_i)_{2\leq i\leq l}$ with $\sum_i a_i = \eta$
for each $x_1x_2 \in \Sigma^2$ with $L(x_1x_2) = a_2$
for each $x_3 \in \Sigma$ with $L(x_3|x_1x_2) = a_3$
for each $x_4 \in \Sigma$ with $L(x_4|x_2x_3) = a_4$
(a) output $x_1x_2x_3x_4$

Con $L(xz)$ e $L(y|xz)$ indico i livelli per le probabilità iniziali e condizionate.

Per comprendere al meglio l'idea di base di OMEN svolgo un esempio che si può ritrovare anche nel lavoro di Castelluccia et al. [6] in maniera meno discorsiva. Si consideri una password di lunghezza pari a $l = 3$ che costruirò utilizzando un alfabeto $\Sigma = \{a, b\}$, con probabilità di partenza (ovvero i "livelli", dopo aver discretizzato le probabilità) pari a:

$$L(aa) = 0, L(ab) = -1,$$

$$L(ba) = -1, L(bb) = 0$$

E con probabilità condizionate (transizioni) pari a:

$$L(a|aa) = -1, L(b|aa) = -1,$$

$$L(a|ab) = 0, L(b|ab) = -2,$$

$$L(a|ba) = -1, L(b|ba) = -1,$$

$$L(a|bb) = 0, L(b|bb) = -2$$

Ricordo la regola generale: dato un livello η si dovrebbe ottenere una password di lunghezza l tale che la somma delle probabilità/livelli delle "parti" che compongono la password sia pari a η.

$\eta =$ **0** Partendo dal livello $\eta = 0$, si deve ottenere una o più password tale che sia la probabilità iniziale sia quella condizionata siano pari a 0: in altre parole un vettore di elementi le cui probabilità sono alla pari di (0,0). Si noti che le stringhe che hanno probabilità di partenza pari a 0 sono *aa* e *bb*. Ora che si ha la stringa iniziale si deve cercare l'ultima parte di password tale che la somma dei livelli rimanga 0. La password *bba* soddisfa questi requisiti: infatti $L(a|bb) = 0$ ammette la possibilità di generare la stringa *bba* inalterando la somma dei livelli. Non si trovano però corrispondenze per la stringa *aa*. Infatti nessuna delle probabilità condizionate contenente *aa* come evento di partenza, ha livello 0.

$\eta =$ **-1** Procedo come poc'anzi. Il livello η = -1 manderà in output due vettori. Il primo (-1,0), con al suo interno *aba* (il prefisso *ba* non ha transizioni di livello 0), ed il secondo vettore (0,-1) che contiene *aaa* e *aab*.

$\eta =$ **-2** Il livello η = -2 restituirà tre vettori: (-2,0), senza niente al suo interno (nessuna delle probabilità iniziali vale -2), (-1,-1) con *baa* e *bab* e infine (0,-2) con *bba*.

altri η. Si procede così per i restanti livelli.

Il software vero e proprio è composto da due programmi:

- **createNG**. La generazione delle password con OMEN comincia con il calcolo delle probabilità degli n-grammi. Il modulo createNG svolgerà i suoi calcoli utilizzando le probabilità e basandosi sulle impostazioni predefinite (10 livelli, ecc.). Il modello sarà addestrato su un documento di testo contenente le password di training, che saranno disposte una per riga. A fine esecuzione il programma genererà quattro file aggiuntivi contenenti informazioni sugli n-grammi e sulla lunghezza della password. Questi file hanno tutti estensione *.level* :
  - IP.level (Initial Probability): Salva le probabilità (della prima parte di password in pratica) del primo (n-1)-grammo per ogni password.

  - CP.level (Conditional Probability): Memorizza le probabilità condizionate.
  - EP.level (End Probability): Le probabilità (dell'utlima parte di password in pratica) dell'ultimo (n-1)-grammo di ogni password sono salvate in questo file.
  - LN.level (Length): Memorizza le probabilità per ogni lunghezza possibile delle password in input.

  Queste probabilità saranno espresse con un valore compreso tra 0 (più probabile) e 10 (meno probabile).

- **enumNG**. Una volta che i file necessari alla generazione delle password sono stati creati allora si passerà all'effettiva creazione con il modulo enumNG. Queste password verrano visualizzate a video, quindi facilmente salvabili su file o altro.

Il password cracker OMEN verrà utilizzato nei test insieme al PCFG-Cracker. Sviluppi e test futuri potranno analizzare singolarmente questo tool. A tal proposito bisogna puntualizzare che la lunghezza degli n-grammi che l'OMEN implementato nel PCFG (che chiameremo $PCFG_{OMEN}$) utilizza, è quattro (questo di default ma è poi personalizzabile aumentando/diminuendo il valore e di conseguenza il consumo di risorse). I file .level saranno tutti settati su lunghezza 4. Oltre ai soliti file, il modulo $createNG_{PCFG}$ ne creerà altri utili alla sua integrazione con il PCFG, come *pcfg_omen_prob.txt* che contiene la probabilità di una password di una certa lunghezza trasportato nelle probabilità del PCFG (quindi dal range 0-10 si passa al valore in % classico).

3.1.3 PassGAN

Il processo di sviluppo di apparati adatti al Password Cracking sta ricevendo una grossa mano dall'intelligenza artificiale. I calcolatori sono in grado ora di riconoscere molti più pattern e schemi in una password. Tra i primi e più importanti strumenti incentrati sull'IA applicata al password attack troviamo **PassGAN**. Il problema della comprensione di una struttura della password viene ora stravolto da questo nuovo approccio Deep Learning Based. Invece di utilizzare tecniche probabilistiche o pattern prestabiliti, il core di PassGAN fa uso di *Generative Adversarial Network o GAN* una particolare rete neurale che gli permetterà di generare password possibili di altissima qualità. Con le rules classiche di generazione delle password il numero di termini univoci che otterremo sarà prestabilito dal numero di rules stesse e dalla dimensione del set di dati in input. PassGAN può invece produrre un numero (pressochè) illimitato di password.

L'implementazione del modello di PassGAN utilizza una specifica rete neurale chiamata WGAN (descritta da Gulrajani et al. [14] e realizzata sempre da esso, scaricabile da GitHub [13]). Gli *iper-parametri* sono quei valori settati tendenzialmente prima della fase di learning che migliorano la fase stessa. I settaggi degli iperparametri sono rimasti quasi tutti invariati rispetto al paper iniziale [16] e sono:

- **Batch Size**. Il training set potrebbe essere troppo grande per essere elaborato tutto in una volta. Quindi si può suddividere in sottogruppi uniformi, chiamati *batch*. Il numero di esempi contenuti in ogni batch è chiamato *batch size*. Nel nostro caso il batch size sarà il numero di password che dal training set sarà inviato attraverso la GAN per ogni step dell'ottimizzatore. Il batch size è fissato a 64.

- **Numero di iterazioni**. Il *numero di iterazioni* corrisponde al numero di batch necessari a completare un *epoch*, ovvero quando il modello è stato allenato sull'intero training set. Se ad esempio avessimo un training set di 200 istanze e lo dividessimo in batch da 50, avremmo allora che una epoch è formata da 4 iterazioni. Nel nostro caso specifico il numero di iterazioni indica quante volte il GAN invia i dati e riceve un feedback. Per ogni iterazione di GAN, il generatore compie un'iterazione mentre il discriminatore ne farà una o più. Inizialmente il modello veniva allenato per 199 mila iterazioni ma per scarsità di risorse ne compierò 35 mila.

- **Numero di iterazione del discriminante rispetto alle iterazione del generatore**. Come detto poco fa ad ogni iterazione del generatore il discriminante può fare molteplici iterazioni. Questo numero quindi sta ad indicare quante ripetizioni il discriminante fa per ognuna del GAN. Per default il valore è 10.

- **Dimensione del modello**. Una rete neurale può essere immaginata come composta di diversi "layer" (strati) di nodi, ciascuno dei quali è collegato ai nodi del layer successivo. Questo valore rappresenta il numero di dimensioni per ciascun layer. Nel modello del PassGAN si avranno 5 layer (che chiameremo Residual Block) sia per la rete generativa sia per quella discriminatoria. Ciascun layer avrà 128 dimensioni ovvero 128 features/parametri che il nostro modello cercherà di ottimizzare al meglio.

- **Gradient Penalty Coefficent o λ**. E' un iperparametro che serve per regolarizzare la parte di training, quindi renderla più stabile. In particolare specifica la penalty (riduzione/aggiustamento) che verrà applicata alla norma del gradiente della rete discriminatoria rispetto al suo input. Più l'iperparametro è alto più tende ad essere stabile, ovviamente entro certi limiti. Nel paper di PassGAN [16], è impostato a 10 questo valore.

- **Lunghezza massima delle sequenze di output**. Indica la lunghezza massima che le stringhe in uscita da PassGAN avranno. Questo valore è impostato a 10 e sarà uno dei valori più critici e su cui ci si dovrà soffermare durante la fase di testing.

- **Dimensione del vettore random/noise**. Come si può notare dalla figura 2.1 e dalla figura 3.4 dove viene mostrata la struttura del PassGAN, la rete che genera password artificiali prende in ingresso un vettore di numeri random ("rumore"). La quantità di numeri random che verranno forniti in ingresso dipenderà da questo iperparametro, fissata a 128. Il "rumore" segue una distribuzione normale (ovvero una distribuzione di numeri che segue una curva gaussiana o normale) e tali dati saranno di tipo *float*.

- **Numero massimo di istanze in fase di training**. Il parametro indica il numero di oggetti che verranno presi in considerazione durante la fase di training. Qui l'intero dataset di password in input sarà usato come train-set.

- **Iperparametri dell'Ottimizzatore Adam**. Il deep learning è un processo iterativo con molteplici parametri da settare propriamente. Il nostro obiettivo è generare un modello capace di generalizzare i dati (ovvero predire correttamente istanze future) con un'alta accuratezza, cioè che abbia la minima differenza possibile tra i valori predetti e quelli reali. La differenza tra valori predetti e reali viene detta *loss function* e sarà misurata su ogni osservazione. Si deve minimizzare la loss function individuando i valori ottimizzati per ogni parametro del modello. Ecco che entrano in gioco gli *algoritmi di ottimizzazione*. Attraverso iterazioni multiple questi algoritmi consentono l'individuazione dei valori dei parametri migliori che minimizzano la loss function. *L'ottimizzatore Adam* è uno dei più utilizzati e, per funzionare al meglio, deve aver impostato alcuni iperparametri:

  - **Learning rate**. Il Learning rate o velocità di apprendimento o step size, è un iperparametro di ottimizzazione per un algoritmo che determina la dimensione del passo ad ogni iterazione mentre si sposta verso una funzione minima di perdita. Il valore utilizzato rimane invariato rispetto a quello di default di 10^{-4}.
  - **Coefficente β_1 e β_2**. Valori utili al procedimento di ottimizzazione interno dell'Adam Optimizer. Impostati rispettivamente a 0.5 e 0.9.

La figura 3.4 indica l'architettura di PassGAN. Senza entrare troppo nel dettaglio si può notare come entrambe le reti neurali abbiamo 5 layer, che aiutano a ridurre l'errore di training del modello (più layer, minore errore), connesse a loro volta con un *layer convoluzionale 1-dimensionale*.

Brevemente, nell'input delle reti neurali bisogna spesso cercare di riconoscere dei pattern che possano essere presenti in diverse porzioni dell'input. Inserendo all'inizio della rete uno strato *Conv1D* e impostando sia il numero sia la lunghezza dei pattern da cercare, si ottiene in output un array 2D che indica quali pattern sono stati individuati e in che punto dell'input.

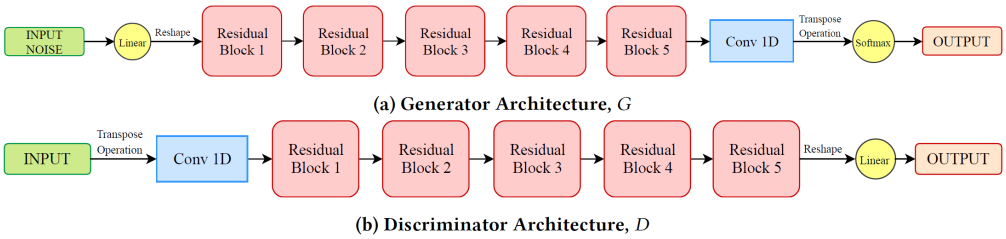


Figura 3.4: Architettura di PassGAN [16].

3.1.4 CPG: Cyber Password Guesser

Se la password si basa sui dati dell'utente (ad esempio una combinazione di nome, cognome, data di nascita, nomi di bambini, numero di telefono), allora tale password può essere considerata debole. Una password debole può anche essere facilmente indovinata da qualcuno che profila l'utente, quindi arrivando a dati personali e, ad esempio, utilizzandoli con parole o password comuni. Una tipologia di attacco che combina vocaboli con certe regole prestabilite in partenza è, come già visto, il rules based attack. Queste rules hanno però il difetto di essere complicate da definire, soprattutto per un inesperto nel campo del password cracking. Le persone si affidano quindi spesso a strumentazioni che applicano modifiche pre-identificate alle password che gli vengono passate in input. Questo semplicissimo concetto è stato poi esteso ai dati personali per generare password.

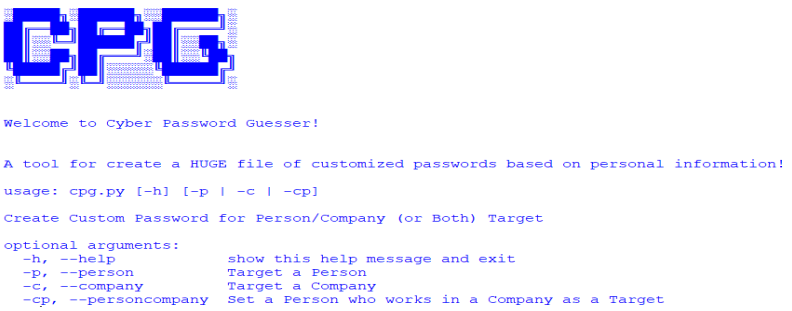
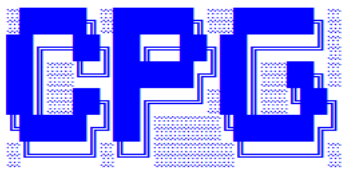


Figura 3.5: Schermata di Help di CPG.

Automatizzare il processo di generazione password strettamente basate su informazioni personali era una sfida già raccolta da altri sviluppatori. Citiamo tra i più noti CUPP [2]. La necessità di intraprendere lo sviluppo di un progetto simile a CUPP, ma diverso da esso, la si può ritrovare nella scarsa quantità di password generate (e di conseguenza le poche modifiche effettuate sulle informazioni personali) oltre che sulla scelta obbligata di prendere una persona come target e non altro (come un'azienda). Questo è il motivo per cui ho creato **CPG** o **Cyber Password Guesser**: un generatore di password da semplici informazioni personali come nome, cognome, data di nascita, ecc. CPG è inoltre in grado di profilare tipologie di target diversi (persona, azienda e persona-in-azienda) tramite richieste da tastiera differenziate in base al tipo. L'utente dovrà preoccuparsi solo di inserire la tipologia di target e, quando richiesto dal software, le informazioni personali in suo possesso: CPG creerà un enorme file di testo dove saranno contenute le possibili password personali.



```
Welcome to Cyber Password Guesser!

A tool for create a HUGE file of customized passwords based on personal information!

Insert the information about the Person Target

If you don't know the info, hit enter

Enter first name: Angelo

Enter surname: Parrinello

Enter the nickname:
```

Figura 3.6: Screenshot del CPG in azione.

Molti strumenti più innovativi, visti in precedenza, esigono un ingente numero di password delle più disparate lunghezze. Tool come CUPP non permettono di creare tante password e di conseguenza, nel caso in cui si avessero poche password relative alla vittima, i software più recenti potrebbero essere meno efficaci. Un tool come CPG *potrebbe* tornare utile in questi casi (anche se non è propriamente così) ma non solo: è altrettanto perfetto per un attacco a dizionario standard.

Il funzionamento di CPG è di per sè alquanto semplice. Quando viene lanciato lo script bisognerà indicargli quale categoria di target si sta attaccando tra persona, azienda e persona-in-azienda. Una volta definito ciò, si passerà alla profilazione vera e propria del target. Verrà chiesto di inserire diverse informazioni a seconda dell'obbiettivo (nel caso uno non sappia rispondere alla domanda che gli viene posta basta premere *Invio*). Se ad esempio volessi provare ad attaccare un database aziendale, CPG chiederebbe come informazione l'anno di fondazione della compagnia (dato che per una persona-target non domanderebbe). Questa piccola ma importante differenza permette di generare password più specifiche per un tipo di target rispetto ad un altro. Sia durante il mio tirocinio curricolare sia quello in preparazione alla prova finale mi è stato insegnato, infatti, come le aziende tendano a salvare password d'accesso ai propri sistemi con pattern molto semplici (es: nomeazienda_annodifondazione). Questa regola è una debolezza che un generatore di password come il mio può ampiamente sfruttare. Le informazioni inserite da tastiera vengono poi controllate affinchè abbiano la giusta configurazione (es: l'anno di nascita deve avere esattamente quattro numeri) e, a seconda dei casi, tutti i caratteri presi in input per una certa domanda verranno trasformati in minuscolo. Esempio: i caratteri del nome e del cognome verranno portati tutti in minuscolo. Mentre il nickname del target rimarrà invariato perchè alcuni dati non vanno modificati: si perderebbero informazioni intrinseche in essi (un nome ha un contenuto generico, non identifica un target mentre un nickname lo identifica maggiormente). Le informazioni personali "corrette" verranno poi salvate in un'unica lista per comodità. La generazione delle password prosegue poi in maniera lineare, dalle password più facili a quelle maggiormente complicate, rimuovendo alla fine le *password duplicate*. Come è stato chiarito nei capitoli precedenti le password duplicate sono fondamentali in diverse procedure di password cracking. Le password duplicate dovrebbero essere però password "reali" e non artificiali, perchè si falserebbe il modello che si sta allenando, inserendo informazioni sulla distribuzione non vere. Inoltre nel caso in cui non si rimuovessero le password, i file generati da CPG sarebbero, in certi circostanze, davvero enormi ($\sim$ 30GB). Per questi motivi è stato scelto di non inserire duplicati nel dizionario di output di CPG.

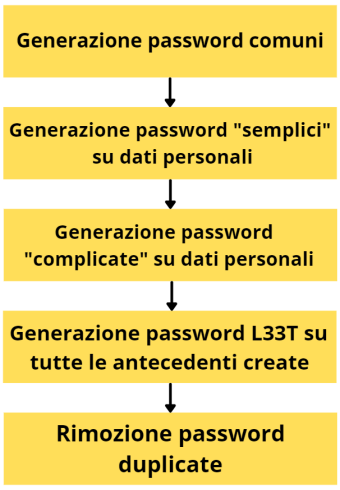


Figura 3.7: Protocollo creazione dizionario personalizzato CPG.

CPG è in grado di generare un enorme quantitativo di password mediante l'uso di funzioni basate sulle *combinazioni* e sulle *permutazioni*. I concetti della combinazione e della permutazione sono particolarmente usati nella fase di creazione password: le password nella loro interezza sono spesso combinazioni di due o più parole. Le informazioni personali vengono da CPG combinate con altre informazioni, simboli (come _ o #) o con altre password comuni (ad esempio qwerty123). CPG fa avvenire questi accorpamenti a fine di una parola, a metà o in un punto particolare di essa; le permutazioni invece avvengono o su una parola (generando quindi tutte le permutazioni possibili di essa es: password → apssword, pawssord, ecc.) oppure di due in due su una lista di parole, per creare in seguito combinazioni di esse. Si pensi alla lista di parole [nome, cognome, anno di nascita, mese di nascita]: una funzione usata da CPG creerà tutte le permutazioni possibili tra di loro unendole in tuple come (nome, cognome) o (anno di nascita, nome). A queste tuple saranno poi applicate modifiche di diverso tipo come la semplice combinazione di esse o l'unione delle due con al termine un "_". Queste modifiche basate su concetti del calcolo combinatorio avvengono per lo più nel terzo step della figura 3.7 e accadono non solo sulle parole personali singole ma anche su trasformazioni di esse.

L'alfabeto *L33t* (o Leet) è un altro protagonista del CPG. Questo alfabeto è una forma di inglese identificata dall'uso di caratteri non alfabetici (es: e → 3) al posto delle normali lettere, trasformate così per somiglianza o fonetica. Questa particolare variazione del lessico di una parola avviene molto spesso nelle password. Per CPG ho sviluppato una funzione ba-

sata sulle più importanti trasformazioni leet (vedi figura 3.8) in grado di generare ogni possibile modifica leet alla parola presa in considerazione in quel momento, come mostrato in figura 3.11. Quindi al quarto step della figura 3.6 tutte le password che si sono generate fino a quel momento saranno date in ingresso alla nostra funzione leet che, per ognuna di esse, applicherà *tutte* le trasformazioni leet possibili. Ad esempio da *password* si genererà *p4ssword*, *p45sword*, *pas5w0rd* e così via. Il quarto passaggio è quello più durevole fra i cinque, soprattutto nel caso in cui siano state inserite tutte le informazioni personali e queste avessero lunghezza abbondante. Si noti come la funzione in figura 3.11 utilizza lo statement *yield*. L'uso dell'istruzione yield nella definizione di una funzione è sufficiente a creare una funzione generatore invece che una normale funzione. I generatori sono utili in quanto producono un elemento alla volta ad ogni chiamata della funzione. Questo permette quindi di risparmiare spazio in memoria: infatti la funzione *leet_psw_yiled(password)* in diversi casi generava liste talmente lunghe da non essere contenute interamente in memoria causando così un *MemoryError*. Usando la keyword yield la funzione "ritornava" un oggetto per volta a chiamata così da salvarsi solo un elemento per volta. Ovviamente gli elementi che vengono restituiti dalla chiamata alla funzione non saranno salvati in memoria ma sul disco fisso altrimenti saremmo di nuovo al punto di partenza tornando a generare un altro MemoryError.

```
leet_chars = {"a":"4",
              "b":"8",
              "e":"3",
              "g":"6",
              "i":"1",
              "o":"0",
              "s":"5",
              "t":"7",
              "z":"2"
    }
```

Figura 3.8: Dizionario leet utilizzato in CPG.

Il software CPG è composto da due parti:

1. **cpg.py**. Programma Python principale che crea le classi del target, profila le persona e richiama le funzioni definite in functions.py nell'ordine corretto per creare il dizionario finale.

2. **functions.py**. Modulo creato per modificare in determinate maniere le password. Molte funzioni definite da me in questa libreria sono perfettamente utilizzabili per altri progetti simili al mio. Contiene inoltre il dizionario leet e le liste di alcune parole e simboli comuni. Utilizza massivamente il modulo *itertools* presente nella

libreria standard di Python contenente funzioni combinatorie [15]. Di seguito alcuni snippet delle funzioni del modulo da me creato.

```
#Creates a permutations (of two in two) from a word set.
def words_permutations(words):
    """
    Creates permutations (of two in two) from a word set.
    """
    repeat = 2
    single_words = set(words)
    all_perm_set = itertools.permutations(single_words, r=repeat)

    return all_perm_set
```

Figura 3.9: Funzione che crea tutte le possibili permutazioni di due in due dato un insieme di password.

```
#Removes all duplicates from an input txt file
def unique_words_txt(input_file, output_file):
    """
    Create a new file without duplicates from an input file.
    """
    seen = set()
    with open(input_file, 'r') as fin, open(output_file, 'a') as fout:
        for line in fin:
            h = hash(line)
            if h not in seen:
                    fout.write(line)
                    seen.add(h)
```

Figura 3.10: Funzione che elimina tutti le righe duplicate da un file in input e le riscrive singolarmente su un file di output.

```python
#Create a generator of list of all possible leet password from an input word
def leet_psw_yield(passwords):
    """
    Create a generator of a list of words to which it was appliedall possible
    leet trasformation.
    """
    for psw in passwords:
        total = []
        temp = []
        for dig in psw.lower():
            lettini = leet_chars.get(dig, dig)
            for letti in lettini:
                if letti == dig:
                    temp.append((dig,))
                else:
                    temp.append((dig, letti))
        total = ["".join(t) for t in itertools.product(*temp)]

        yield total
```

Figura 3.11: Funzione che genera tutte le possibili trasformazioni leet per ogni parola in una lista di parole.

```python
#Create a permutations of ALL character from a each word in an input word set
#PAY ATTENTION! From a simple word of legnth 8 you'll receive 40 320 and so on...
def character_permutations(words):
    """
    Creates all permutations of each word from a word set. Pay attention at input string length.
    """
    single_words = set(words)
    char_perm=[]
    for psx in single_words:
        temp_list = itertools.permutations(list(psx))
        for p in temp_list:
            convert_string = convert_list_to_string(p)
            char_perm.append(convert_string)

    return char_perm
```

Figura 3.12: Funzione che ritorna tutte le permutazioni possibili di ogni singola parola da una lista di parole.

Infine dopo aver generato le password ed averle salvate su disco, con una funzione apposta (figura 3.10) elimina i duplicati, salva le parole singole su un ulteriore file e infine elimina il primo file. La funzione legge riga per riga il file contenente le password duplicate: ad ognuna viene applicata una funzione hash (per risparmiare spazio) e salvata in un set. Se una password è gia contenuta nel set allora verrà scartata essendo duplicata; nel caso non ci fosse allora viene salvata su un secondo file che non contiene duplicati.

3.1.5 PRINCEProcessor

Il **PRINCEProcessor** è l'ultimo strumento che analizzo tra quelli usati per la fase di *Password Guessing*. Nella pratica l'algoritmo PRINCE è stato creato specificatamente per questo tool, quindi con la precedente discussione riguardo il *Prince Attack* ho già detto molto anche riguardo questo tool.

Il PRINCEProcessor è un generatore di password che utilizza l'algoritmo PRINCE. PRINCE è l'acronimo di PRobability INfinite Chained Elements, cioè gli elementi fondamentali dell'algoritmo. PRINCEProcessor (PP per brevità) è stato pensato nella sua essenza come un attacco combinatorio avanzato. Al posto di ricevere in input due diversi elenchi di parole, come l'attacco combinatorio tradizionale vorrebbe, e poi creare tutte le possibili combinazioni di due parole, PP riceve solo un dizionario di parole in input e costruisce "chain" (catene) di parole combinate. Le "chain" che si formano si basano su concatenazioni da 1 a N parole dall'elenco di input. Il numero di parole concatenate massimo/minimo può variare modificando delle opzioni dello script. Se si stessero generando password di quattro lettere, le genererà seguendo l'iter seguente:

Parola di 4 lettere
Parola di 2 lettere + parola di 2 lettere
Parola di 1 lettera + parola di 3 lettere
Parola di 3 lettere + parola di 1 lettera
Parola di 1 lettera + parola di 1 lettera + parola di 2 lettere
Parola di 1 lettera + parola di 2 lettere + parola di 1 lettera
Parola di 2 lettere + parola di 1 lettera + parola di 1 lettera
Parola di 1 lettera + parola di 1 lettera + parola di 1 lettera + parola di 1 lettera

Il PP è stato progettato per attaccare hash lenti ma si comporta egregiamente anche con hash più rapidi (come MD5). Ricordiamo infine che uno dei punti di forza di PRINCE (e di conseguenza anche di PP) è il runtime pressochè infinito, quindi questo strumento andrebbe lasciato lavorare per tantissime ore per ottenere risultati soddisfacenti. Ricerche hanno dimostrato come PP sia capace di recuperare più del 60% di password lavorando per 24 ore [26].

3.2 Strumenti per il Password Cracking

3.2.1 Hashcat

La tecnica classica per conservare in maniera sicura una password in un Database consiste nell'applicare una funzione di **Hash** su di essa. Questa tecnica crittografica permette di convertire una password "in chiaro", in un lungo elenco di caratteri casuali praticamente inutilizzabile. Queste funzioni hash devono essere facili da calcolare ma estremamente difficili da invertire (*proprietà one-way*) e la firma hash (ovvero la trasformazione che la funzione hash applica sull'elemento in considerazione), per una chiave dovrà essere univoca: due elementi diversi non potranno avere la stessa immagine per un dato algoritmo di hashing (*proprietà claw-free*) [20]. Le famiglie delle funzioni di hashing sono molteplici: le più importanti e celebri sono *MD5* (Message Digest Version 5), *SHA* (Secure Hash Algorithm) e *RIPEMD-160* (Race Integrity Primitive Evaluation Message Digest).
Anche se si entrasse in possesso di password criptate sarebbe impossibile invertire la funzione hash che "protegge" la parola (proprietà one-way). L'unica maniera per identificare la password in chiaro è generarne di nuove, applicare ad esse la stessa funzione hash della password target e confrontare le due immagini hash. Gli strumenti in grado di eseguire questa attività in maniera avanzata sono molteplici, *John-The-Ripper*, *CainAbel* e soprattutto *Hashcat* ovvero quello da noi utilizzato. **Hashcat** è il tool di recupero password più veloce e avanzato al mondo, in grado di supportare 7 modalità di attacco per oltre 300 algoritmi di Hashing ottimizzati.

Figura 3.13: Logo di Hashcat.

Hashcat permette di preparare attività di Password Recovery (recupero password) altamente flessibili e personalizzabili. Allo stato attuale supporta CPU, GPU (e altro hardware) sui principali sistemi operativi quali Windows, Linux e macOS. Hashcat fornisce anche la strumentazione per il *cracking distribuito* delle password, ovvero crackare password con più sistemi contemporaneamente. Uscito nel 2015, Hashcat continua ad essere un tool estremamente prezioso per chiunque operi nel campo della sicurezza informatica. Allo stato attuale la repo su Github conta più di 80 collaboratori e offre una community molto forte grazie anche al forum molto florido.

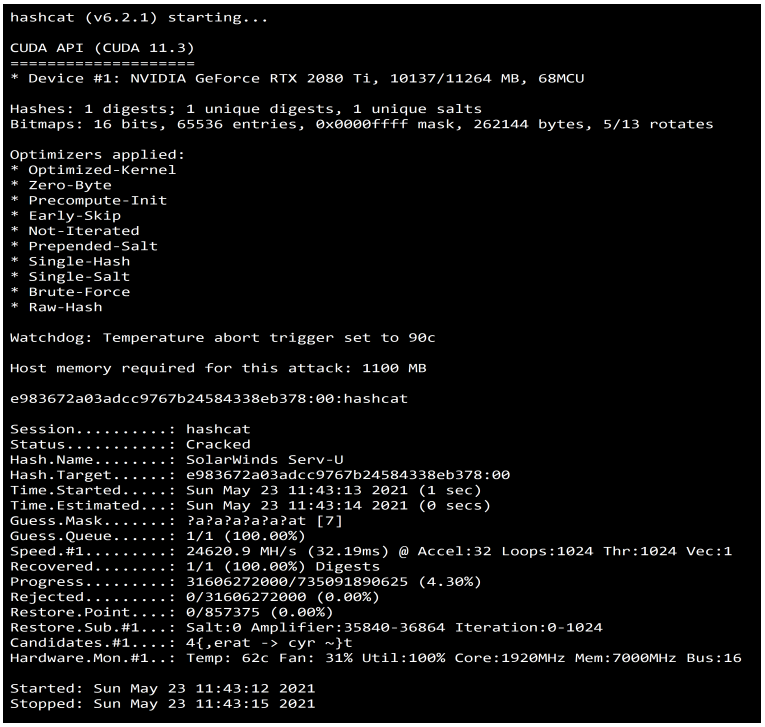


Figura 3.14: Screenshot di un esecuzione completata di Hashcat [31].

3.3 Strumenti aggiuntivi

3.3.1 phpMyAdmin

Il software per basi di dati **phpMyAdmin** è un'applicazione web gratuita scritta in PHP, destinata a gestire dei database di tipo MySQL o MariaDB attraverso qualsiasi browser. Le operazioni di uso frequente (gestione di database, tabelle, colonne, relazioni, indici, utenti, permessi, ecc.) possono essere eseguite tramite l'interfaccia utente, pur avendo la possibilità di eseguire direttamente qualsiasi istruzione SQL [4]. L'app può essere utilizzata sia dall'amministratore sia dall'utente generico. Verrà utilizzata in fase di manipolazione e modifica di dati.

Figura 3.15: Logo di phpMyAdmin [4]

3.3.2 XAMPP

XAMPP è una distribuzione Apache gratuita per creare un server web multipiattaforma tramite tecnologie open source. XAMPP è composto da Apache HTTP Server, MySQL (o MariaDB), PHP e Perl [23]. La X iniziale fa riferimento al suo essere multipiattaforma (al contrario di LAMP, WAMP e MAMP): è infatti compatibile con i sistemi operativi Windows, Linux, Mac e Solaris. XAMPP viene utilizzato per creare un web hosting locale sul proprio computer grazie ad una distribuzione facile da installare; è particolarmente usato per scopi di testing nella programmazione web.

Figura 3.16: Logo di XAMPP [23]

3.3.3 PACK: Password Analysis and Cracking Kit

PACK (Password Analysis and Cracking Toolkit) [19] è un insieme di utility ideate per analizzare elenchi di password. L'obbiettivo è migliorare il password cracking attraverso il rilevamento di maschere, regole, set di caratteri e altre caratteristiche delle password. PACK è in grado di creare file utilizzabili da *Hashcat*. Le features di questo programma sono molteplici ma userò la funzione base dove, dato un dizionario di input, otterremo lunghezza, set di caratteri e altre caratteristiche più comuni delle password.

Figura 3.17: Screenshot di un risultato di PACK [19]

3.3.4 Google Colab

I modelli di Machine Learning e Deep Learning richiedono una potenza di calcolo davvero importante. Durante le prime fasi di lavoro può essere sufficiente la propria macchina, ma con il crescere delle dimensioni del dataset le possibilità in termini di tempo, spazio e potenza di calcolo possono aumentare a dismisura. Per ovviare a questa necessità il gigante di Montain View ha creato **Colab**. Google Colab [10] è uno strumento gratuito che consente di scrivere codice Python direttamente dal proprio browser. Le macchine virtuali messe a disposizione in Google Colab ospitano un ambiente configurato nel quale sono presenti numerose librerie Python e si può usufruire di GPU e TPU per dare boost computazionali importanti ai nostri lavori. Per esempio le GPU vengono spesso utilizzate in progetti di reti neurali, come PassGAN nel mio caso. In Google Colab le risorse che vengono messe a disposizione agli utenti sono limitate, e variano a seconda delle fluttuazioni nella domanda e del consumo nei passati giorni. Un giorno si potrebbe venire assegnati ad una macchina virtuale con 32 GB di RAM e il giorno successivo ad una macchina virtuale con 12 GB di RAM. Ma non solo: se si dovessero usare le periferiche Google per più giorni consecutivi allora le sessioni in Colab dureranno sempre meno. Una soluzione potrebbe essere quella di creare almeno 3 account diversi e, a turno, cambiarli se la sessione termina. Inoltre Colab è stato creato per l'uso continuativo quindi ciclicamente (circa ogni 2/3 ore) apparirà un Captcha, che andrà cliccato: pena la distruzione della sessione in corso. Se si ha bisogno di maggior stabilità, di macchine più performanti o di accedere a GPU e TPU più potenti, è possibile utilizzare la licenza a pagamento, ad oggi disponibile solamente negli Stati Uniti, Brasile, India e in pochi altri paesi.

Figura 3.18: Logo di Google Colab [10]

Capitolo 4

Esperimenti e risultati

La fase sperimentale di questo lavoro ha l'obiettivo di analizzare e confrontare le prestazioni di diverse soluzioni di password cracking targettizzato. Questo passo capitolo di testing è stato realizzato seguendo cinque macro step:

1. Ricerca dei corretti dataset;
2. Divisone in training-set e test-set dei dataset trovato;
3. Creazione dei dizionari finali da testare;
4. Generazione delle password con i vari strumenti;
5. Cracking effettivo tramite Hashcat.

Oltre ai test sui singoli strumenti (PCFG-Cracker$_{OMEN}$, PassGAN e PRINCEProcessor) valuterò la bontà del CPG utilizzandolo assieme ad alcuni di questi tool (PCFG-Cracker$_{OMEN}$ e PRINCEProcessor), come mostrato in figura 3.1.

4.1 Raccolta dei Dataset

4.1.1 Descrizione

Per i miei scopi userò tre dataset preliminari (ovvero che in un secondo momento andranno a formare i vari train-set): il data leak (perdita di dati) di *ClixSense*, contenente dati personali e password di migliaia di utenti, il dizionario generico *Rockyou* ed infine il vocabolario specifico per le 50 mila persone obbiettivo generato tramite *CPG*.

ClixSense

In rete sono disponibili diversi dataset per la valutazione di algoritmi di password cracking. Per questo progetto sono stati valutati tre dataset contenenti dati di utenti reali resi pubblici ain seguito ad attacchi informatici:

- **Ticketfly**. Nel 2018 l'azienda statunitense, con sede a San Francisco, che si occupa della grande vendita di biglietti di diverso genere fu esposta ad un data leak. Gli hacker si impossessarono di più di 25 milioni di informazioni personali come nome, cognome, città. Tra questi innumerevoli dati mancava però la password legata all'utente, fondamentale per i nostri scopi.

- **ClixSense**. ClixSense (ora Ysense) è un sito che permette di guadagnare denaro attraverso dei sondaggi online. Nel 2016 questo sito fu preso di mira da un gruppo hacker che riuscì a prelevare dai loro database 1 milione di dati tra cui password personali.

- **Libero**. Sempre nel 2016 Libero Mail subì un attacco informatico da parte di un gruppo internazionale di maleintenzionati informatici. Questi ultimi resero pubbliche le informazioni di 700 mila persone.

Il primo ed il terzo dataset avevano problemi legati alla scarsità/mancanza di dati: le istanze erano in numero adeguato ma le loro proprietà non erano adatte per le mie intenzioni. Quindi ho deciso di optare per il dataset di *ClixSense*. Questo dataset è composto da 11 features:

- Username - nickname usato per accedere al sito.
- Password - password in chiaro usata per l'accesso.
- First Name - nome dell'utente.
- Last Name - cognome dell'utente.
- Email - email d'accesso al sistema.
- Address - indirizzo della persona.
- City - città di residenza.
- Country - paese di provenienza.
- Zip - codice postale.
- Date of birth - data di nascita.
- Gender - sesso.

Di questo dataset non userò tutte le colonne ma soltanto la data di nascita, la città, il cognome, il nome e lo username. Questa scelta ha una doppia motivazione: alcuni miei test preliminari hanno evidenziato come queste informazioni personali siano le più rilevanti; inoltre per motivi computazionali è preferibile ridurre il numero di campi passati in input all'algoritmo CPG. Inoltre, a causa della limitata disponibilità di risorse (tempo, hardware, ecc.) i test saranno eseguiti utilizzando un subset di tutte le milioni di righe del dataset. Questa scelta è ampiamente utilizzata in letteratura, dove molte ricerche utilizzano dataset di 200/300 mila istanze, con divisoni tra training-set e test-set tipicamente del 70-30 o 80-20.

Rockyou

Nel 2009 una società di nome RockYou è stata hackerata. L'attacco avrebbe generato ben pochi problemi all'azienda se avessero memorizzato tutte le loro password crittografate e non in chiaro come è stato fatto. Come spiegato nel primo capitolo Rockyou è un dizionario di password utilizzato per eseguire diversi tipi di attacchi di cracking delle password. È la raccolta delle password più utilizzate al mondo. E' possibile trovare in Kali Linux, sistema operativo usato anche per queste verifiche, rockyou.txt per impostazione predefinita. Tale wordlist sarà usata in combinazione ad altri dizionari. Al momento l'ultima versione di rockyou contiene circa 8 miliardi di password, ma il dizionario che userò sarà quello "storico" contenente circa 14 milioni di password.

Dizionario CPG

L'ultimo dataset, che formerà i diversi train-set, è generato dal CPG ed occuperà circa 2GB di memoria secondaria. Il dizionario è stato creato attraverso i dati personali delle 50 mila persone target.

4.1.2 Pulizia dei dati

Il set di dati principale ClixSense andava corretto per via del numero eccessivo di righe e per la mancanza di alcune informazioni. Se un valore di una colonna di nostro interesse per un utente utilizzato come target presentava il valore *Nan* rischiava di essere male interpretato dal nostro sistema.

Per correggere i dati agevolmente è stato creato innanzitutto un database di appoggio sul DBMS *MySQL*, utilizzando l'interfaccia di *phpMyAdmin*, in locale con *XAMPP*. Il file del data leak era inizialmente in formato .sql, tipologia di formato che creava problemi. Quindi è stato istanziato il database tramite quel file. Il DB, una volta completato, è stato esportato in formato .csv.

Lo step successivo è stato quello di estrapolare solo i dati necessari ed eliminare le righe contenenti informazioni nulle. Questo stadio della fase di pulizia dei dati è stata raggiunta utilizzando uno script Python in figura 4.1. Come prima cosa sono state importate le librerie *csv*, *Pandas* utili per gestire tabelle. In secondo luogo sono stati rimpiazzati i valori nulli (vedi figura 4.1); infine sono stati prelevati in maniera casuale le righe (ovvero gli utenti) da inserire nei training-set e test-set (assicurandosi che un utente non fosse in ambo gli insiemi).

```
import csv
import pandas as pd

data = pd.read_csv("ClixSenseDBLeaked_CSV.csv", delimiter=",",
                   error_bad_lines=False, encoding='latin-1')
data = data.fillna('')

first_half_data = data.head(500000)
second_half_data = data.tail(500000)

data_200k_train = first_half_data.sample(200000)
data_50k_test = second_half_data.sample(50000)
```

Figura 4.1: Codice per la pulizia dei dati su Colab [10]

I training set e test set per ClixSense sono stati testati con tre diverse divisioni. Se un oggetto utente era già in un set, allora non era sicuramente nell'altro, inoltre non sono state rimosse le password duplicate per coeranza con quanto scritto sopra e quanto studiato nei paper. Di seguito i dataset parziali testati con una breve spiegazione:

1. **1 milione di istanze: 700 mila train, 300 mila test/target (70-30)**. I test inizialmente erano stati apportati sulla metà dei dati a disposizioni dal DB di Clixsense. Dopo poco però mi sono reso conto di quanto tempo e risorse mi sarebbero state necessarie per ultimare tutti i test, in particolare quello di PassGAN.

2. **100 mila istanze: 70 mila train, 30 mila test/target (70-30)**. Come seconda fetta di dataset si è usata una porzione di 100

mila istanze. Le risorse consumate erano adeguate ma i risultati erano troppo scarsi a causa del basso numero di password.

3. **250 mila istanze: 200 mila train, 50 mila test/target (80-20)**. L'ultima divisione è quella usata nel mio esperimento: mi ha permesso il giusto trade-off tra consumo di risorse e risultati accettabili.

In conclusione a questa fase sono state estrapolate dai dataset solo le informazioni che mi sarebbero servite per creare i dizionari finali, colonna per colonna (figura 4.2). Nel caso della tabella del training set (200 mila utenti) si è estrapolata solo la colonna password, mentre nell'altra tabella (50 mila utenti) sia le password sia le varie informazioni personali.

```
dati = pd.Series(data_50k_test['city'])
with open("testSet50kCity.txt", "a", encoding='latin-1') as city_file:
  for dato in dati:
    print(dato, file=city_file)
```

Figura 4.2: Esempio di estrazione di una colonna (in questo caso "city") salvata in un file esterno.

4.1.3 Manipolazione

La fase di manipolazione dei dati racchiude tutte quelle attività che ho compiuto per arrivare dai dati "grezzi" ai dizionari di input per i nostri software.

I file ricavati dal passaggio precedente sono inutilizzabili se presi singolarmente: infatti i tool che uso accettano un dizionario unico e non più dizionari. E' stato necessario fondere le parole sensibili in un unico file, in cui l'ordine delle parole è importante. L'ordinamanto utilizzato è "per utente": nome, cognome, username, data di nascita e città e così via per 50 mila utenti. Lo script Python che esegue la fusione è mostrato in figura 4.3

```
for x in range(1,int(num_person)+1):
    for info in personal_info:
        print(info.readline().strip(), file=output)
```

Figura 4.3: Ciclo principale del multi_cpg_file_creator.py dove per ogni persona target scrive ordinatamente sul file di output nome, cognome, username, data di nascita e città.

Si avrà ora un documento di testo dove sono racchiusi dati sensibili delle 50 mila persone target: chiameremo questo file *PI50k.txt*.

L'utilizzo del CPG nel mio progetto doveva interessare la generazione di password a partire da più parole possibili di input. Come visto il dataset di ClixSense ha in sè diverse colonne di dati personali: lo scopo era quello di usarle pressochè tutte. Il CPG però è stato creato per generare una lista molto sostanziosa di password a partire anche da poche parole. Questo è un bene nel caso si dovesse cercare una password per una persona sola ma un male nel caso in cui si avessero molte persone da targettizzare automaticamente. Se il fine è quello di creare possibili password per tantissime persone impiegando per ognuna molte notizie personali allora si rischia di eccedere con le risorse a disposizione. A conferma di questo fatto sono stati eseguiti dei test per vedere come variava la dimensione del file di output in funzione delle parole in input. Ciò è graficamente visibile in figura 4.4. Il test è stato eseguito identificando come target fittizio una persona-in-azienda:

- **test1, 2 parole in Input, il minimo ovvero nome target e nome azienda**: 22,15 secondi — 9,49MB — 592 886 psw
- **test2, 7 parole in Input**: 127,49 secondi — 91,8MB — 4 230 915 psw
- **test3, 13 parole in Input**: 336,91 secondi — 329MB — 15 117 068 psw
- **test4, 31 parole in Input**: 2424.87 (circa 40 minuti) secondi — 3,09GB — 153 863 871 psw

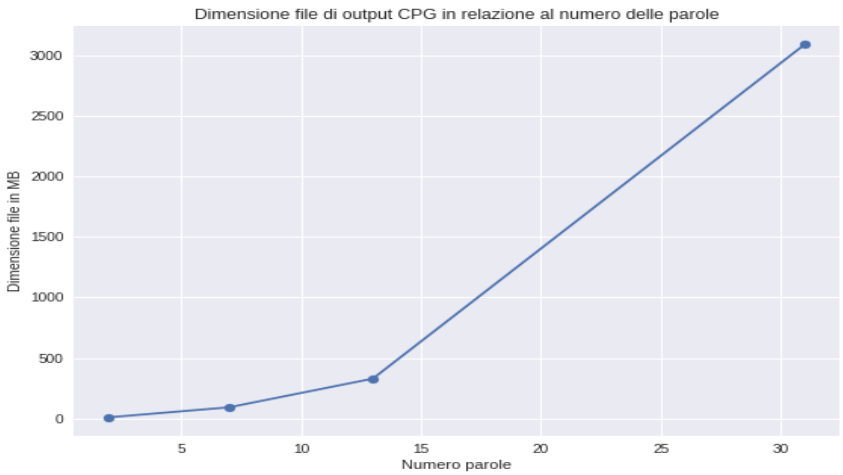


Figura 4.4: Grafico che mostra l'aumento di dimensione del file di output di CPG in relazione alle parole in input.

Ovviamente vanno osservati anche i tempi necessari per la creazione del file, che risultano altrettanto impegnativi. Puntualizziamo che il runtime e le dimensioni del file variano, seppur in maniera minore, dalla lunghezza delle stringhe in input: una parola da 4 caratteri impigherà meno tempo rispetto ad una da 10 per essere processata. Alla luce di questi fatti ho ridimensionato il lavoro del CPG: è evidente come non si possa impiegare, con i mezzi a mia disposizione, diversi minuti e tantissimo spazio per ognuna delle 50 mila persone target.

Si è deciso così di costruire una seconda versione (*mini_ cpg.py)* che riduceva le trasformazioni alle password, diminuendo di conseguenza tempo e spazio necessari, ma mantenendo comunque la "filosofia" di base con cui è stato creato. Le modifiche iniziali al codice sorgente di CPG andarono ad interessare alcune funzioni che adoperavano eccessive risorse (in gergo *bottleneck*): principalmente si cambiarono gli input e la logica interna di diverse funzioni combinatorie o che eseguivano traduzioni leet. Questo permise di ridurre i costi ma non abbastanza se iterato per 50 mila persone. La scelta seguente fu quella di ridurre le parole in ingresso al CPG: si passò dalle 10/11 pensate inizialmente alle 5 finali. A fine procedimento sono arrivato ad ottenere all'incirca **10/15 mila password in 0.03 secondi con un peso compreso tra i 100 e 200 kB** per persona. Risultato di ottime prospettive. Superato questo intoppo fu facile generare il dizionario completo, dal peso di circa 8GB. La stragrande maggioranza delle stringhe nel test-set, da 50 mila istanze, arriva fino a 14 caratteri: questo fatto mi ha portato a limitare ulteriormente la password, cancellando quelle che avevano lunghezza maggiore di 14 ed ottenendo un documento di peso attorno ai 2 GB. Il file finale lo chiamerò *generatedCPG14.txt*. I dizionari che effettivamente userò nella fase di train per i miei esperimenti sono:

- **Dizionario con le password del train-set da ClixSense (200kClixSense)**. La prima tipologia di test impiegherà solamente il set di allenamento estrapolato dal DB di ClixSense, che contiene 200 mila espressioni.

- **Dizionario con train-set e rockyou (rockyou_200k)**. In seguito eseguirò i test sul vocabolario di parole del set da 200 mila utenti di ClixSense, quindi tendenzialmente *simili* a quelle del test-set (si pensi alle regole di creazione password per un dato sito che impone parole lunghe almeno 8 caratteri, 2 numeri, ecc. : questa peculiarità le rende simili), unito al dizionario generico *rockyou*.

- **Dizionario con train-set, rockyou e informazioni personali dei target (rockyou_200k_pi50k)**. Il terzo lessico viene generato dalle 200 mila password di ClixSense, rockyou e il documento ottenuto dopo la prima parte di manipolazione dati (PI50k.txt).

- **Dizionario con train-set, rockyou, PI50k e password generate da CPG (rockyou_200k_pi50k_CPGgenerated)**. Il quarto e ultimo set di dati che useremo per le fasi di training si ricava dall'unione dei quattro sotto-vocabolari principali: train-set di ClixSense, rockyou, PI50k e generatedCPG14.

Il nome dato a questi dizionari non è casuale ma rispecchia l'ordine con cui si ritrovano le parole in essi. Ad esempio nel dizionario *rockyou_200k_pi50k* prima si troveranno le password di rockyou, poi le 200 mila di ClixSense e in seguito le informazioni personali delle 50 mila persone target.

Analizzo ora le password di test, ovvero quelle target. Le password da testare saranno 50 mila: chiamerò il file che le contiene **50kClixSense**. La sola mutazione che praticherò su tali parole è dovuta alla funzione hash **MD5**. MD5 (Message Digest versione 5) costruisce un'immagine di 128 bit per qualunque messaggio. Ideata da Rivest nel 1992, verrà impiegata per molto tempo in applicazioni crittografiche. Anni più tardi sono stati scovati diversi attacchi che ne hanno ridotto l'uso. Data però la sua semplicità resta largamente utilizzata. Per la sua efficenza e rapidità viene definita una funzione di hash *rapida*. L'applicazione automatica di MD5 alle password è stata eseguita da un mio banale codice Python che leggeva di volta in volta le stringhe, le alterava ed infine le scriveva su un secondo file. L'archivio di password di test andava alterato per essere accettato da Hashcat: quest'ultimo infatti non accetta confronti con password di destinazione in chiaro.

4.1.4 Training e Test Set

Nel dataset di test, ovvero quello contenente le password di 50 mila persone target elaborato dal database di ClixSense, si incontreranno password uguali: questo per emulare il più possibile una reale sessione di password cracking. Il dataset contiene password reali e molte di queste contengono informazioni personali più o meno in chiaro come si può notare dalla figura 4.5. Di seguito un'analisi delle password di test, effettuata con PACK. Il primo grafico (figura 4.6) mostra la distribuzione della lunghezza delle password (come anticipato la stragrande maggioranza di password hanno lunghezza minore o uguale a 14) mentre il secondo (figura 4.7) mostra, in percentuale, che tipo di grammatica/charset utilizzano: dal grafico si intuisce come la maggior parte di password abbia un charset limitato: ciò è sintomo di password non troppo robusta.

```
james111076
michele89LOVE
laurita
judith12345
sonusauk
waganish
dima1979
sahara2475
```

Figura 4.5: Esempi di password che si trovano del dataset target.

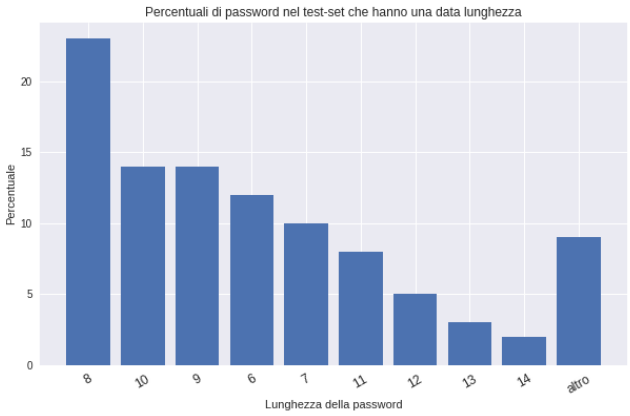


Figura 4.6: Grafico che mostra quante password del test-set hanno una data lunghezza.

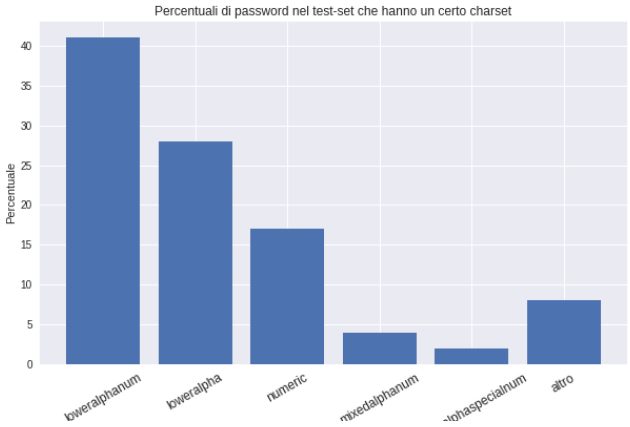


Figura 4.7: Grafico che mostra quante password del test-set hanno un dato charset.

Si estende poi l'analisi anche a due liste di password tra quelle che andranno a comporre i quattro train-set finali: le 200 mila password estrapolate dal DB di ClixSense e le password generate dal CPG. Non andrò ad analizzare Rockyou in quanto decine di analisi sono già state effettuate su questo dizionario (ad esempio [19]). Nemmeno i vari train-set completi andrò ad analizzare: effettuare questa analisi su di essi non avrebbe senso, infatti i grafici avrebbero tutti risultati simili. Per la prima lista di password verificherò la distribuzione della lunghezza e del charset di esse ed infine le 10 stringhe più ricorrenti all'interno del file (figure 4.8, 4.9 e tabella 4.1). Per la seconda analisi studierò solo le distribuzioni (figure 4.10 e 4.11) : non avrebbe senso cercare le password più frequenti in un dizionario che per sua costruzione non ha duplicati. Di seguito i risultati degli studi.

- **Dizionario con 200 mila password di ClixSense**.

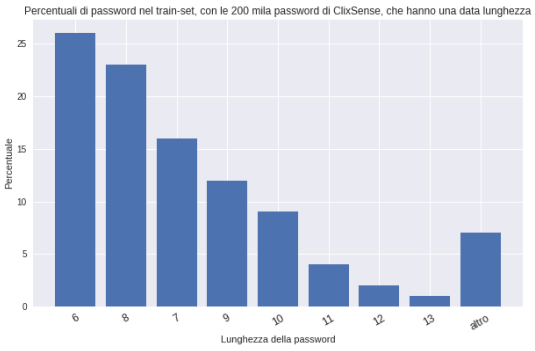


Figura 4.8: Il grafico mostra quante password hanno una data lunghezza. La distribuzione è simile al test-set essendo presi dallo stesso set di dati.

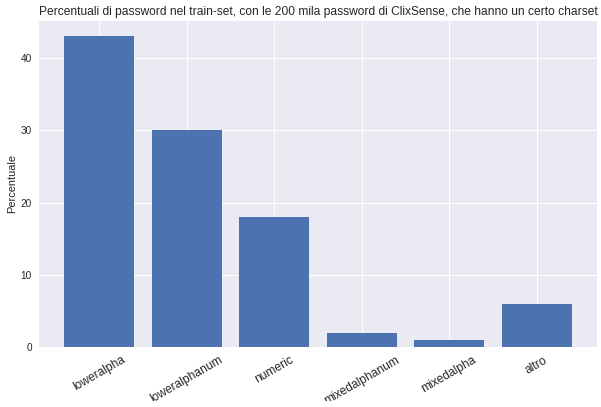


Figura 4.9: Il grafico mostra quante password hanno una certa grammatica. La distribuzione è simile al test-set essendo presi dallo stesso set di dati.

Password	**Occorrenze**
123456	3839
123456789	522
111111	414
password	376
12345678	328
1234567	235
qwerty	224
clixsense	221
iloveyou	196
123123	182

Tabella 4.1: Le 10 password più usate nel set da 200 mila istanze prelevato da ClixSense.

- **Dizionario generato da CPG sulle 50 mila persone target con password limitate a 14 caratteri**.

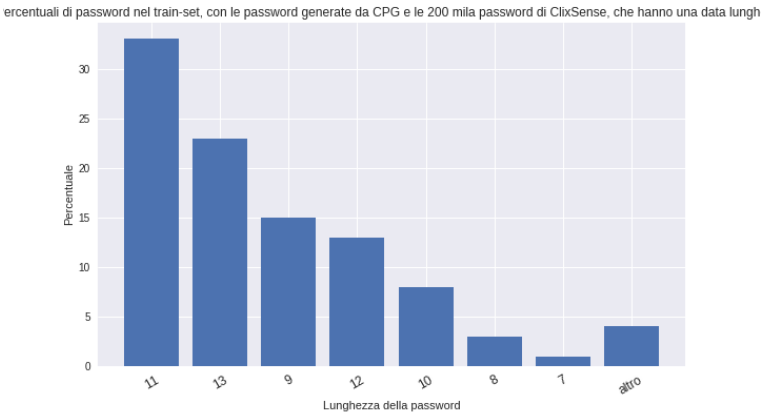


Figura 4.10: Il grafico mostra quante password hanno una data lunghezza. La distribuzione è diversa dai due set precedenti, indice di poche password comuni.

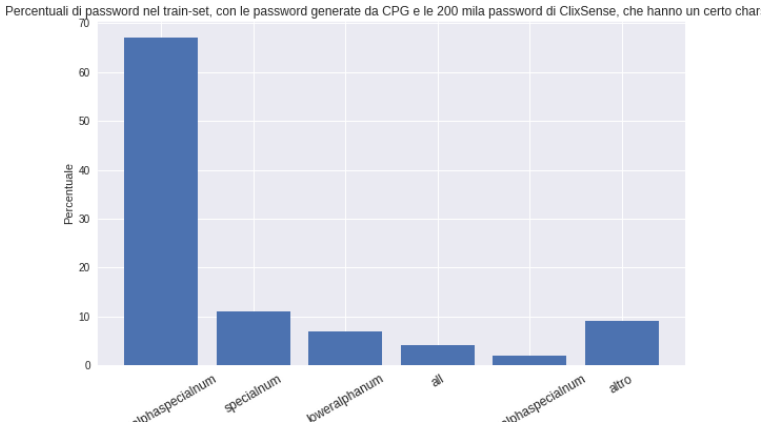


Figura 4.11: Il grafico mostra quante password hanno una certa grammatica.La distribuzione è diversa dai due set precedenti, indice di poche password comuni.

Queste analisi mi hanno permesso di evidenziare differenze e similitudini tra alcuni dei vocabolari principali del nostro documento: se da un lato si è appreso quanto i due set di dati presi dal file di ClixSense siano simili, dall'altro ho messo in evidenza le differenze tra il dizionario generato dal CPG e le password di ClixSense. Questa seconda affermazione mi fa presagire che le password da me generate non siano troppo efficaci in questo esperimento e con queste condizioni. Sviluppi futuri potrebbero portare al miglioramento di CPG in modo che tenga conto anche delle distribuzioni delle password sorelle a quelle target.

Ciascun dizionario contiene un numero diverso di parole, come mostrato nella tabella 4.2. A causa di ciò, anche il numero di ipotesi generate varierà dal dizionario di input.

Train-set	**Numero di password**
200kClixSense	200 000
rockyou_200k	14 544 391
rockyou_200k_pi50k	14 794 386
rockyou_200k_pi50k_CPGgenerated	204 986 034

Tabella 4.2: Numero di password contenute in ciascun dei quattro dataset di train.

L'utilizzo di dati personali delle persone target (ovvero quelle di testing) nei dizionari di train può risultare strano: convenzionalmente, nei software basati in qualche maniera sul Machine Learning, non si dovrebbero usare informazioni relative al test-set. Ma vista la logica su cui si basano i nostri strumenti, era più sensato inserire nei vari train-set le informazioni personali degli utenti target. Questa scelta non è così distaccata dalla realtà: nei database le informazioni personali di una persona non vengono criptate tendenzialmente quindi, una volta entrati in possesso di un database, l'attacco alle password hashate può migliorare con dei dati anagrafici.

4.2 Protocolli di testing

I potenziali protocolli da seguire, per effettuare al meglio questi test, sono principalmente due: attaccare in un'unica sessione di cracking tutte gli utenti target oppure attaccare di volta in volta le singole persone. Assieme ad uno dei dipendenti di Cyberloop (azienda affiliata a Clipperz tramite il gruppo Imola Informatica presso la quale ho svolto i miei tirocini), Canducci Marco, si è optato di scegliere per la prima tipologia di protocolli: generando un unico file per tutte le persone. Sviluppi futuri potrebbero saggiare l'altro protocollo di testing individuato. Per tutta la fase di sperimentazione userò test-set e target-set (o password target)

come sinonimi: infatti le password degli utenti target sono quelle che verranno utilizzate come test-set.

In generale si definisce un attacco migliore rispetto ad un altro se a parità di risorse impiegate e dizionario di parole utilizzate il recupero di password risulta maggiore. Le metriche con cui misurerò la bontà dei test sono standard e si ritrovano in [40]. Le sessioni di cracking dureranno tutte un'ora. Questo tempo, confrontato con le sessioni reali, è esiguo ma mi ha permesso di ottenere risultati discreti.

Le metriche sono le seguenti:

- **Numero di password craccate**. Metrica classica e fondamentale nel password cracking. Mostra quante password una sessione di cracking è riuscita a recuperare. Ovviamente due sessioni per essere confrontate dovranno avere o lo stesso numero di password tentate o lo stesso tempo di cracking a parità di dizionario di input.
- **Numero di password craccate in relazione al tempo**. Approccio simile a quello citato poche righe fa, ma si concentra maggiormente sull'avanzamento della sessione lungo lo scorrere del tempo. Permette di analizzare maggiormente la potenza del tool utilizzato per la seduta di cracking.
- **Numero di password craccate per password tentate**. Questo valore mostrerà il rapporto tra le password craccate e quelle tentate. Più il rapporto si avvicinerà ad 1, maggiore sarà rilevante la sessione. In situazioni normali molto vicino a 0.

$$0 \leq \frac{PASSWORD_{recuperate}}{PASSWORD_{tentate}} \leq 1$$

In figura 4.12 vengono spiegati gli step necessari, per un dato esperimento, per arrivare ad un risultato. Inizialmente i dati di addestramento vengono passati in ingresso al modulo di training dello strumento che si intende testare (es: se testo il PCFG-Cracker invierò i dati al pcfg-trainer). Il software di train genererà un insieme di regole (ruleset): questo è specifico per ogni set di dati che ho usato come allenamento. In questi strumenti se si cambiasse il set di train si dovrebbe riprocedere all'addestramento dell'intero modello. L'esperimento procede inserendo, in ingresso al modulo di password generation dello strumento in considerazione, le regole create (es: se testo il PCFG-Cracker invierò le regole al pcfg-guesser). Il guesser restituirà delle possibili password basate sulle regole inserite. Infine le password generate artificialmente saranno l'input del password cracker (nel mio caso Hashcat) che consegnerà i risultati finali.

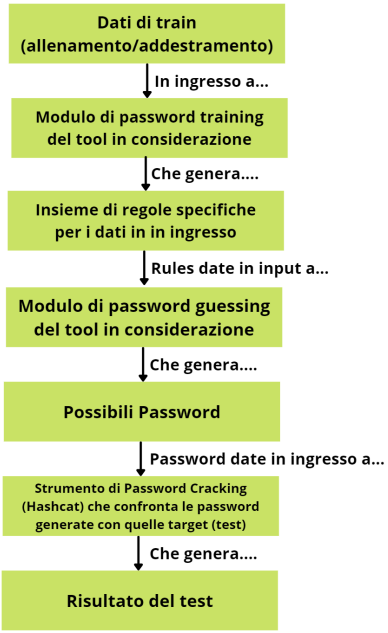


Figura 4.12: Il disegno mostra i passaggi eseguiti in ogni esperimento.

4.3 Descrizione esperimenti

Presento ora con maggior dettaglio gli esperimenti da me condotti. Questi test hanno il fine di identificare quali strumenti siano migliori per un attacco di password cracking targettizzato. Le prove da me condotte si concentrano anche sul consumo di risorse: non tutti hanno a disposizione infrastrutture estremamente potenti. Nel mio caso infatti tutte le verifiche sono state svolte su macchina virtuale *Kali Linux* o *Google Colab* ed in piccola parte su macchina fisica (utilizzata solamente per la generazione di password con CPG): questo perchè mi risultava impossibile, soprattutto in tempi di Covid dove si svolge tutto da remoto, occupare il mio computer per un solo compito per un lungo tempo.

Kali Linux è una distribuzione Linux basata su Debian. Il sistema è indirizzato principalmente per l'ambiente della sicurezza informatica. L'uso di Kali per i miei scopi era necessario: molti dei software utilizzati risultano pre-installati (come *Hashcat*) e di facile utilizzo dal terminale. Per la VM di Kali Linux sono stati dedicati 8GB di RAM, 50 GB di hard disk e 4 processori. I processori e la RAM sono stati i due settings più difficili da indovinare. I nostri esperimenti consumavano infatti molte risorse ed è stato necessario eseguire diversi test preliminari per capire quale potesse essere il giusto equilibrio tra macchina fisica e virtuale. Per mezzo di Kali linux ho testato il **PCFG-Cracker**$_{OMEN}$, sia in fase di guessing sia in fase di cracking, **PRINCEProcessor**, anche questo in

ambedue le fasi, ed infine anche il **PassGAN** ma solo durante l'effettiva fase di cracking.

La macchina virtuale messa a disposizione da *Google Colab* è stata di vitale importanza con **PassGAN**. PassGAN è infatti una rete neurale molto pesante e che richiedeva moltissime iterazioni per essere addestrata in modo corretto: moltissime iterazioni volevano dire runtime e disponibilità hardware enormi. Infatti, dopo pochissime ore di test su macchina fisica, mi resi subito conto dell'impraticabilità di esso sul mio calcolatore. I motivi della difficoltà d'uso non erano solo a livello di risorse ma anche a livello di versione delle librerie. Il codice sorgente di PassGAN utilizza vecchie versioni di alcuni moduli altamente impiegate nel ML come Numpy, Tensorflow. Una volta effettuato il downgrade di tali moduli PassGAN lavorava correttamente ma con risultati deboli. Gli scarsi risultati erano da imputare principalmente alla scarsa potenza della mia GPU. E' risaputo come alcuni tipi di hardware come GPU (Graphics Processing Unit) o TPU (Tensor Processing Unit, scheda costruita da Google per essere impiegata con alcune reti neurali) possano incrementare notevolmente le prestazioni in un modello di Machine Learning: Colab mette a disposizione gratuitamente anche queste due. L'uso di PassGAN su Colab è stato facilitato da alcuni file .ipynb (tipo di documento utilizzato da Google Colab) che si trovavano sulla repository GitHub di PassGAN [18] che svolgevano passo per passo la fase di train del modello e la fase di guessing generation. La VM di Colab mi metteva a disposizione risorse variabili a seconda dell'uso che ne avevo fatto nei giorni scorsi e a seconda della domanda da parte di altri utenti. Tipicamente nei test mi venivano forniti 12 GB di RAM, 75 GB di disco fisso e una GPU tra Nvidia K80s, T4s, P4s e P100s che non era possibile scegliere ma veniva assegnata di default.

In situazioni reali una sessione di password cracking può essere strutturata a "layer": utilizzando di volta in volta strumenti e idee diverse, al fine di craccare più password possibili per un determinato dataset. Ad esempio in certe situazioni un gruppo di password può essere più attaccabile con una determinata regola rispetto ad un altro gruppo: si tentano quindi diverse strade per arrivare a più password possibili. Io mi focalizzerò su un solo "layer" di questo ipotetico attacco, cercando di massimizzare in esso il recupero di password.

Gli esperimenti tratteranno solamente attacchi con strumenti innovativi, l'intenzione del documento è appunto confrontare attacchi alle password all'attuale stato dell'arte. Come ampiamente spiegato per sviluppare queste offensive utilizzerò sistemi di password guessing uniti ad Hashcat. In particolare vorrei analizzare velocemente una scelta sperimentale applicata ad Hashcat. Hashcat dà la possibilità di applicare alcune rules ottimizzate alla password che riceve in input, per aumentare ulterior-

mente lo spazio delle password tentate. Ad esempio esiste un file di rules preparato appositamente per l'uso di PRINCEProcessor con Hashcat [28] ma non solo. Questo avrebbe portato ad un'esplosione dei test per determinare quale rules sia migliore per un determinato attacco: di conseguenza ho deciso di non utilizzare nessuna rules ottimizzata nelle mie analisi. Esperimenti futuri potrebbero studiare gli effetti di queste regole sugli strumenti e quale di esse ottiene maggior successo.

Figura 4.13: Generico comando per una fase di cracking con Hashcat.

In figura 4.13 viene indicata un'approssimazione del comando che lancerò per una data sessione di cracking (a seconda del password guessing tool varieranno le opzioni). Si capisce chiaramente come le password che vengono man mano generate sono poi poste in input ad Hashcat che a sua volta riverserà i risultati finali su un file di testo. Nella tabella 4.3 vengono indicati i vari dizionari usati nelle prove: il set di training e il set di destinazione che attaccheremo.

Train-set	**Test-set**
200kClixSense	50kClixSense
rockyou_200k	50kClixSense
rockyou_200k_pi50k	50kClixSense
rockyou_200k_pi50k_CPGgenerated	50kClixSense

Tabella 4.3: In tabella sono racchiusi i vocabolari principali che userò.

I primi tre train-set, che si notano nella tabella 4.3, saranno eseguiti su PCFG-Cracker$_{OMEN}$, PassGAN e PRINCEProcessor. Il quarto e ultimo train-set della figura verrà invece sviluppato solamente sul PCFG-Cracker$_{OMEN}$ e sul PRINCEProcessor, a causa dei tempi estremi di PassGAN (su un file così grande impiegherebbe giorni per completare il lavoro). In pratica con questo ultimo dizionario di train si testerà il CPG e PCFG-Cracker assieme e poi il CPG e il PRINCEProcessor insieme (come indicato negli ultimi due step della figura 3.1). Nei vari comandi per gli esperimenti che eseguirò si vedrà come spesso la fase di allenamento del modello ha un comando; mentre la fase di generazione effettiva delle password e il cracking di queste ne avranno un altro.

Di seguito i comandi utilizzati per gli attacchi con i vari tool:

- **PCFG-Cracker**$_{OMEN}$.
  - Comando generico per il Training:

    ```
    python3 pcfg_cracker/trainer.py -r
    RULESET_NAME -t TRAIN_SET_PATH -e latin-1
    ```

    In questa fase di train verranno generate le regole utili al secondo script python del PCFG (*RULESET_NAME*). L'opzione *-e latin-1* indica il charset da utilizzare in questa fase per riconoscere le parole in input. Tutti i vocabolari sia di test sia di train utilizzano la codifica latin-1.
  - Comando generico per il Guessing + Cracking:

    ```
    python3 pcfg_cracker/pcfg_guesser.py -r RULESET_NAME |
    hashcat --potfile-disable -O -m 0 -a 0 PASSWORD_TARGET_PATH
    --runtime=3600 --status-timer=120
    -o RESULT_FILE >> STATUS_FILE
    ```

    In questa fase si genereranno effettivamente le password, basate sulle rules identificate prima, che saranno inviate ad Hashcat. Tra le opzioni rilevanti di Hashcat si noti *–potfile-disable*, *-O*. Il primo flag disabilita il **potfile** ovvero un file in cui Hashcat salva tutte le password recuperate da sessioni precedenti: il primo controllo che Hashcat esegue una volta avviato è se alcune delle password target sono già state recuperate e contenute nel potfile. Il potfile va quindi disabilitato per evitare di falsare i miei test. E' estremamente utile in situazioni reali in cui si cerca man mano di craccare più password possibili da un dataset. La seconda opzione citata è una feature di Hashcat chiamata Optimized Kernel o **-O** o –**optimized-kernel-enable**. Tale proprietà, che prima di Hashcat 4 era di default, permette di velocizzare il processo di cracking ma limita la lunghezza della password ipotizzata (in genere la limita ad un massimo di 55 caratteri) [27]. Per capire l'effettiva bontà di ciò, ho effettuato un brevissimo test preliminare applicando e poi rimuovendo il flag **-O** su un attacco Mask-Attack (Brute force) ad una password criptata in MD5. Nelle foto 4.14 e 4.15 sotto possiamo vedere l'impatto che ha l'optimized kernel: nel primo caso arriviamo a toccare i 190 MH/s, nel secondo invece ci fermiamo all'incirca sui 95 MH/s. Da notare anche il tempo stimato per controllare tutto lo spazio delle password possibili (keyspace): nel primo caso e' meno della meta' rispetto al secondo.

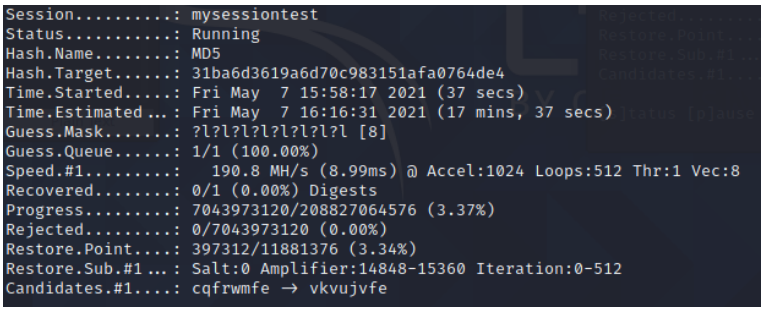


Figura 4.14: In figura il primo test effettuato sul flag Optimized Kernles. Il flag **-O** e' attivo.

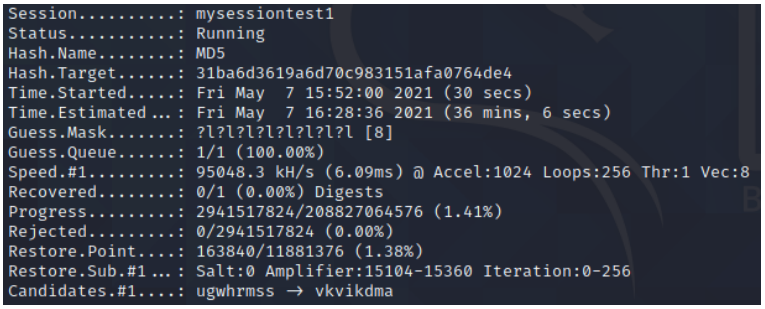


Figura 4.15: In figura il secondo test effettuato sul flag Optimized Kernles. Il flag **-O** e' disattivato.

Le opzioni restanti usate sono relative al tipo di attacco (*-m 0* funzione hash MD5 che bisogna applicare, *-a 0* tipologia di attacco a dizionario), al tempo di esecuzione del processo (*runtime=3600* indica un'ora di lavoro), al file di output (*-o RESULT_FILE* all'interno la password in chiaro e la sua corrispondente funzione hash) e ad ogni quanti secondi deve inviare uno status sui progressi (*–status-timer=120* indica uno status ogni 2 minuti); tali progressi saranno inviati al file denominato STATUS_FILE (*» STATUS_FILE*).

- **PassGAN**.
  - Training e Password Guessing per PassGAN:
    Come è stato anticipato nelle righe precedenti l'uso di PassGAN passerà ampiamente attraverso Colab. In particolare sia la fase di addestramento della rete neurale sia la fase di password generation avverrà sul portale di Google. Nel capitolo 3 sono stati spiegati gli iperparametri in input, identificando criticità nella *lunghezza massima delle stringhe in output* e nel *numero di iterazioni* per la fase di train della rete. Questi due parametri nei nostri test saranno impostati rispettivamente a 10 ed a 35 mila, quando in realtà si potrebbero generare password più lunghe ed eseguire moltissime iterazioni in più. Questo perchè persino Colab, che offre un ambiente dedicato, fatica a generare password più lunghe e a portare a termine più iterazioni. Ovviamente queste scelte saranno incisive in questa fase sperimentale portando a risultati inaspettati.
  - Comando generico per il Cracking:

    ```
    hashcat --potfile-disable -O -m 0 -a 0 PASSWORD_TARGET_PATH
    PASSWORD_GENERATED_PATH --runtime=3600 --status-timer=120
    -o RESULT_FILE >> STATUS_FILE
    ```

    Il comando di cracking questa volta prenderà direttamente in input tutto il dizionario generato (e non le password singole che venivano man mano generate come nel precedente): effettuando in pratica un classico attacco a dizionario. Questo comporterà la mancanza di un valore nella metrica relativa alle password craccate rispetto allo scorrere del tempo.

- **PRINCEProcessor**.
  - Comando unico per train, guessing e cracking:
    La facilità d'uso del PRINCEProcessor si ritrova anche nel comando da lanciare per far partire l'attacco. Un unico comando per gestire l'intera offensiva.

    ```
    /princeprocessor-0.22/pp64.bin TRAIN_SET_PATH |
    hashcat --potfile-disable -O -m 0 -a 0 PASSWORD_TARGET_PATH
    --runtime=3600 --status-timer=120
    -o RESULT_FILE >> STATUS_FILE
    ```

Comandi specifici per il CPG (ricordo che userò una versione ridotta come spiegato nella sezione 4.1.3) non esistono: una volta generate le password con esso, si potranno inserire come input nei vari programmi a piacere.

4.4 Analisi dei risultati

In quet'ultima sezione del capitolo quattro analizzerò i risultati ottenuti dai test. Ricordo che l'ordine dei tool con cui sono stati eseguiti gli esperimenti si trova in figura 3.1, mentre i vari set che verranno utilizzati si trovano nella tabella 4.2. Come spiegato ad inizio della sezione 4.1.4 per emulare il più possibile una situazione reale ho deciso di non eliminare le password duplicate dal test-set: dalle figure che seguono (dalla 4.16 in poi) si noterà come le password uniche siano **48410** contro le 50 mila iniziali. Nelle figure che appariranno (ad esempio nella figura 4.16) saranno presenti ulteriori informazioni, tipiche di una sessione di cracking effettuata con Hashcat:

Speed Indica la velocità di hashing di Hashcat (tipicamente indicata in kH/s o MH/s) ovvero la velocità con cui quest'ultimo sta hashando le password in ingresso.

Recovered Indica le password recuperate da Hashcat fino a quell'istante.

Remaining Indica le password non ancora recuperate da Hashcat fino a quell'istante.

Recovered/Time Indica una stima del tempo necessario al recupero di tutte le password.

Progress Indica tutte le password generate e testate fino a quel momento.

Procedo ora con l'analisi dei risultati sui vari esperimenti.

- **PCFG-Cracker**$_{OMEN}$
  Il primo esperimento da me condotto è stato sul PCFG-Cracker che implementa anche OMEN. Il primo dataset testato su di esso è stato quello da 200 mila password estrapolate da ClixSense o, come spesso chiamato in questo documento (vedi tabella 4.2), **200kClixSense**.

```
Speed.#1.........:   413.1 kH/s (1.06ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered........: 19086/48410 (39.43%) Digests
Remaining........: 29324 (60.57%) Digests
Recovered/Time...: CUR:50,N/A,N/A AVG:318,19086,458071 (Min,Hour,Day)
Progress.........: 1035935744
```

Figura 4.16: In figura il numero di password craccate dal PCFG-Cracker$_{OMEN}$ in seguito ai dati di input **ClixSense**.

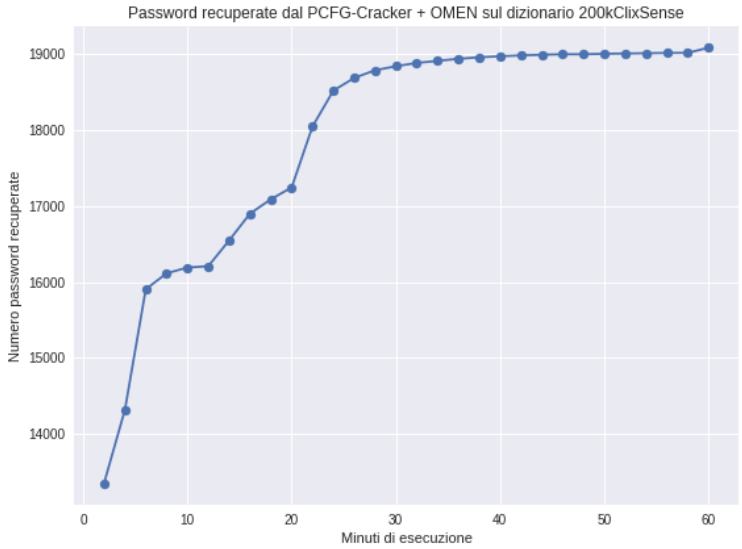


Figura 4.17: In figura il numero di password craccate dal PCFG-Cracker$_{OMEN}$ in seguito ai dati di input **200kClixSense** rispetto al tempo di esecuzione.

Questo primo test ha evidenziato l'ottima similitudine tra i due set di password estrapolati dallo stesso database di dati. Con solo 200 mila password di train il PCFG ha recuperato dopo un'ora di test il **39.43%** di password; il coefficente tra password recuperate e password generate è pari a **0,0000184** che sta a significare una password corretta ogni circa **54300** password generate. Il grafico in figura 4.17 mostra però come, date le poche password in input, dopo **circa 40 minuti** il trend di crescita del numero di password generate si affievolisca.

Il secondo test per questo strumento verrà condotto con l'ausilio del dizionario **rockyou_200k**, ovvero quel file nato dall'unione del vocabolario generico rockyou e le 200 mila password usate anche nel test precedente.

```
549 Speed.#1.........:   183.1 kH/s (1.31ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
550 Recovered........: 22595/48410 (46.67%) Digests
551 Remaining........: 25815 (53.33%) Digests
552 Recovered/Time...: CUR:38,N/A,N/A AVG:376,22594,542277 (Min,Hour,Day)
553 Progress.........: 849733770
```

Figura 4.18: In figura il numero di password craccate dal PCFG-Cracker$_{OMEN}$ in seguito ai dati di input **rockyou_200k**.

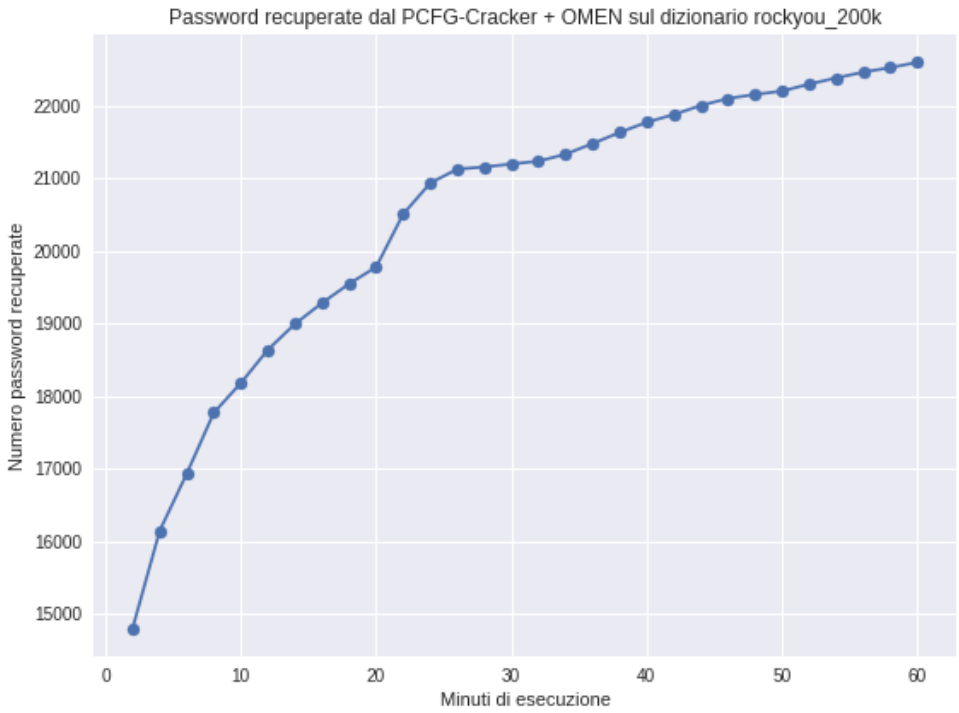


Figura 4.19: In figura il numero di password craccate dal PCFG-Cracker$_{OMEN}$ in seguito ai dati di input **rockyou_200k** rispetto al tempo di esecuzione.

Le figure 4.18 e 4.19 mostrano come un aumento delle password di train migliorino le prestazioni. In particolare le password di rockyou sono anche altamente generiche quindi ottime per qualsiasi attacco alle password: era prevedibile un aumento delle password craccate. Il test ha evidenziato una tendenza positiva nel numero di password recuperate in funzione del tempo (figura 4.18): prima di vedere una stabilizzazione della crescita sarebbe stato necessario ulteriore runtime. Le password di train hanno permesso di arrivare al **46,67%** di password in chiaro. Il rapporto tra password recuperate e password generate è pari a **0,00002659** che sta a significare una password corretta ogni circa **37600** password generate.

Il terzo ed ultimo test che ho svolto con il PCFG-Cracker ha confermato l'ipotesi sui cui questo documento si fonda: la potenza del password cracking viene **aumentata** se si usano informazioni personali legate al target. L'esperimento impiega il dizionario **rockyou_200k_pi50k**.

```
549 Speed.#1.........:   123.3 kH/s (1.15ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
550 Recovered........: 23115/48410 (47.75%) Digests
551 Remaining........: 25295 (52.25%) Digests
552 Recovered/Time...: CUR:29,N/A,N/A AVG:385,23115,554776 (Min,Hour,Day)
553 Progress.........: 806283038
```

Figura 4.20: In figura il numero di password craccate dal PCFG-Cracker$_{OMEN}$ in seguito ai dati di input **rockyou_200k_pi50k**.

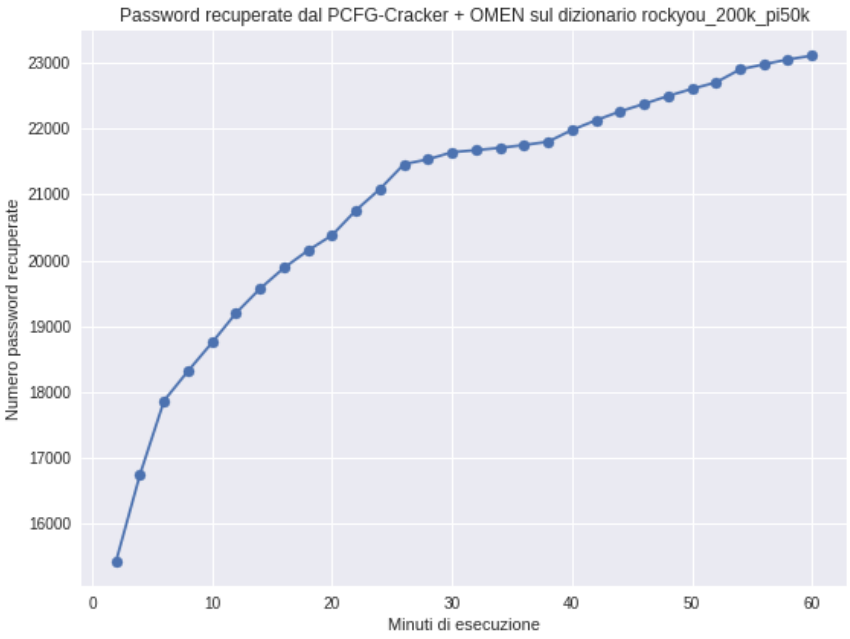


Figura 4.21: In figura il numero di password craccate dal PCFG-Cracker$_{OMEN}$ in seguito ai dati di input **rockyou_200k_pi50k** rispetto al tempo di esecuzione.

Il trend di crescita delle password craccate rispetto al tempo viene mantenuto anche in questo caso. Si noti nel grafico 4.21 una leggera flessione nella curva: probabilmente quel periodo di rallentamento nella scoperta di nuove password, è dovuto ad un uso prolungato di informazioni personali su possibili password (che miravano però a recuperare password target che non ne contenevano). Le password recuperate sono il **47,75%** del totale (ovvero 23115 su 48140). Il rapporto tra password recuperate e password generate in questo caso vale **0,00002866** o in altri termini una password recuperata ogni circa **34800** password generate.

Infine la figura 4.22 evidenzia quanto detto in precedenza: il PCFG-Cracker unito ad OMEN, a parità di risorse, dà i migliori risultati con il dizionario **rockyou_200k_pi50k**.

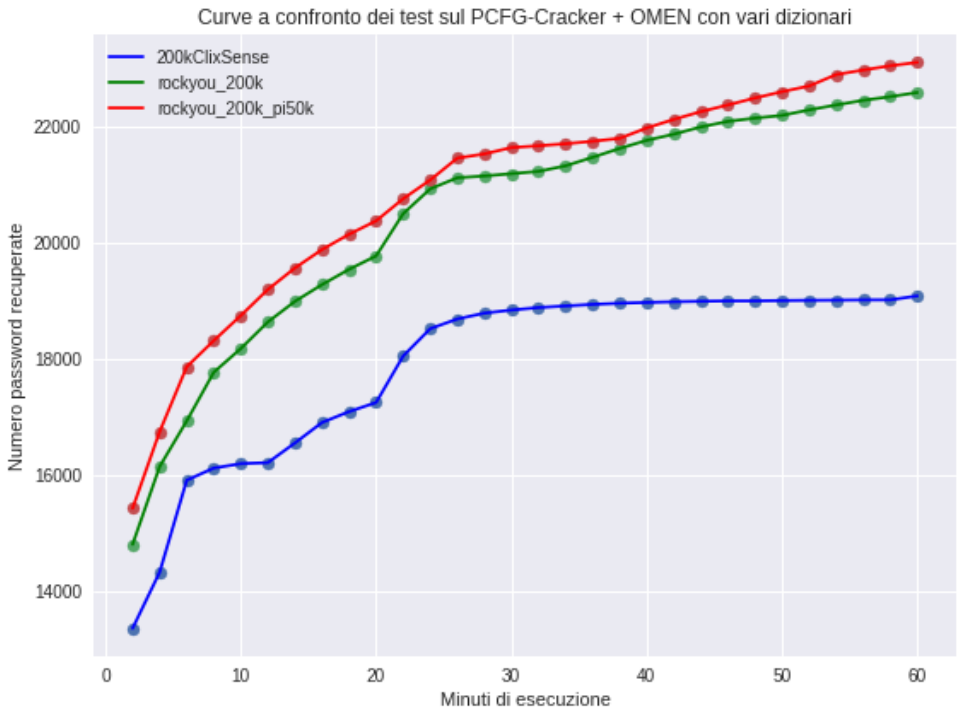


Figura 4.22: Il grafico mostra l'andamento del recupero password del PCFG in base al dizionario in input.

- **PassGAN**.
  I test su PassGAN sono stati quelli più *ardui* e che hanno portato risultati più *inaspettati*. Ardui perchè la complessità della rete neurale ha portato tempi di training e guessing molto alti; inaspettati perchè, data la potenza della rete, mi aspettavo risultati migliori.

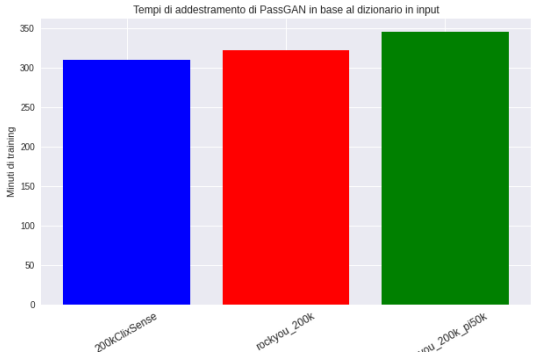


Figura 4.23: Tempo (in minuti) di allenamento della rete neurale PassGAN in base al dizionario.

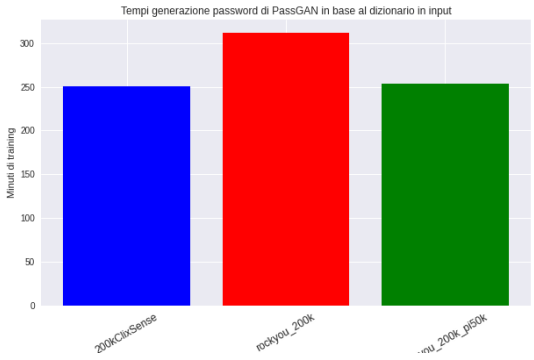


Figura 4.24: Tempo (in minuti) di generazione password della rete neurale PassGAN in base al dizionario.

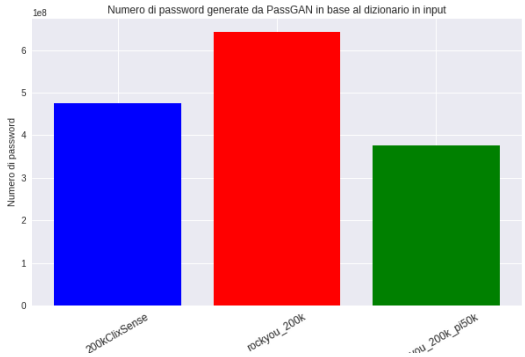


Figura 4.25: Password (in centinaia di milioni) generate da PassGAN in base al dizionario.

Le fasi di allenamento e generazione password di PassGAN erano vincolate dai tempi e le risorse che Google Colab mi metteva a disposizione. Colab non mi permetteva di lasciar lavorare PassGAN per tutto il tempo necessario: tendenzialmente dopo 4/5 ore venivo espulso dalla sessione con conseguente perdita dei dati raccolti fino a quell'istante. Così ho deciso di far processare la rete finchè potevo in entrambe le fasi (training e guessing), arrivando ad iterare il modello di addestramento per **35 mila** iterazioni e a far generare password al modulo di password guessing per circa **4/5 ore** per ogni dizionario. Come si evidenzia nelle figure 4.23 e 4.24 i tempi impiegati dalla rete per ogni dizionario sono proibitivi; se si considera che sarebbero state necessarie **199 mila** iterazioni di train e tra le **800 milioni** ed il **miliardo** di password generate (per poter confrontare a dovere questo modello con il PCFG-Cracker) le ore occorrenti a PassGAN sarebbero state notevolmente di più. Il numero di password generate è in figura 4.25: è evidente come PassGAN non sia arrivato all'obiettivo minimo di 800 milioni di password create.

```
Time.Started.....: Thu Sep  9 14:31:40 2021 (3 mins, 12 secs)
Time.Estimated...: Thu Sep  9 14:34:52 2021 (0 secs)
Guess.Base.......: File (PASSGAN_generated_pass_200kpsw.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  2738.8 kH/s (0.66ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered........: 7615/48410 (15.73%) Digests
Remaining........: 40795 (84.27%) Digests
Recovered/Time...: CUR:551,N/A,N/A AVG:2372,142359,3416629 (Min,Hour,Day)
Progress.........: 476161024/476161024 (100.00%)
```

Figura 4.26: In figura il numero di password craccate dal PassGAN in seguito ai dati di input **200kClixSense**.

```
Time.Started.....: Sun Sep 19 15:38:58 2021 (4 mins, 3 secs)
Time.Estimated...: Sun Sep 19 15:43:01 2021 (0 secs)
Guess.Base.......: File (PassGAN_generated_pass_200k_rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  3008.2 kH/s (0.62ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered........: 5987/48410 (12.37%) Digests
Remaining........: 42423 (87.63%) Digests
Recovered/Time...: CUR:429,N/A,N/A AVG:1476,88575,2125818 (Min,Hour,Day)
Progress.........: 642049029/642049029 (100.00%)
```

Figura 4.27: In figura il numero di password craccate dal PassGAN in seguito ai dati di input **rockyou_200k**.

```
86  Time.Started.....: Thu Sep  9 14:49:40 2021 (2 mins, 8 secs)
87  Time.Estimated...: Thu Sep  9 14:51:48 2021 (0 secs)
88  Guess.Base.......: File (PassGAN_generated_pass_200kpsw_rockyou_PI50k.txt)
89  Guess.Queue......: 1/1 (100.00%)
90  Speed.#1.........:  2762.6 kH/s (0.82ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
91  Recovered........: 3307/48410 (6.83%) Digests
92  Remaining........: 45103 (93.17%) Digests
93  Recovered/Time...: CUR:533,N/A,N/A AVG:1545,92755,2226127 (Min,Hour,Day)
94  Progress.........: 376833037/376833037 (100.00%)
```

Figura 4.28: In figura il numero di password craccate dal PassGAN in seguito ai dati di input **rockyou_200k_pi50k**.

Hashcat è stato in grado di craccare, con i tre dizionari generati da PassGAN e basati su **200kClixSense**, **rockyou_200k** e **rockyou_200k_pi50k**, rispettivamente il **15.73%**, **12.37%** e **6.83%** di password target. I tre test hanno un coefficente di password corrette su password create rispettivamente di **0.00001599**, **0,00000932** e **0,00000877**: in pratica sono una password corretta ogni **62 529**, **107 290** e **113 950**. Si capisce chiaramente (figure 4.26, 4.27 e 4.28) come PassGAN non abbia dato buoni risultati. Ma ha mostrato un'insolita peculiarità: il numero di password corrette è più alto se si usano le sole 200 mila password di ClixSense. Questo mi fa capire come la rete neurale lavori molto meglio solo con password "sorelle" o che provengano dallo stesso set di dati delle password target. Alla luce di questi risultati così scarsi e a causa dei tempi cosi estremi (proporzionali all'aumentare della dimensione del file di input) ho deciso di non testare questo programma sul quarto ed ultimo dizionario di train (rockyou_200k_pi50k_CPGgenerated).

- **PRINCEProcessor**.

  Il terzo delle cinque prove che ho fatto riguarda il PRINCEProcessor. L'ordine di test sui dizionari è quello convenzionalmente usato finora.

```
549 Speed.#1.........:   992.5 kH/s (0.69ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
550 Recovered........: 4056/48410 (8.38%) Digests
551 Remaining........: 44354 (91.62%) Digests
552 Recovered/Time...: CUR:0,N/A,N/A AVG:67,4055,97326 (Min,Hour,Day)
553 Progress.........: 3157577728
```

Figura 4.29: In figura il numero di password craccate dal PRINCEProcessor in seguito ai dati di input **200kClixSense**.

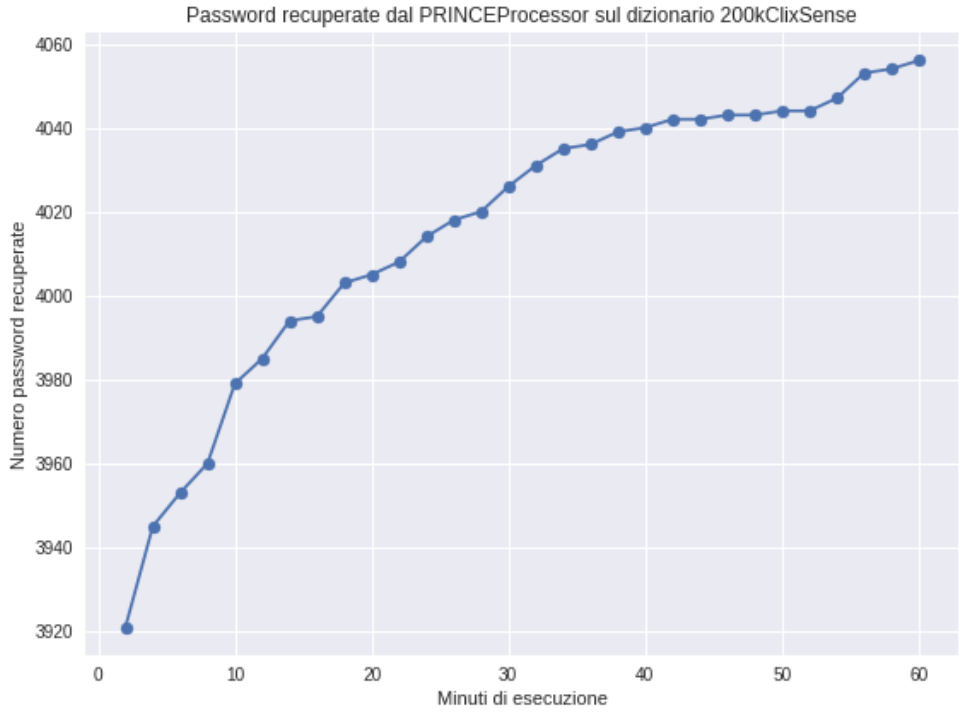


Figura 4.30: In figura il numero di password craccate dal PRINCEProcessor in seguito ai dati di input **200kClixSense** rispetto ai minuti di esecuzione.

Sul dizionario **200kClixSense** PRINCEProcessor genera blandi risultati. Sebbene la velocità di hashing delle password sia alta (vuol dire anche che l'input viene generato rapidamente da PRINCE) le password recuperate dopo un'ora sono a malapena l'8%, in particolare l'**8.38%** (figura 4.29). Genera una password corretta ogni **954 816** tentate (rapporto del **0.00000128**): è un dato leggermente falsato dalla maggiore velocità di creazione password del PRINCEProcessor.

```
Speed.#1.........:   966.1 kH/s (0.72ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered........: 19743/48410 (40.78%) Digests
Remaining........: 28667 (59.22%) Digests
Recovered/Time...: CUR:9,N/A,N/A AVG:329,19743,473843 (Min,Hour,Day)
Progress.........: 3443154944
```

Figura 4.31: In figura il numero di password craccate dal PRINCEProcessor in seguito ai dati di input **rockyou_200k**.

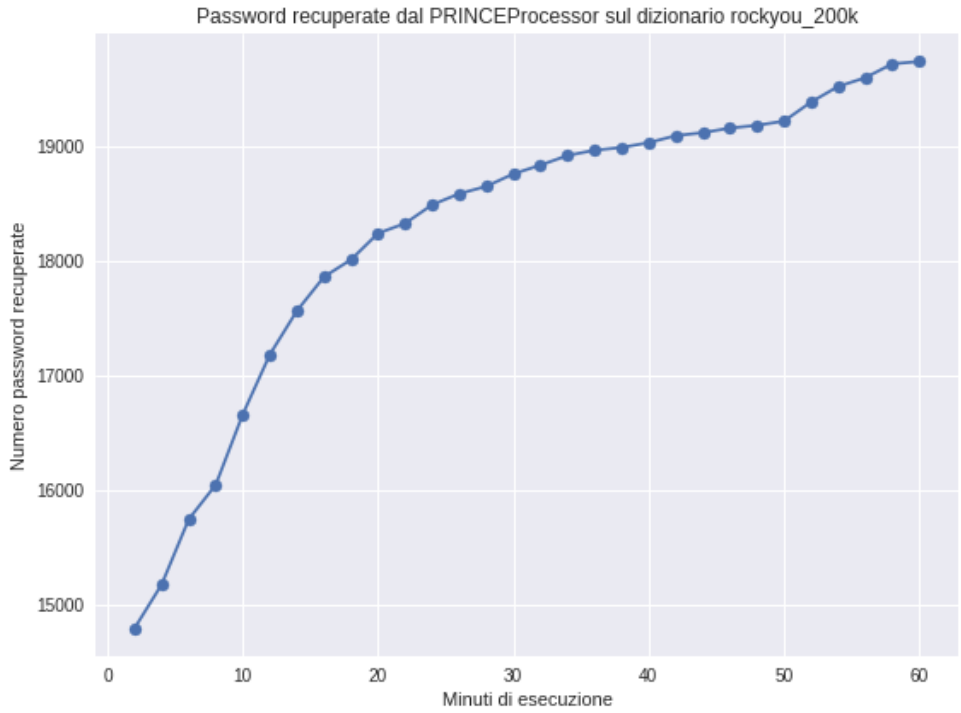


Figura 4.32: In figura il numero di password craccate dal PRINCEProcessor in seguito ai dati di input **rockyou_200k** rispetto ai minuti di esecuzione.

Durante la seconda analisi del PRINCEProcessor è stato adoperato il dizionario **rockyou_200k**. Questo test ha portato risultati decisamente migliori rispetto allo scorso. Le password recuperate dopo un'ora sono il **40.76%** del totale (figura 4.31). La generazione di password corrette avviene ancora ad un tasso molto alto (a causa sempre della velocità di hashing): una password corretta ogni **174 398** tentate con un rapporto quindi del valore di **0.00000573**. Infine si noti come la curva (figura 4.32) tenda a stabilizzarsi dopo un'ora ma senza mai appiattirsi del tutto: questo prova la generazione pressochè infinita di password da parte del PRINCEProcessor, come si vedeva in [30].

```
549 Speed.#1.........:   919.1 kH/s (0.65ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
550 Recovered........: 19719/48410 (40.73%) Digests
551 Remaining........: 28691 (59.27%) Digests
552 Recovered/Time...: CUR:13,N/A,N/A AVG:328,19718,473252 (Min,Hour,Day)
553 Progress.........: 3318013952
```

Figura 4.33: In figura il numero di password craccate dal PRINCEProcessor in seguito ai dati di input **rockyou_200k_pi50k**.

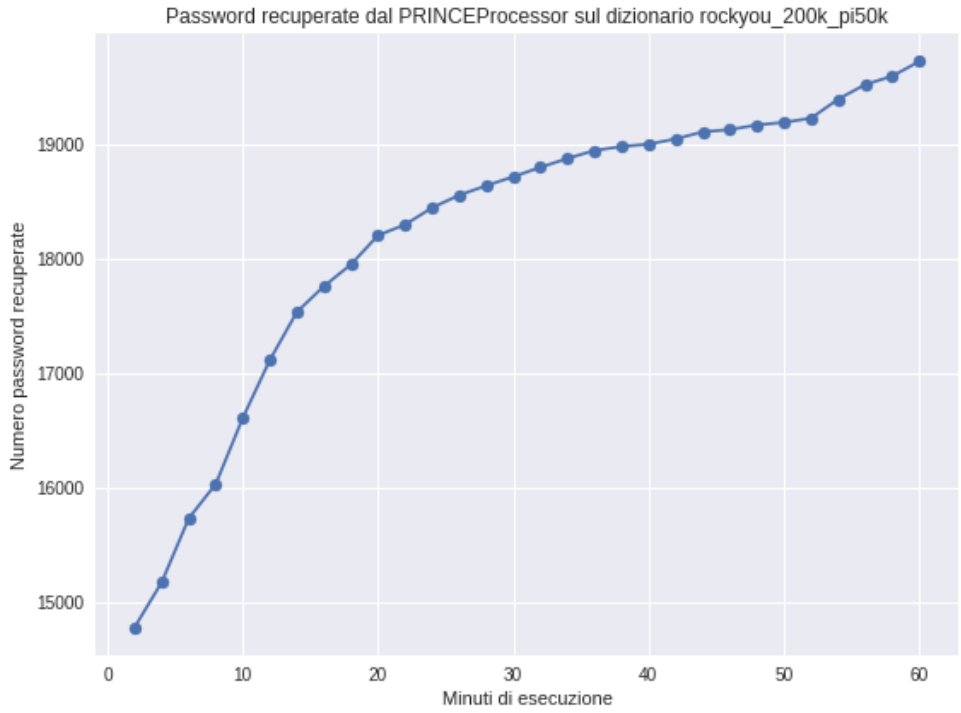


Figura 4.34: In figura il numero di password craccate dal PRINCEProcessor in seguito ai dati di input **rockyou_200k_pi50k** rispetto ai minuti di esecuzione.

Per l'utlima prova del PRINCEProcessor è stato adoperato il dizionario **rockyou_200k_pi50k** che ha fruttato risultati simili al test precedente. Questo potrebbe far supporre la scarsa abilità nell'adoperare dati personali nel dizionario di input da parte del PRINCEProcessor. Le password recuperate dal test sono il **40.73%** del totale (figura 4.33). La relazione tra password corrette e password provate si aggira attorno allo **0.00000594** che corrisponde a **168 264** tentate ogni corretta. Tutte e tre le curve di risultati con il PRINCEProcessor (vedi figura 4.34, 4.32, 4.30) hanno una "gobba" tipica circa a metà sessione: è lì che lo strumento iniza a perdere potenza e per assicurarsi buoni risultati andrebbe lasciato processare ancora a lungo.

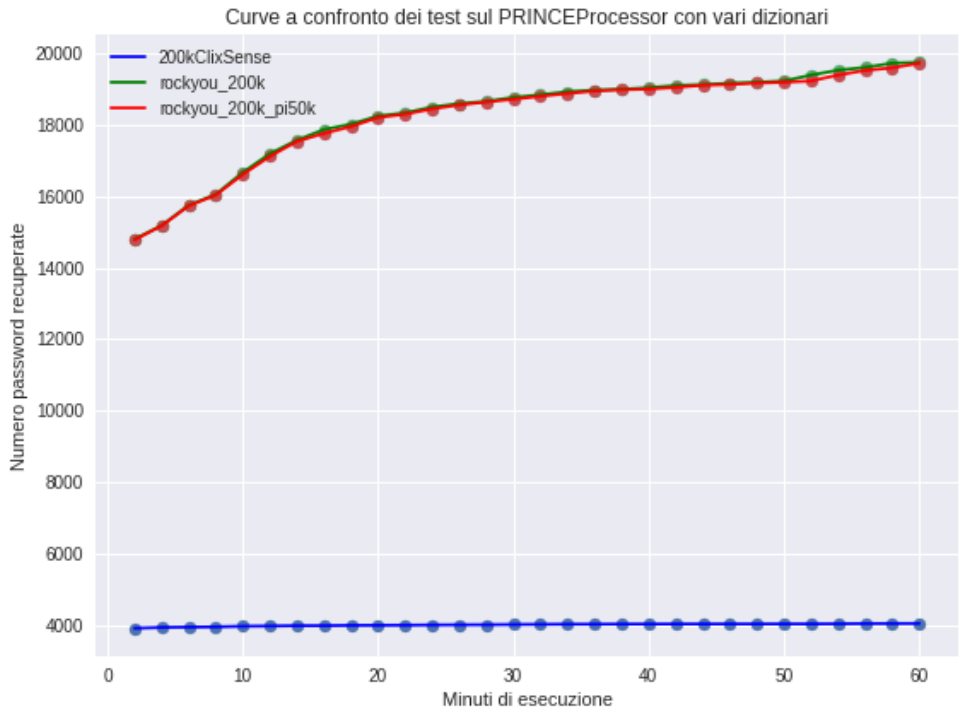


Figura 4.35: Il grafico mostra l'andamento del recupero password del PRINCEProcessor in base al dizionario in input.

Confrontiamo infine le tre curve dei tre test: il primo test, a confronto con gli altri due, è pressochè stabile nella ricerca di nuove password. La seconda e terza curva dei rispettivi test invece si assomigliano visibilmente (figura 4.35).

- **CPG | PCFG-Cracker**$_{OMEN}$.
  In questo test verrà utilizzato il dizionario contenente anche le password create con il mio tool CPG: il **rockyou_200k_pi50k_CPGgenerated**. Se si analizzasse solamente il numero di password craccate questo sarebbe esiguo rispetto agli altri risultati ottenuti con il tool basato sui PCFG: la verifica ha permesso di recuperare solo il **21.68%** di password. Estendendo l'analisi anche alle altre metriche di giudizio ci rendiamo conto però che il rapporto tra password corrette e tentate è del **0.0000863**, generando così una password buona ogni **11 579**: valori al di sopra di quelli visti finora. Va evidenziato però come la velocità di hashing sia crollata. Il modello PCFG essendo stato allenato su un dizionario così grande ha generato un ruleset altrettanto grande: questo fatto comporta una notevole riduzione della velocità in fase generativa. La velocità di hashing si basa sul numero di password in ingresso: con poche password la velocità diminuisce. Per ottenere risultati più in linea con gli altri sarebbe stato necessario prolungare il test per più ore. Le figure 4.36 e 4.37 riportano graficamente i risultati.

```
549 Speed.#1.........:    26329 H/s (2.68ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
550 Recovered........: 10495/48410 (21.68%) Digests
551 Remaining........: 37915 (78.32%) Digests
552 Recovered/Time...: CUR:18,N/A,N/A AVG:174,10497,251940 (Min,Hour,Day)
553 Progress.........: 121528335
```

Figura 4.36: La figura mostra le password recuperate dal PCFG-Cracker$_{OMEN}$ con il dizionario generato dal CPG mescolato con il rockyou_200k_pi50k, denominato **rockyou_200k_pi50k_CPGgenerated**.

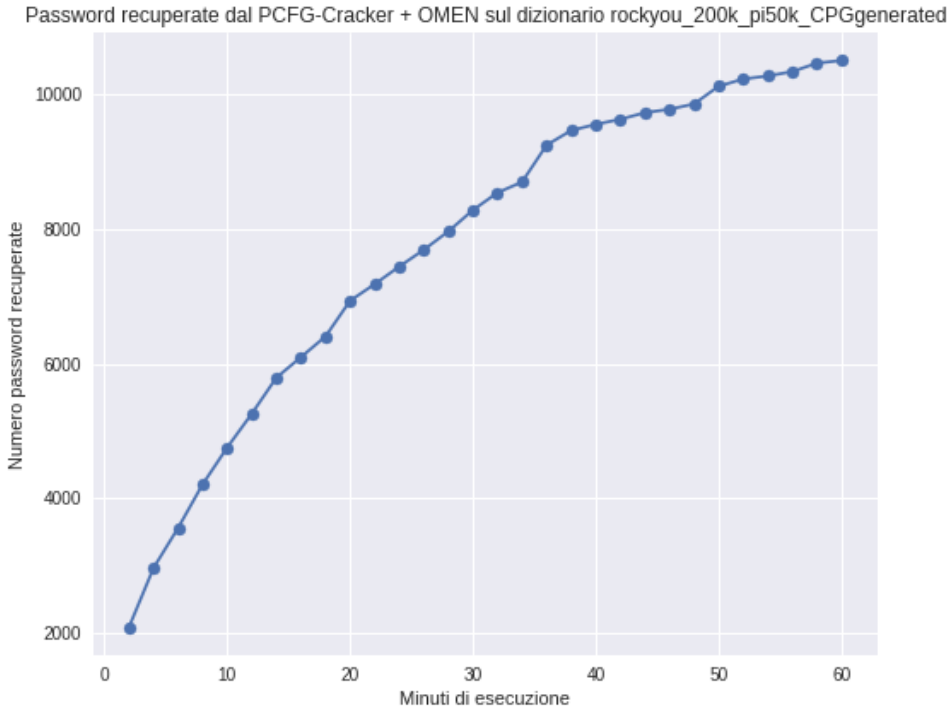


Figura 4.37: Il grafico mostra l'andamento del recupero password del PCFG-Cracker$_{OMEN}$ in base al dizionario generato dal CPG mescolato con il rockyou_200k_pi50k, denominato **rockyou_200k_pi50k_CPGgenerated**.

Dato che questo è l'ultimo test effettuato con il PCFG-Cracker confronto tutti i risultati ottenuti in un unico grafico (vedi figura 4.38).

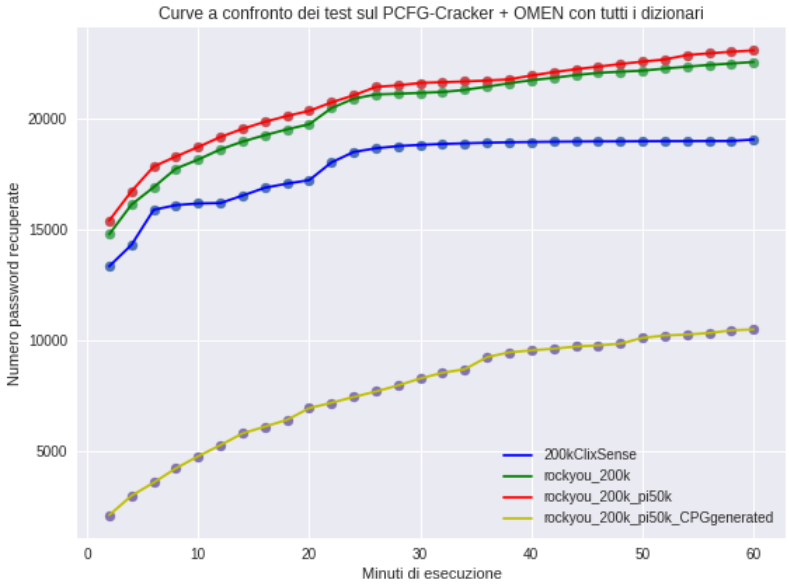


Figura 4.38: Il grafico confronta l'andamento del recupero password del PCFG-Cracker$_{OMEN}$ in base a tutti e quattro i dizionari di training.

- **CPG | PRINCEProcessor**.
  L'ultimo test eseguito in questo documento riguarda il dizionario **rockyou_200k_pi50k_CPGgenerated** messo in input al PRINCEProcessor. I risultati sono in linea con il secondo e il terzo test effettuati con questo tool; il primo esito segna **39.85%** di password craccate, il secondo è il rapporto tra password buone e quelle generate che è dello **0.0000061** (equivalente ad una password valida ogni **162 467** generate). Questo risultato evidenzia ancora una volta come i dati personali (o meglio ancora le password generate su dati personali) non sempre siano adatti per un PRINCE attack (figure 4.39 e 4.40). Infine confrontiamo tutti i PRINCE attack portati finora (figura 4.41).

```
550 Speed.#1.........:   930.4 kH/s (0.64ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
551 Recovered........: 19291/48410 (39.85%) Digests
552 Remaining........: 29119 (60.15%) Digests
553 Recovered/Time...: CUR:45,N/A,N/A AVG:321,19290,462972 (Min,Hour,Day)
554 Progress.........: 3134152704
```

Figura 4.39: La figura mostra le password recuperate dal PRINCEProcessor con il dizionario generato dal CPG mescolato con il rockyou_200k_pi50k, denominato **rockyou_200k_pi50k_CPGgenerated**.

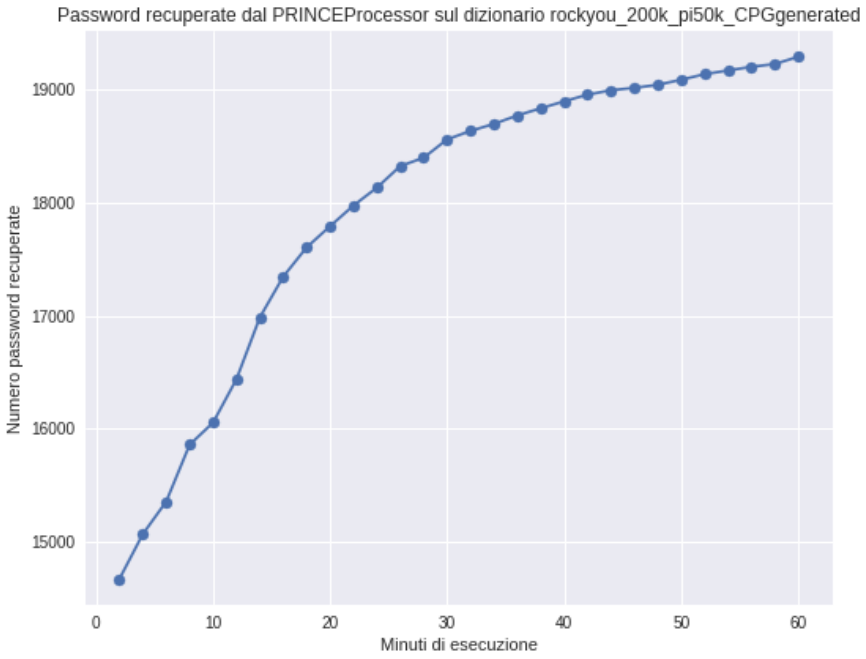


Figura 4.40: Il grafico mostra l'andamento del recupero password del PRINCEProcessor in base al dizionario generato dal CPG mescolato con il rockyou_200k_pi50k, denominato **rockyou_200k_pi50k_CPGgenerated**.

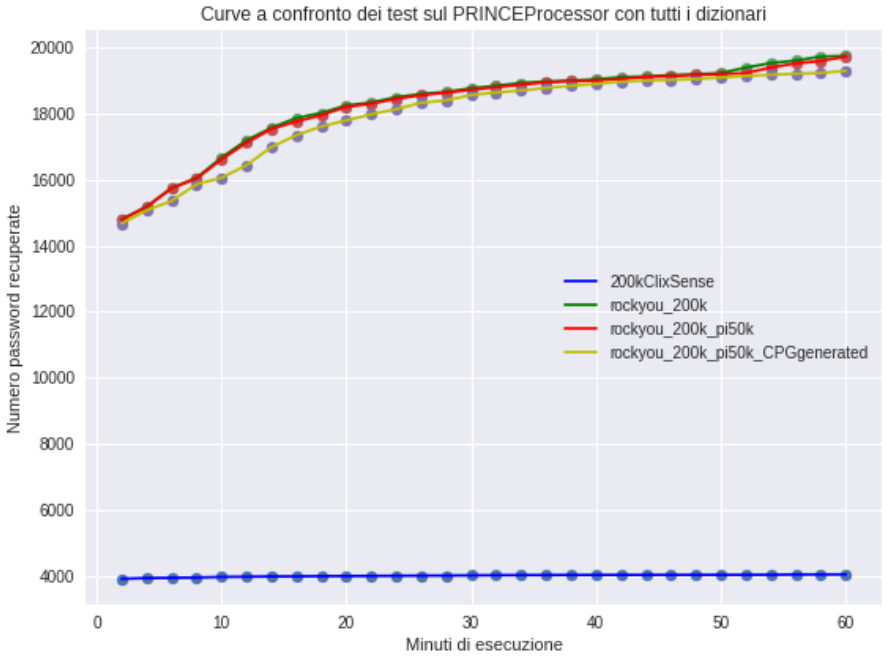


Figura 4.41: Il grafico confronta l'andamento del recupero password del PRINCEProcessor in base a tutti e quattro i dizionari di training.

Capitolo 5

Conclusioni

5.1 Obiettivi raggiunti

Nel corso di questo progetto sono stati testati strumenti allo stato dell'arte del password cracking in situazioni classiche e nuove rispetto a quanto visto nei paper citati. Oltre ai test sui singoli tool ho sviluppato un software che, date delle parole personali in input e attraverso permutazioni e altre modifiche, genera possibili password legate ad una persona target.

I test che ho affrontato sono stati realizzati in situazioni il più reale possibile: ciò che farebbe un soggetto che entra in possesso di un database di password criptate. Sono stati provati vari dizionari per capire quale effettivemente fosse il migliore e quanto incidano le informazioni personali nell'ambito del password cracking. Dai test effettuati si è dedotto che uno strumento in grado di sfruttare al meglio (ovvero in maniera efficiente e scalabile) le informazioni personali di una persona target *non esiste*. Esistono però strumenti, tra quelli analizzati, in grado di utilizzare questi dati in maniera più o meno efficace.

Il *PCFG-Cracker* con all'interno un'implementazione di *OMEN* è quello che nel complesso ha dato migliori risultati riuscendo a sfruttare bene anche i dati sensibili delle persone target. I test su *PassGAN* hanno rivelato come questo sia inutilizzabile per una persona che abbia a disposizione risorse "comuni" (ovvero che disponga di meno di 16 GB di RAM e non possieda una o più GPU dedicate): in questo caso i tempi per allenare il modello e poi fargli generare le password sono tremendamente lunghi (giornate intere di allenamento). Si consideri infatti che PassGAN non permette di allenarlo solo un parte di un dataset in input se questo contiene password su cui si è già addestrato: infatti ogni train-set genererà una serie di regole univoche solo per quel set di dati. Inoltre ha evidenziato la scarsa predisposizione ad usare informazioni personali, ma anche le password con struttura diverse da quelle target. La predizione di password corrette poteva essere decisamente più buona per PassGAN se fosse stato possibile allenarlo per un numero di iterazioni consono; essendo inoltre

lo strumento più recente e sofisticato tra quelli testati le mie aspettative erano alte. Il *PRINCEProcessor* ha dimostrato come possa essere un valido strumento nel mondo del password cracking. I meriti di tale validità vanno alla facilità d'uso ed al numero di password che, grazie al runtime infinito, potrebbe craccare. L'accuratezza delle password create dal PRINCE diminuisce (seppur di poco) una volta introdotte in input informazioni personali: segnale chiaro del fatto che il PRINCEProcessor non gestisca in maniera limpida le informazioni personali del target. Infine i risultati sul tool da me sviluppato *CPG* sono discretamente buoni. CPG ha messo in evidenza come esso possa essere un valido strumento da tenere in considerazione per una sessione di password cracking: con determinate condizioni può portare a numerosi benefici. I risultati per questo tool sono incoraggianti, considerando che nei test da me condotti non si utilizza una versione completa bensì una ridotta. E' evidente però come i margini di miglioramento siano ampi.

L'obiettivo di questa tesi è sviluppare e analizzare tecniche innovative nel contesto del password cracking targettizzato. Nella stesura della tesi, partendo da una base teorica, ho deciso di adottare un approccio pragmatico alla risoluzione dei problemi incontrati cercando sempre di verificare e testare quanto studiato nella documentazione di riferimento. Considerando la mancanza di tutorial, o documenti che spiegassero come utilizzare gli strumenti da me impiegati, posso dire, in conclusione, di ritenermi ampiamente soddisfatto sia dei risultati ottenuti sia dal tool creato. Se ci si trovasse in una situazione di password cracking (targettizzato o meno) in cui non si disponesse nè di molto tempo nè della conoscenza dell'hardware a disposizione, consiglierei un attacco basato sul **PRINCEProcessor** per via della sua velocità e facilità d'impiego. Lo strumento che nel complesso ha dato risultati migliori, come si vede dalla figura 5.1, utilizzando il dizionario di input **rockyou_200k_pi50k**, è stato il **PCFG-Cracker** con l'aiuto di **OMEN**. Questo conferma come i dati sensibili delle persone siano utilizzati nella creazione di una password, quindi possano tornare utili per un attacco ad esse.

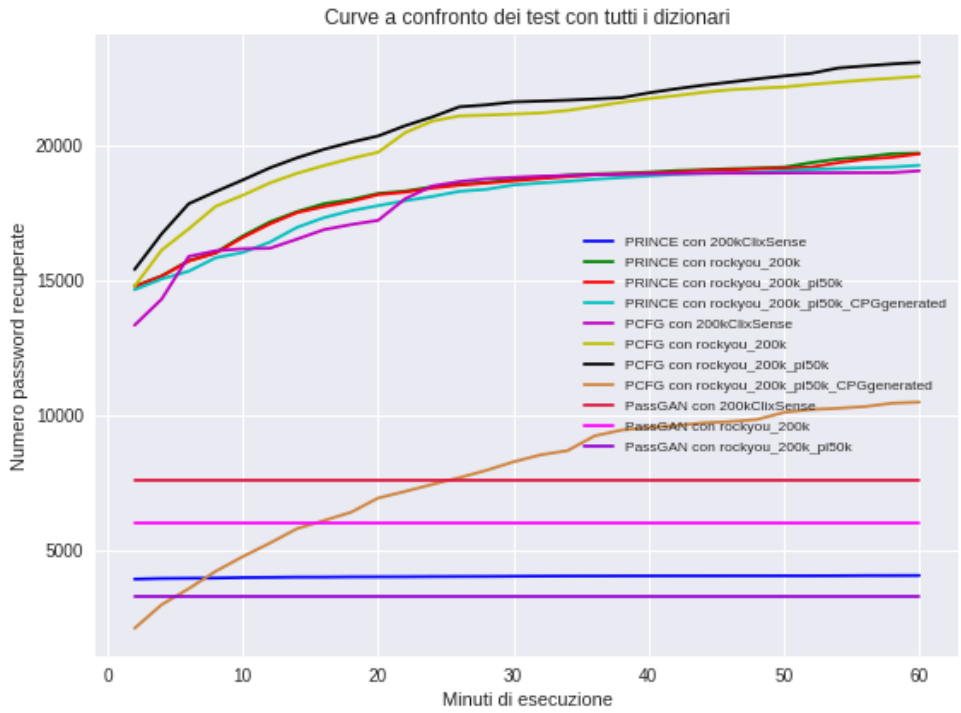


Figura 5.1: Il grafico confronta l'andamento del recupero password di tutti i test fatti.

5.2 Sviluppi futuri

I miglioramenti che si potrebbero apportare ai test ed al programma Python da me creato, sarebbero molteplici; infatti le ricerche nell'ambito del password cracking che si focalizzano maggiormente su un tipo di attacco che sfrutta anche informazioni personali sono pochi.

Cito ora alcuni sviluppi futuri al mio progetto:

- Sarebbe interessante fare uno studio sul dizionario, ovvero capire l'importanza dei termini che contiene, assegnando loro un "peso", per poi ad esempio duplicare o triplicare le parole più importanti. Un seconda fase di training potrebbe portare a migliori risultati sapendo quali parole vengono più usate.

- Per tutti i tool studiati sarebbe interessante svolgere un attacco a poche singole password (non banali): prima utilizzando solo password sorelle o generiche e poi inserendo anche informazioni extra relative solo a quel target. In pratica rifacendo i test da me attuati ma con il secondo protocollo di testing individuato da me e Marco (spiegato all'inizio della sezione 4.2).

- Studiare l'effettiva efficacia di OMEN se preso singolarmente (non come nel mio caso in cui è implementato nel PCFG-Cracker).

- Sperimentare rules di ottimizzazione per capire quale di queste porti maggiori benefici ad una sessione di password cracking targettizzata.

- Per capire nello specifico quali password vengono scoperte da un dato strumento si potrebbero ricondurre i test con il *potfile* di Hashcat attivo. Usando anche il potfile avrei l'opportunità di capire quale test mi porti ad un maggior beneficio (quali parole sono più importanti e quali regole di generazione sono più importanti).

- Sviluppi futuri di CPG potrebbero:

  - migliorarne le prestazioni (ad esempio non generando password troppo lunghe o considerando quelle già create)
  - fare in modo che la generazione avvenga seguendo per lo più la distribuzione delle password target (se si hanno ovviamente password in chiaro dallo stesso database)
  - fare in modo che generi password utilizzando anche il concetto dei vezzeggiativi, nomignoli, diminuitivi e accrescitivi: molte password, per lo più italiane, usano queste deviazioni morfologiche della parola.

- Sviluppare un ulteriore tool basato totalmente sul Machine Learning che riesca a sfruttare al meglio le informazioni personali contenute nelle password. Se a tale tool venissero date in ingresso password simili e informazioni personali relative ad esse, il modello dovrebbe , in fase di allenamento, scoprire pattern specifici relativi ai dati personali (ad esempio come vengono usati i dati in quel pool di password di training). Una volta addestrato il modello, l'idea sarebbe quella di fornirgli in input le informazioni sensibili di uno o più target in modo che generi possibili password su tali dati personali.

Capitolo 6

Ringraziamenti

Ringrazio innanzitutto Marco Canducci, del team di Cyberloop, azienda affiliata a Clipperz tramite il gruppo Imola Informatica presso la quale ho svolto i miei tirocini. Marco nel corso del progetto e della tesi mi ha sempre affiancato e aiutato, donandomi costanti spunti e opinioni sul lavoro: fondamentali in un ambito da me mai percorso. Vorrei ringraziare anche la professoressa Lumini Alessandra che mi ha permesso di redarre questa tesi con lei, supportandomi e donandomi idee regolarmente nella stesura. E' doveroso poi ringraziare la mia famiglia: mia madre Marina, mio papà Carlo, mia sorella Giorgia e la mia morosa Sofia. Tutti questi mi hanno sostenuto in questi ultimi mesi di laurea triennale, periodo molto complicato. Un ringraziamento anche ai miei amici e compagni di studio/progetti del MASP: Giacomo, Nicola, Albi e Nikolas; questo percorso di studi non sarebbe stato lo stesso senza di loro. Un'ultima dedica, ma non per importanza, al mio gruppo ELTA: amici storici che mi hanno sopportato e tirato su il morale in questi tre anni di alti e bassi.

Bibliografia

[1] Sudhir Aggarwal, Shiva Houshmand, and Matt Weir. New technologies in password cracking techniques. In *Cyber Security: Power and Technology*, pages 179–198. Springer, 2018.

[2] Edward B. et al. Cupp github main page, 2020. Available on line.

[3] Daniel V Bailey, Markus Dürmuth, and Christof Paar. Statistics on password re-use and adaptive strength for financial accounts. In *International Conference on Security and Cryptography for Networks*, pages 218–235. Springer, 2014.

[4] Isaac Bennetch et al. phpmyadmin website, 2021. Available on line.

[5] Ruhr University Bochum. Omen github main page, 2017. Available on line.

[6] Claude Castelluccia, Abdelberi Chaabane, Markus Dürmuth, and Daniele Perito. When privacy meets security: Leveraging personal information for password cracking. *arXiv preprint arXiv:1304.6584*, 2013.

[7] Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Adaptive password-strength meters from markov models. In *NDSS*, 2012.

[8] Presidenza del Consiglio dei Ministri. Glossario intelligence - il linguaggio degli organismi informativi, 2019. Available on line.

[9] Digital360 et al. Cybersecurity360 apt page, 2021. Available on line.

[10] Google. Colab home page, 2021. Available on line.

[11] Google. Google course on gan, 2021. Available on line.

[12] Wendy Goucher. Look behind you: the dangers of shoulder surfing. *Computer Fraud & Security*, 2011(11):17–20, 2011.

[13] Ishaan Gulrajani. Iwgan github main page, 2017. Available on line.

[14] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. *arXiv preprint arXiv:1704.00028*, 2017.

[15] Raymond Hettinger et al. Itertools python main page, 2021. Available on line.

[16] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. Passgan: A deep learning approach for password guessing. In *International Conference on Applied Cryptography and Network Security*, pages 217–237. Springer, 2019.

[17] Shiva Houshmand and Sudhir Aggarwal. Using personal information in targeted grammar-based probabilistic password attacks. In *IFIP International Conference on Digital Forensics*, pages 285–303. Springer, 2017.

[18] Daichi K. et al. Passgan updated github repository, 2019. Available on line.

[19] Peter Kacherginsky et al. Pack github home page, 2019. Available on line.

[20] Margara Luciano. Firma digitale - crittografia, 2021.

[21] OpenWall. John the ripper main page, 2021. Available on line.

[22] Ruhr-University Bochum Mobile Security Section. Nemo github main page, 2019. Available on line.

[23] Kai 'Oswald' Seidler et al. xampp website, 2021. Available on line.

[24] Aditya K Sood and Richard J Enbody. Targeted cyberattacks: a superset of advanced persistent threats. *IEEE security & privacy*, 11(1):54–61, 2012.

[25] Jens Steube et al. Hashcat mask attack page, 2015. Available on line.

[26] Jens Steube et al. Princeprocessor simple tutorial on hashcat's forum, 2015. Available on line.

[27] Jens Steube et al. Hashcat optimized kernel, 2017. Available on line.

[28] Jens Steube et al. Prince optimized rules, 2017. Available on line.

[29] Jens Steube et al. Princeprocessor main page, 2019. Available on line.

[30] Jens Steube et al. Princeprocessor example on hashcat forum, 2020. Available on line.

[31] Jens Steube et al. Hashcat home page, 2021. Available on line.

[32] Jens Steube et al. Hashcat rule based attack page, 2021. Available on line.

[33] Red Hat team. Redhat malware page, 2021. Available on line.

[34] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. " i added'!'at the end to make it secure": Observing password creation in the lab. In *Eleventh Symposium On Usable Privacy and Security ({SOUPS} 2015)*, pages 123–140, 2015.

[35] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1242–1254, 2016.

[36] Charles Matthew Weir. *Using probabilistic techniques to aid in password cracking attacks*. The Florida State University, 2010.

[37] Matt Weir. Unlock download page, 2009. Available on line.

[38] Matt Weir. Omen python github main page, 2018. Available on line.

[39] Matt Weir. Pcfg-guesser compiled c github main page, 2019. Available on line.

[40] Matt Weir, Sudhir Aggarwal, Breno De Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE Symposium on Security and Privacy*, pages 391–405. IEEE, 2009.

[41] Matt Weir et al. Pcfg-cracker github repository, 2020. Available on line.