UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea
in Ingegneria Informatica

Tesi di Laurea

Honey Encryption
per messaggi in Linguaggio Naturale

Relatore: Chiar.mo Prof. Luca Giuzzi

Laureando:
Yanez Diego Parolin

Matricola n. 715432

Anno Accademico 2018/2019

A chi utilizza sempre la stessa password

"Treat your password like your toothbrush. Don't let anybody else use it, and get a new one every six months"

Clifford Stoll

Sommario

Generalmente, quando si prova ad accedere a una risorsa, protetta da password, utilizzando una chiave errata, otterremmo una risposta chiaramente negativa: un accesso negato, un risultato incomprensbile ecc.

Chiaramente in virtù di questa circostanza, un eventuale utente malevole, potrebbe proseguire a tentativi, fintantoché non verrà trovata una password che restituirà un esito positivo. Questa tipologia di attacco viene comunemente definita attacco a forza bruta, o in inglese "brute force attack". In questa tesi verrà presentata l'Honey Encryption, una recente tecnica crittografica presentata all'EuroCrypt 2014 [1] volta a risolvere questo problema. Alcune sue applicazioni e, nel dettaglio, la mia personale implementazione applicata al Linguaggio Naturale, in particolare alla codifica e decodifica di semplici messaggi in lingua inglese.

Utilizzando degli strumenti di analisi del testo e di natural language processing, liberamente disponibili e accessibili sia a scopo didattico che commerciale, come spaCy, libreria Python per l'analisi grammaticale, i messaggi da criptare, verranno destrutturati in stringhe di caratteri alfanumerici. E successivamente, in fase di decriptazione, decodificati, ricostruendo il messaggio originario, se la password dovesse essere corretta, altrimenti in testi alternativi, ma plausibili, che un utente malevolo non potrebbe automaticamente riconoscere come falsi. Il grosso problema, sia del mio progetto che delle altre implementazioni basate sul linguaggio naturale, è quello di riuscire a esprimere dei dati flessibili, come le frasi, attraverso delle rigide regole di struttura, al contrario di quanto accade con dati "di tipo semplice" come i numeri di carte di credito o i codici fiscali, altamente strutturati, dove la loro costruzione è vincolata da forti e precise regole logico-matematiche.

Indice

Elenco delle figure

Elenco delle tabelle

Listings

Capitolo 1

Introduzione

1.1 Presentazione

Attraverso articoli, blog e talk si è sentito innumerevoli volte parlare di tematiche legate alla sicurezza delle password, di quanto sia importante non solo di evitare l'utilizzo della medesima password, ma anche di cambiarla periodicamente, di strutturarla con precise regole e infine di utilizzare password manager, cioè dei servizi che permettono di memorizzare e di generare password complesse (spesso offrono la piena compatibilità con i nostri sistemi operativi, desktop o mobile, e le applicazioni più usate, come Chrome, Firefox . . .).

Eppure, sia per pigrizia che per abitudine, finiamo per utilizzare sempre le solite chiavi d'accesso, o password facili, composte da parole a noi vicine (data di nascita, soprannome ...), oppure una molto complessa, con caratteri speciali, numeri e maiuscole, ma corte e con sostituzioni prevedibili (a con 4 o @, i con 1 o ! . . .).
Così facendo, stiamo semplicemente esponendo indirettamente tutti i nostri dati sensibili, volenti o nolenti, rendendoli disponibili e facilmente recuperabili da malintenzionati, che, utilizzando tecniche di *brute force attack*, una tecnica che prevede di procedere con numerosi tentativi per scoprire la password, con il sostegno di dictionary, liste di parole correlate a noi (che potrebbero comporre parzialmente la nostra chiave).
Inoltre, gli stessi siti web che siamo soliti utilizzare, purtroppo sono esposti e vulnerabili, tanto ché spesso avvengono leak dei dati privati degli utenti (codice fiscale, nome, data di nascita, password) che, trapelando, vanno anche ad alimentare questi famosi dizionari, andando a favorire un'attacco con l'obiettivo di violare la nostra sicurezza da parte di un un hacker.

1.1.1 Il progetto in breve

Il lavoro che verrà presentato in questo documento, si basa su una nuova tecnica crittografia denominata **Honey Encryption** [1], e spiegata nel dettaglio in seguito, che ha l'obiettivo di rendere vani attacchi a forza bruta utilizzando delle *decoy*, cioè dei risultati plausibili ottenuti anche quando la password inserita risulta errata.
Nel dettaglio, il mio intento è quello di proteggere le comunicazioni e i messaggi inviati, generando una messaggio grammaticalmente corretto anche quando un eventuale utente malevolo intercetti la comunicazione e provi a decriptarla con una chiave sbagliata, rendendo inutilmente costoso e laborioso il tentativo di brute force attack, visto che andrebbe analizzata ogni singola decoy.

Le potenzialità di questa applicazione dell'Honey Encryption al linguaggio naturale sono evidenti, il problema, come vedremo, sarà l'effettiva implementazione, soprattutto nella sfera del linguaggio naturale.

1.2 Accenno storico sulla crittografia

La crittografia accompagna l'uomo da millenni, le quali origini, sebbene si tratti più di un archetipo piuttosto che di una vera tecnica, risalgono addirittura all'Antico Regno d'Egitto, con il primo archetipo di cifrario a sostituzione basato sui geroglifici; Anche in Mesopotamia e in India sono stati rinvenuti esempi di utilizzo di basilari sistemi crittografici, per esempio procedendo con la sostituzione della parte finale delle parole con suffissi stereotipati [2].

Bisogna però attendere i greci con l'utilizzo della *scitala* (un cifrario a trasposizione) e il popolo d'Israele con il *cifrario di Atbash*, per avere le prime vere forme strutturate di crittografia, applicate a più ambiti della società, per fini militari, diplomatici e commerciali.

Ovviamente non si può non ricordare il noto Cifrario di Cesare e i successivi studi di crittoanalisi e crittologia avvenuti in epoca medievale e rinascimentale da parte di numerosi matematici arabi, come al-Kindi tramite i suoi manoscritti, ed europei, in primis Leon Battista Alberti che sviluppò il *disco cifrante*.

Per poi giungere a un anticipo della crittografia moderna, dove i calcolatori ne fanno da padroni, con Claude Shannon e i suo *Communication Theory of Secrecy Systems* [3], una raccolta di appunti e studi crittografici.

1.3 Il binomio Uomo-Sicurezza Crittografica

La storia della crittografia è avvolta su se stessa due due aspetti in perpetuo conflitto-equilibrio, che ricordano lo yin e yang della filosofia cinese, la criptazione e la decrittazione. Due elementi che non possono esistere senza l'altro, il primo rappresenta lo

studio dietro allo sviluppo di un sistema crittografico e lo sforzo per rendere sicuro qualcosa, il secondo il processo inverso e la sua violazione.

Dopo secoli di analisi, sviluppo e studio di nuovi sistema, gli esperti sono ancora alla ricerca, spasmodicamente, di un tecnica che si possa definire perfetta, o che perlomeno, si possa avvicinarsi a tal punto da rendere inutile un tentativo di violazione: per complessità di calcolo, impossibilità o sconvenienza.
Questi aspetti sono però relativi e in continuo divenire, rendendo dunque necessaria una ricorrente rivalutazione.

C'è però un ostacolo che si discosta dall'abituale ottica di ragionamento, non dettato dai limiti tecnologici del tempo, bensì dalla fallacia dell'uomo, come ben ricordano i tedeschi con la *Macchina Enigma*.
Gli errori umani variano dall'influenzabilità e l'ingenuità di lasciare libero accesso a determinati dati, a banali disguidi oppure a errori nello sviluppo, ma uno dei più evidenti è senza ombra di dubbio quello relativo a una cattiva scelta e gestione delle password personali.

1.3.1 Il problema delle password nel XXI secolo

Soffermandosi sui giorni nostri, sebbene esistano sistemi molto elaborati e di difficile decrittazione (come SHA, OTP, alcune crittografie a chiave pubblica ...), determinate (cattive) scelte portano ad avere uno scarso livello di robustezza, rendendo dunque il sistema vulnerabile sebbene a livello teorico non lo sia.
Per esempio, in RSA, la selezione di numeri "piccoli" durante lo sviluppo della chiave può portare a esito positivo un attacco eseguito con test di primalità [4].
Inoltre esistono attacchi elaborati, partendo dall'utilizzo di dizionari e algoritmi, volti a violare messaggi criptati da funzioni crittografiche di Hash che dovrebbero essere one-way, dunque senza possibilità di dedurre il testo originale (generalmente è possibile a causa di messaggi con un entropia bassa).

Non solo, uno dei casi più eclatanti è rappresentato dalla *Password Based Encryption (PBE)* che consiste nel cifrare un elemento attraverso una password, che verrà poi (spesso) "salvata" sotto forma di hash per poi eseguire i successivi check e controlli.
Si tratta di una della tecniche più diffuse in assoluto (per proteggere database, per l'accesso a siti web o a un proprio Password Manager come KeePass, 1Password o Google Lock . . .) ma che a sua volta, similmente a quanto accade con l'hashing, la scelta delle password da parte degli utenti (e la gestione da parte dei provider del servizio) la rendono altamente insicura e vulnerabile.

Generalmente vengono utilizzate password banali o facilmente ricostruibili (data di nascità, soprannome, colore preferito ecc.), oltre ad essere utilizzata la stessa chiave per più servizi. Rendendo gli utenti vulnerabili ad attacchi brute force e dictionary e, se gli attaccanti sono riusciti a recuperare la chiave da un altro servizio,

all'accesso diretto.
Secondo i dati raccolti da Wombat e presentati nel *"User Risk 2018"* [5], più del 60% degli utenti non usa password manager e la riusa per più account, servizi, siti ecc.

I dati raccolti dagli ultimi leak

In seguito ai numerosi attacchi effettuati dal 2014 ad oggi: tra cui MySpace [6], Yahoo [7], Dropbox [8], LinkedIn [9] ... Si è raccolto un enorme database contenente password, email, username e informazioni personali. Queste raccolte di dati sono state, e lo sono tuttora, oggetto sia di studi che di fonte per attacchi hacker.
Recenti analisi su alcuni database hanno evidenziato che oltre 1% degli utenti utilizza la medesima password (123456), che l'entropia per più del 50% delle password risulta inferiore a 22 bit (circa 5 minuti il tempo per violarla) e che l'entropia minima media risulta essere minore di 7 [10].

Infatti le password più utilizzate risultano essere *"123456", "qwerty", "password"* ... e quasi il 50% sono costituite da nomi, slang o comunque parole vulnerabili a un attacco di tipo dictionary. Questo comporta che gli stessi hash, che in teoria dovrebbero univoci (anche attraverso salazione) diventano così superflui. Circa il 30% degli utenti usa password con meno di 6 caratteri [11].

Password Popularity – Top 20

Rank	Password	Number of Users with Password (absolute)
1	123456	290731
2	12345	79078
3	123456789	76790
4	Password	61958
5	iloveyou	51622
6	princess	35231
7	rockyou	22588
8	1234567	21726
9	12345678	20553
10	abc123	17542

Rank	Password	Number of Users with Password (absolute)
11	Nicole	17168
12	Daniel	16409
13	babygirl	16094
14	monkey	15294
15	Jessica	15162
16	Lovely	14950
17	michael	14898
18	Ashley	14329
19	654321	13984
20	Qwerty	13856

Figura 1.1: Top 20 Most Popular Internet Passwords
©IMPERVA

Servizi utili

Molto interessante, il servizio offerto dal sito *Have I Been Pwned?* [12], dove, inserendo la propria email, permette di scoprire se c'è stato un leak di informazioni e se alcuni nostri account e su che siti, sono stati violati.
Generalmente queste copiose raccolte di informazioni sono sparse nei meandri del Web e non accessibili a chiunque, ma tramite il sito *GhostProject* [13] (non aggiornato con i recenti leak del 2018-2019) è anche possibile avere un'anteprima delle password associate alla nostra email.

L'anello debole è rappresentato dall'utente, ciò costringe a rivalutare gli obiettivi e il punto di focalizzazione della protezione dei dati.

Capitolo 2

Honey Encryption

2.1 Introduzione

Un'ingegnosa e oltremodo interessante tecnica crittografica, è stata presentata all'*EuroCrypt del 2014*; *Ari Juel* e *Thomas Ristenpart* dell'Università del Wisconsins, presentarono un nuovo sistema di criptazione, denominato **Honey Encryption**, di seguito presentato, che potrebbe rappresentare una (parziale) soluzione al suddetto problema [1].
Sulla falsa riga del concetto di honeypot, honeyword et similia, l'obiettivo (e l'essenza) della Honey Encryption è quello di *"produrre un testo cifrato, il quale, una volta decifrato con una chiave errata, presenta comunque una testo plausibile ma ovviamente incorretto"*: nello specifico data una qualsiasi chiave (estratta dal malintenzionato attraverso conoscenze parziali, brute force attack ...) il risultato della decrittazione sia plausibile, limitando dunque la vulnerabilità intrinseca della password stessa, eliminando i risultati palesemente errati.
Dunque qualsiasi chiave, secondo questo sistema, potrebbe essere quella corretta e restituire un risultato, al lordo di una possibile analisi, che sia apparentemente quello criptato con la password corretta.

Teoricamente applicando HE si andrebbe ad evitare gli attacchi di tipo brute force, proteggendo l'utente finale, spesso ignaro del corretto utilizzo delle password.

2.1.1 Differenze con la Deniable Encryption

Questo concetto di "decoy key", non è assolutamente una novità nel mondo della criptologia; infatti l'esistenza di una chiave alternativa/diversiva che restituisca un falso-positivo, richiama un'altra nota forma di crittografia, la **Deniable Encryption** [14], presentata nel 1997 da *Canetti* e tuttora utilizzata, per esempio nella creazione di un volume nascosto cifrato (vedi *Veracrypt* o *Truecrypt*).
La **HE** però si differenzia da quest'ultima poiché non possiede un numero limitato e

prestabilito di chiavi secondarie, bensì dovrebbe restituire una soluzione per l'intero spazio delle chiavi.

2.2 Funzionamento

La chiave di volta che regge l'intera Honey Encryption consiste nella costruzione del meccanismo esprimibile informaticamente che permetta tutto ciò, che come vedremo, non è assolutamente banale.

2.2.1 Struttura generale

Prima di entrare nel dettaglio del funzionamento, è necessario identificare gli elementi su cui si basa la struttura di questa nuova tecnica.

Message Space (M)

Il **Message Space** (in seguito abbreviato in **M**), o in italiano "spazio dei messaggi", non è altro che l'insieme di tutti i messaggi che possono essere inviati. Per esempio le parole di un dizionario, tutti i numeri da 0 a 1000, un insieme di messaggi scelti secondo un determinato criterio ecc.
Rappresenta l'agglomerato delle possibili scelte dell'utente.

Ad ogni messaggio viene generalmente attribuita una probabilità, che può essere banalmente *1/n* (con n il numero di messaggi contenuti in **M**) o può essere più complessa: relativamente alla diffusione delle parole nel linguaggio naturale oppure suddivisione in base alle preferenze o ancora seguendo degli algoritmi particolari.

Un primo elemento critico è rappresentato dalla costruzione di **M**: la scelta del range, la distribuzione di probabilità degli elementi, l'ampiezza, come assicurarsi che il messaggio originario sia contenuto in questo insieme ...
Nelle varie applicazioni, presentate successivamente, saranno mostrate varie possibilità: avere un insieme statico ampio e prestabilito (costruito con algoritmi particolari) oppure avere un insieme dinamico che varia in base al messaggio da inviare.

Seed Space (S)

Strettamente legato a **M**, il **Seed Space** (**S**) viene costruito tenendo conto del numero degli elementi di **M**. La combinazione delle stringhe binarie di n-bit, dette seed, permette di generare lo spazio dei seed, che dunque, in base alla scelta di n, avrà un'ampiezza e valori fissi, l'importante è possa essere unicamente associato a ogni messaggio almeno un seme.
Dunque ogni elemento appartenente a **M** verrà mappato, in base alla sua distribuzione di probabilità, su **S**. Dunque il numero di seed che verrà assegnato a un

determinato messaggio dipenderà dalla sua distribuzione di probabilità e dalla ampiezza di **S**: nel caso questa fosse *1/n*, il procedimento sarà immediato, negli altri casi saranno necessari ulteriori passaggi matematici per una corretta distribuzione. Il **DTE**, di seguito descritto, è il responsabile di questa delicata procedura

Distribution-Trasforming Encoder (DTE)

Il **Distribution-Trasforming Encoder**, abbreviato in **DTE**, può essere considerato l'elemento cardine su cui ruota l'intero sistema.
Possiede, come già accennato in precedenza, un ruolo fondamentale, infatti mappa **M** su **S**, e viceversa, attraverso apposite funzioni e algoritmi che tengono conto della struttura dei due spazi, dell'obiettivo da raggiungere e dal tipo di messaggio da condividere.
Il **DTE** riceve in input un messaggio e restituisce in output il rispettivo set di seed, dunque effettuando una semplice codifica messaggio → seed.
Viceversa, ricevendo in input un seed ne restituirà il relativo messaggio, applicando dunque la relativa decodifica seed → messaggio.

La costruzione e la condivisione del **DTE** rappresenta l'ennesimo punto critico e delicato dell'HE, forse il più importante, difatti le principali problematiche e applicazioni ruotano attorno a questo elemento (indipendenza tra chiave e distribuzione di probabilità del messaggio, come e dove condividere il **DTE**, come effettuare l'associazione ecc.).

2.2.2 Descrizione meccanismo di funzionamento

Dato un insieme di possibili messaggi di partenza (*M1, M2, M3 ... Mn*), associo a ognuno uno o più seed appartenenti a un Sp di p-bit. Questo processo viene eseguito dal **DTE** che si occupa di mappare lo spazio dei messaggi su uno spazio di seed, data una funzione di distribuzione.

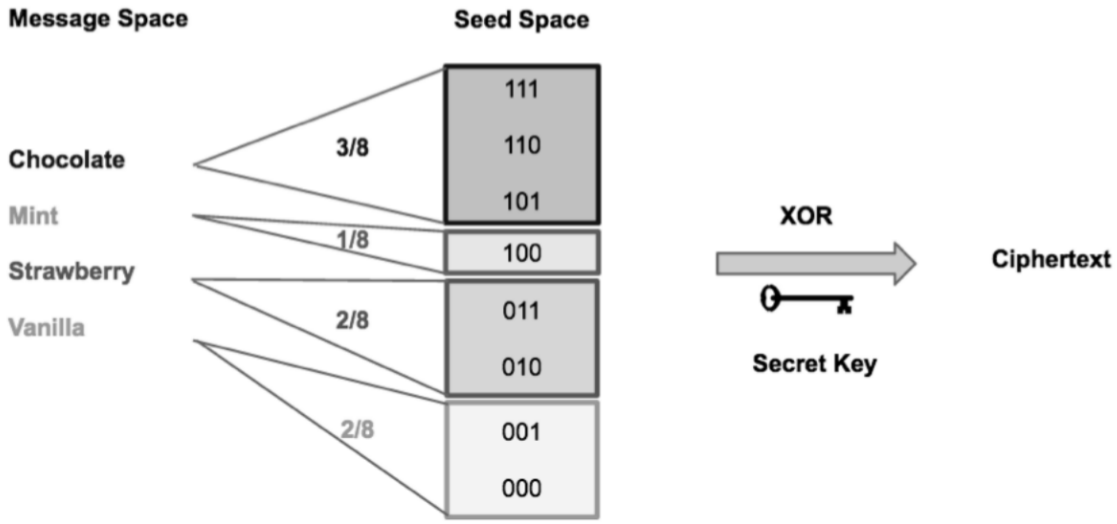

Figura 2.1: Schema Honey Encryption
©Honey Encryption - Applications [15]

Il mittente sceglie uno dei possibili messaggi **m** di **M**, il **DTE** ricevendo quest'ultimo in input, ne ricava il rispettivo seed **s**, questo viene criptato (generalmente attraverso uno xor) con la chiave **k** scelta dall'utente, il messaggio cifrato **c**. Successivamente **c** può essere ulteriormente cifrato con un algoritmo noto (DTE-then-Encryption) e inviato attraverso un canale di comunicazione prestabilito. Il destinatario, che può anche essere un server o il mittente stesso, utilizzando la **k2** in suo possesso, decifrerà **c** e, a prescindere dalla correttezza della chiave, otterrà un seed che verrà decodificato dal **DTE** (nel dettaglio verrà usata la tabella inversa di distribuzione di probabilità per ricostruire **M**), ottenendo un messaggio **m2** che, se **k2** sarà uguale a **k**, allora **m2** corrisponderà a **m**.
Un attacker, utilizzando una chiave errata, otterrà un seed diverso da quello originale che equivarrà a un messaggio diverso da quello originariamente inviato dal mittente; ma che, se **M**, **S** e **DTE** saranno correttamente costruiti, risulterà plausibile.

2.3 Limiti

Le principali limitazione emergono già durante la descrizione degli elementi componenti Honey Encryption (complessità nella produzione di **M**, **S** e nell'algoritmo di **DTE**), ma oltre a quanto precedentemente accennato, un'altra grossa problematica è rappresentata dal fatto che anche il legittimo destinatario potrebbe essere ingannato, semplicemente inserendo una chiave errata, dovuta o a un errore di battitura(in

media il 42% delle persone effettua almeno un errore durante la digitazione [16]) o a errori mnemonici.

2.3.1 Evitare Typo-Error

Il Dipartimento Informatico dell'Università di Seoul ha recentemente pubblicato un paper [17] dove presenta tre possibili schemi per cercare di porre un rimedio per evitare di confondere l'utilizzatore in caso di errore nella digitazione della chiave (si prevede una comunicazione tra un utente e un server).

One-factor

Consideriamo una comunicazione fra due entità, il *server* e un *utente*. In questo schema, quando l'utente, successivamente all'accesso sul server, cerca di prelevare delle informazioni precedentemente inviate, il server restituisce anche un testo cifrato con la stessa chiave usata per l'identificazione contenente alcune informazioni private dell'utente, condivise nella fase iniziale di registrazione.
Questo prevede che i dati sensibili siano realmente personali e che solamente l'utilizzatore originale del servizio ne sia a conoscenza, oltre alla responsabilità da parte del server di proteggere tali dati.

Two-factor

L'obiettivo è di migliorare l'incertezza e l'ambiguità delle informazioni fornite dallo schema precedente, la soluzione è rappresentata dall'introduzione di una terza entità, il database manager.
L'utente, oltre alla password usata per cifrare i messaggi, possiede un *PIN* (entrambe conservate nel database) che viene usato, al posto delle informazioni personali, per verificare l'integrità della chiave usata in fase di request. Infatti l'utente questa volta riceverà il codice cifrato e potrà verificare che sia corrispondente a quello da lui inserito, se non lo dovesse essere, saprebbe di aver inserito una password errata.

Hash-based

Al fine di escludere completamente la presenza di informazioni personali dell'utente sul server, viene sfruttata la proprietà di unicità delle *funzione di hashing*.
I dati sono cifrati nel database come hash, l'utente quando richiede un dato, riceve un hash che può confrontare con quello inizialmente condiviso con il server.

2.4 Applicazioni

Nonostante sia un sistema ancora giovane e sia notevolmente limitato, sono state pubblicate numerose implementazioni nei più svariati ambiti.
Alcune di queste seguono alla lettera l'espressione originaria, altre ne propongono un'evoluzione oppure ne prendono semplicemente spunto, per poi sviluppare un ulteriore tecnica.

Tutte però conseguono il medesimo obiettivo, cercare di limitare gli attacchi brute force, o quelli a conoscenza parziale, fornendo un sistema di decrittazione con messaggi errati plausibili.

Durante la descrizione degli elementi e del funzionamento dell'Honey Encryption, si è utilizzata come unità "di condivisione" il *messaggio*. Ma tutto ciò è applicabile, come vedremo, anche ad altre tipologie di informazioni.

Chiaramente le prime implementazioni furono quelle introdotte dagli autori stessi dell'Honey Encryption, al fine di dare un'indicazione sull'utilizzo e per mostrarne le potenzialità.

2.4.1 Esempi proposti da A.Juel

A.Juels e T.Ristenpart hanno presentato due possibili applicazioni concentrandosi di più sull'aspetto matematico che sulla realizzazione in sè.

Carte di Credito

La prima su dati estremamente strutturati, nello specifico sulle *carte di credito* [1](il loro numero, PIN, CVV), i quali, seguendo algoritmi di costruzione prestabiliti, permette elaborare un sistema efficiente. In seguito vedremo l'implementazione, in *Python*, proposta da un gruppo di ricercatori del MIT.

RSA

L'altra applicazione riguarda la chiave segreta utilizzata in *RSA* [1], in particolare con l'obiettivo di rallentare gli attacchi di tipo brute force offline.

2.4.2 Honey Encryption per i numeri delle Carte di Credito

Una delle pubblicazioni [15]più importanti è quella presentata da un team di ricerca del reparto di Computer & Network Security del MIT, composto da *Nirvan Tyagi*, *Jessica Wang*, *Kevin Wen* e *Daniel Zou*.

Infatti furono tra i primi a proporre delle applicazioni reali, fornendo il relativo codice (in python), tramutando in realtà la teoria presentata all'Eurocrypt. La prima

implementazione non è altro che un'attuazione di quella già accennata da Juels e Reinstpart sull'utilizzo dell'Honey Encryption per i numeri delle carte di credito [18].

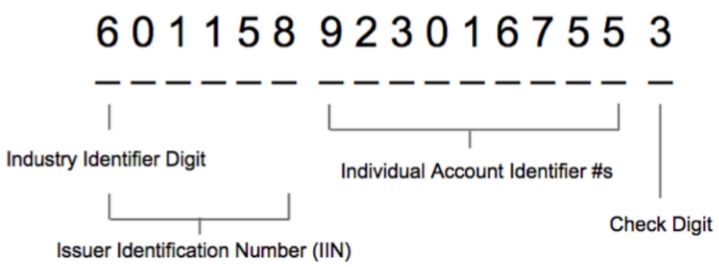

Figura 2.2: Esempio struttura numero di carta di credito
©Honey Encryption - Applications [15]

La struttura delle carte di credito è di seguito illustrata:

- dalle 12 alle 19 cifre per il numero di carta di credito
- la prima cifra indica il produttore della carta di credito (*4 - Visa, 5 - Mastercard ...*) e da indicazione anche sul numero totale di cifre che avrà la carta *(Visa e Mastercard sedici, American Express quindici ...)*
- Le prime 6 cifre si riferiscono all'**IIN** (Issuer Identification Number), il numero identificativo dell'emittente bancario-finanziario che rilascia la carta, per esempio, noi sappiamo che le carte Mastercard iniziano con 5, tutte le carte nel range 500000-599999 verranno ridistribuite ai vari distributori e alle banche, ai quali verranno assegnati dei numeri prestabiliti *(es. 540608 è il numero delle carte di Credito prepagate della Mastercard assegnate alla Banca di Sondrio)*
- Le rimanenti cifre fino alla penultima rappresentano un identificatore personale e univoco (**Account Number**), l'algoritmo generatore è unicamente a disposizione del fornitore della carta ed è segreto.
- L'ultima cifra è un *checksum* calcolato tramite **l'algoritmo di Luhn** A.1

Costruire lo spazio dei messaggi è abbastanza semplice: una volta ottenuti tutti gli **IIN**, tramite un semplice algoritmo è possibile generare l'intero spazio **M** partendo dal numero di carta che si vuole cifrare, infatti in base all'**IIN** si saprà da quanti numeri **p** è composto **Account Number**, si genereranno 10^p possibilità, verrà generato un opportuno **S** e mappato di conseguenza con probabilità $\frac{1}{10^p}$.

Applicazioni reali

Vi è una duplice funzionalità, **HE** potrebbe essere applicata a un credit card database, dove l'utente andrà a salvare i propri dati delle carte di credito. Oppure, semplicemente, durante una comunicazione, per poter inviare in tutta sicurezza i dati (in questo caso andrebbe condiviso anche l'intero spazio dei messaggi, visto che è vincolato alla carta di credito).

Esempio: Credit Card Database

L'utente salva la propria cc nel database, che conserverà **M** criptandolo e un id (che può essere l'identificatore dell'utente) e fornirà all'utente una password, che potrebbe anche essere scelta, la combinazione password e identificatore fornirà il corretto seed che verrà ricercato all'interno di **M**. Nel caso venisse inserita una password errata, verrà comunque restituito un altro seed che corrisponderà a un diverso numero di carta di credito.

2.4.3 Honey Encryption applicato alle semplici domande inglesi

L'altra implementazione riguarda il linguaggio naturale [18], vista la difficoltà di coprire l'intero spettro della lingua inglese, sia dal punto di vista della complessità delle strutture grammaticali che alle dimensioni del dizionario, gli autori si sono limitati ad applicare HE alle domande di tipo semplice, per esempio *"Do you like this?"*.
Attraverso alcune librerie (*nltk* [19], *nodebox* e *inflect* [20]) Python che permettono di generare frasi grammaticalmente corrette partendo da una struttura data (nltk) e di coniugare e declinare correttamente le parole componenti (Nodebox e Inflect), hanno sviluppato il loro progetto.
Le domande in inglese generalmente iniziano con un verbo, hanno un oggetto, un predicato e opzionalmente un complemento.
Hanno dunque utilizzato due dizionari contenenti i più comuni verbi e nomi (nelle varie forme), con associato un relativo valore indicante la frequenza di utilizzo all'interno della lingua, cioè banalmente un indice che indichi quanto viene usato quel termine.
Un algoritmo sfruttando le potenzialità di nltk, ricevendo in input questi dizionari e le possibili strutture delle frasi (vengono attribuite delle regole per la generazione delle strutture, simil a BNF), genera le molteplici combinazioni e ne calcola la frequenza in base a determinati criteri, inserendo il tutto in un database (che sarà il nostro **M**). Successivamente una funzione ($probabilityfunctionAPI$) mappa lo spazio dei messaggi, come avveniva per le altre applicazioni.

Applicazioni reali

Le applicazioni reali, per ora sono limitate, però si potrebbero ampliare, usando un opportuno linguaggio, le possibili strutture generabili. Non è stata posta una soluzione per quanto riguarda il senso finale della frase, essendoci il rischio concreto di produrre messaggi non di senso compiuto.

Esempio: Utilizzo di dizionari relazionati

Il mittente inserisce il messaggio m, che viene decomposto con **spacy** [21] e ogni suo termine t analizzato. Usando **DataMuse** [22] o Wordnet [23] ottiene un insieme di parole "relazionate" al termine t, genera dunque dei sotto-dizionari (per l predicato, verbo, soggetto ...) relazionati.
Un algoritmo che utilizza *NLTK*, ricevendo in input questi sotto dizionari e la struttura della frase ottenuta tramite **spaCy**, produce tutte le possibili combinazioni, di queste ne estratte un set che possieda una distribuzione di probabilità simile a quella originaria, questo sarà **M**. Il **DTE** si occuperà di effettuare e codificare il messaggio, che successivamente attraverso un canale sicuro, verrà condiviso, assieme a **M**. Tralasciando per un attimo il linguaggio, che come è emerso è un aspetto molto delicato e complesso, implementare **HE** su dati aventi una struttura lineare e facilmente esprimibile attraverso funzioni matematiche, risulta molto più semplice e schematico, come si è visto per il numero di carte di credito.

2.4.4 Honey Encryption per i Sensor Data

Le applicazioni che verranno presentate di seguito, sono basate sulla condivisione di dati strutturati, come possono essere quelli rilevati dai sensori hardware.

Molti lavori si focalizzano su un aspetto ben preciso: la condivisione dei dati raccolti da sensori (*IoT generic* [24], WSN ...)
Questi device generalmente rilevano informazioni basiche, molto semplici, come on/off, dati grezzi, valori analogici e digitali, e difficilmente possiedono architetture complesse per elaborarli e analizzarli, tanto che devono o inviarli a una struttura centrale o condividerli con gli altri dispositivi per ottenere e gestirne i risultati; dunque proteggere il canale di comunicazione e gli stessi sensori, risulta essere di fondamentale importanza.
Come accennato in precedenza, si tratta spesso di modellare dati di basso livello come bit, byte o numeri semplici, pertanto è immediata, attraverso semplici funzioni, la creazione di un *Message Space* e la seguente mappatura (magari in base alla frequenza con cui avviene il prelievo).
Una possibile estensione potrebbe prevedere un sistema centrale, con il compito di riconoscere un attacco malevolo verso un suo sensore o su un canale.

Esempio: Sistema centralizzato con Sensori IoT

I sensori, oltre all'hardware di lettura, dovrebbero possedere anche un hw responsabile della comunicazione, per esempio tramite rete wifi, quest'ultimo componente, prima di inoltrare i dati sul canale, li passa al **DTE** che, in associazione con un **M** opportuno per il tipo di dato rilevato, restituisce il seed che verrà criptato con una password condivisa a priori. Quest'ultimo verrà inviato al sistema centralizzato, che avrà in un database ID del sensore e la relativa password, oltre che un'indicazione per l'opportuno **M** da utilizzare. Ricevuto il seed, lo decripterà e leggera il dato.
Per quanto riguarda le richieste di lettura dal sistema centrale al sensore, il procedimento sarà l'inverso. Se un utente malevolo dovesse "simulare" di essere il **SC**, dovrebbe conoscere comunque la password.
Per riconoscere degli attacchi, se il canale di comunicazione dovesse essere interrotto, la soluzione è immediata. Se invece non dovesse esserlo, oltre al dato, si potrebbe inviare un *timestamp* e/o un *bit* indicante che si tratta di un messaggio inviato in seguito a una richiesta, permettendo al **SC** di riconoscere messaggi arrivati in seguito a richieste "esterne".

2.4.5 Honey Encryption per il Cloud e i Database Online

Un gruppo dell'Università di Computer Science di Penang, Malesia, ha pubblicato un'interessante applicazione dell'Honey Encryption, volta a proteggere i file conservati dagli utenti nel cloud [25].
Il meccanismo si basa sul mascherare il nome dei documenti.
La denominazione dei file possiede una struttura ben precisa: un nome (composto da lettere o numeri, anche casuale) e un'estensione. Il nome generalmente non supera i cinquanta caratteri, mentre le estensioni non superano i quattro.
Quando l'utente decide di caricare il file, il programma ne estrae le due parti, successivamente costruisce due Message Spaces, il primo generando nomi con una distribuzione di probabilità simile a quella del nome del file, il secondo, similmente, estraendo le estensioni da una lista prestabilita. **M1** e **M2** vengono dunque mappati su due Seed Spaces con dimensioni opportunamente scelte. Vengono prelevati i rispettivi seed, combinati e criptati con un algoritmo e una password scelta dall'utente. Si ottiene così un file criptato che verrà inviato al cloud server (insieme ai due Message Spaces) attraverso un canale sicuro, generalmente usando *TLS*.
Quando l'utente scarica il file dal cloud, lo decripta in locale, inserendo la password utilizzata nella fase precedente, riottenendo i due seed e, in seguito al processo inverso, la denominazione corretta.
Un eventuale attacker, intercettato il file criptato dal device (PC, smartphone ...) dell'utente, dal cloud o dal canale di comunicazione, inserendo una chiave errata,

otterrà comunque due seed e quindi un file plausibile e sarà costretto ad analizzarlo, aprendolo (ottenendo un documento non valido), perdendo ulteriore tempo e rendendo vani tentativi di recupero password tramite brute force attack.

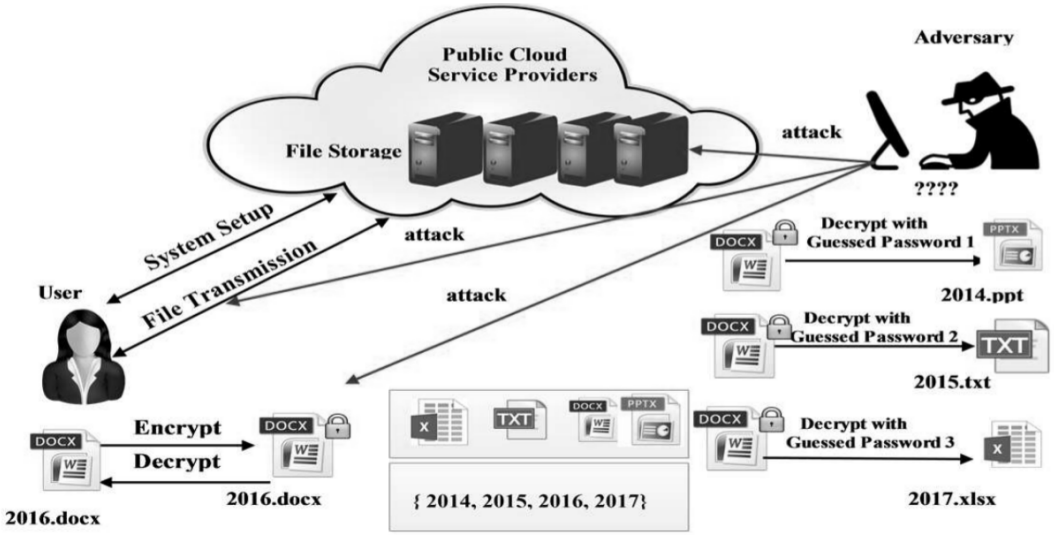

Figura 2.3: Schema Cloud Storage con Honey Encryption
©Implementing the Honey Encryption for Securing Public Cloud Data Storage [25]

Problema

Un possibile problema che potrebbe sorgere è rappresentato da "dove deve avvenire la criptazione" a livello server oppure a livello utente, a livello di efficienza e di uso multiutente, il server garantirebbe un miglior risultato, ma in questo caso bisognerebbe essere sicuri del canale utilizzato, dei protocolli, della sicurezza del device usato dagli utenti e, ultimo ma di fondamentale importanza, della maggior difficoltà di gestione.

2.4.6 Visual Honey Encryption

Allontanandosi da queste tipologie di applicazioni, è interessante presentare il lavoro svolto da *Jin Won Yoon* insieme ad alcuni colleghi delle Università di SungKyunKwan (Korea). Nel 2017 presentarono uno sviluppo dell'Honey Encryption, rielaborato in chiave steganografica, la **Visual Honey Encryption** [26].

L'idea proposta da *J.W. Yoon* consiste nell'unire due tecniche: la steganografia e l'Honey Encryption. La prima si prefigge di nascondere il messaggio segreto all'interno di un contenitore, per esempio nelle immagini, dei file multimediali et similia. La seconda, come già ampiamente presentato, ha l'obiettivo di sviare tentate violazioni

attraverso l'utilizzo di messaggi esca-diversivi.
In generale, è possibile celare all'interno di un'immagine (o video, musica ecc.) un'informazione segreta, così da ingannare un utente malevolo.
Però se l'attaccante dovesse aspettarselo, potrebbe recuperare i dati analizzando l'immagine, sebbene non sia semplice. Volendo si potrebbe encodare in ASCII il contenitore e criptarlo con una chiave segreta, in questo caso però se venisse usata una chiave sbagliata in fase di decrittazione, non si otterrebbe nessuna immagine.
E' stato dunque proposto uno schema dove l'Honey Encryption viene applicata alle immagini, creando non tanto un Message Space, quanto un Image Space.
I due interlocutori scelgono un'immagine **p** da un database pubblico, il mittente genera innanzitutto una pseudo-immagine randomica **r** che serve a creare disturbo e rumore in quella originale. Successivamente viene creato uno spazio delle immagini I visivamente simili a **p** e con una distribuzione di probabilità prefissata (o ricavata da **p**). Si estrae **c**, l'immagine all'interno di I corrispondente a **r**. Viene quindi generata una stego-image **s** tramite lo xor tra **c** e il messaggio da inviare **m**. Infine, prima di procedere all'instradamento dei dati, **s** viene codificata dal nostro **DTE** ottenendo il rispettivo seed, mappando su **I**, e criptata con una chiave scelta **k**.
Nella fase inversa, in base alla chiave utilizzata, si otterranno diversi seeds, che decodificati attraverso il **DTE** e **I** (che verrà condiviso), daranno origine a diverse immagini, indistinguibili a occhio, alle quali dovrà essere eseguita una stego-analisi per ottenere il messaggio.

In questa applicazione, l'Honey Encryption viene applicata solo parzialmente, infatti se l'attacker dovesse riuscire facilmente ad analizzare le immagini ottenute, solamente una restituirà un messaggio di senso compiuto o valido, visto che non vi è garanzia di **HE** a livello di messaggi, ma solo di immagini. Riprendendo le implementazioni momentaneamente messe da parte, relative al linguaggio, queste si rivelano, dal mio punto di vista, le più intriganti, sebbene le più laboriose.

2.4.7 Android Framework Messagging con l'Honey Encryption

Oltre all'applicazione precedentemente descritta, *Rasmita Sahu*, professoressa dell'Università di Bhilai (India), ha pubblicato un *framework Android* sviluppato sull'Honey Encryption per messaggiare con i nostri smartphone [27].

In generale un framework è una struttura che funge da intermediario tra il sistema operativo e il software che lo utilizza. Nel nostro caso, si tratterà di un meccanismo basato sull'Honey Encryption, che, una volta utilizzato dalle app permetterà di cifrare e decifrare i messaggi attraverso questa tecnica.

1. La prima è quella di registrazione: ogni utente si registra al servizio, fornendo l'*IMEI* del proprio smartphone (che è possibile rilevare e recuperare automaticamente usando le librerie android) e delle credenziali, che verranno conservate su un server. Per ogni utente verrà fornita una chiave univoca per l'autenticazione.

2. Nella successiva fase avviene la vera e propria comunicazione: l'utente, autenticato, inserisce il messaggio, al quale verrà assegnato un seed dal DTE in base all'entropia e secondo le regole di Markov, che creando un *Statical coding scheme* A.4 (e un relativo **M**), che verrà condiviso tra gli interlocutori. Il seed viene dunque criptato con un algoritmo Blowfish a una chiave pubblica.

3. La terza, e ultima fase, è quella di decrittazione e decodifica: se un device non autorizzato tenterà di collegarsi durante la comunicazione, il server genererà dei messaggi secondo i modelli *N-Gram* A.5. A sua volta il destinatario inserirà la chiave scelta e, se corretta, otterrà il messaggio originale, recuperato dal *SCS*.

Limite

Vi è un grosso ed evidente limite, in caso di utilizzo di una chiave errata o del rilevamento di un device non autorizzato, il messaggio restituito non segue le regole dell'HE, bensì viene generato dal server. Dunque verranno seguiti due flow diversi e questo permetterebbe all'attaccante di riconoscere i messaggi esca. Le successive implementazioni che verranno introdotte di seguito, le considero più particolari e complesse, visto che cercano di "schematizzare" il linguaggio naturale, cercando di adattarlo all'Honey Encryption.

2.4.8 Honey Chatting

Il *CIST* dell'Università di Seoul (Korea), nel 2016 presentò un'applicazione dell'Honey Encryption [28] basata sulle le c*atene di Markov* (linguaggio costruito partendo da n-grammi, sottosequenza di n elementi di una data sequenza) e un database pubblico di frasi, al fine di generare un programma chat, che, se dovesse essere violato, mostri frasi casuali ma plausibili, almeno per un calcolatore.
Data una piattaforma chat, sviluppata in questo caso con Java, gli utenti accedono al servizio inserendo il proprio username, la password (condivisa e nota tra gli utilizzatori) e un database di riferimento tra quelli in lista, necessario per permettere allo schema di codifica/decodifica su n-grammi di costruire frase semanticamente plausibili.
L'utente, come nei casi precedenti, invia un messaggio, questo viene codificato dal DTE in base all'entropia e allo schema statistico di probabilità. Successivamente si

otterrà la criptazione dei messaggi con un algoritmo a scelta (nel loro caso PBE), utilizzando la password precedentemente inserita.
Chi si collegherà alla chat, avrà a disposizione lo stesso DTE e lo stesso schema, se inserita la password corretta, dal modello verrà ricostruita la frase corretta (o più frasi se presenti), altrimenti, sfruttando il database precondiviso, lo schema precedente e le *regole di Markov*, verrà generata una chat falsa tra i gli interlocutori.

Applicazioni Reali

Come in precedenza, anche qui si è limitati dall'utilizzo di meccanismi statistici per la costruzione delle frasi. Oltre al fatto che generalmente vengono generati messaggi sgrammaticati, ma su quest'ultimo dettaglio, se si utilizzasse Honey Chatting in una piattaforma online ludica, per esempio di videogiochi *MMORPG* o *streaming*, dove spesso la grammatica passa in secondo piano, i messaggi potrebbero risultare fedeli all'originale.

2.4.9 A Novel Approach for Adaptation of HE to Support NL Message

Un gruppo di ricerca del dipartimento di sicurezza informatica dell'Università di Penang (Malesia), all'*IMECS 2018* tenutosi a Hong Kong, pubblicò un notevole lavoro di adattamento dell'Honey Encryption al linguaggio naturale [29], fine di supportare qualsiasi tipologia di messaggio (in inglese).
Utilizzando unicamente due servizi, *nltk* e *WordNet* (un dizionario online), e un complesso algoritmo DTE, sarebbero riusciti a generare frasi esca corrette sia dal punto di vista grammaticale che semantico, dunque completamente irriconoscibili sia da un calcolatore che da un umano.
A differenza de "Honey Chatting" e del framework Android, che, usando dei modelli n-grammi, necessitano di essere sottoposti a una fase di training e il risultato rimane comunque riconoscibile sotto alcuni aspetti (esecuzione di due flow diversi, limiti IA ...), questo nuovo approccio garantirebbe anche una correttezza semantica.
Inoltre, al contrario dell'esempio proposto dal MIT, non si limita a un sottoinsieme di possibili frasi (solo le domande semplice), ma all'intero spettro della linguaggio.
Il funzionamento è stato solamente accennato, probabilmente per questioni legali, ma la struttura generale è stata grosso modo presentata, ed è di seguito descritta: sfruttando le apposite librerie python, nltk riconosce la struttura grammaticale della frase inserita dall'utente, restituendo una lista iterabile composta da token, uno per ogni parola, con un riferimento al lemma, alla dipendenza e al tipo di ruolo assunto nella frase stessa.
Successivamente vengono analizzati i singoli vocaboli e prelevati i codici identificativi

univoci da WordNet e combinati ottenendo un seed numerico **s**. Come in precedenza, avviene la criptazione di s con un algoritmo noto e il messaggio **c** così ottenuto verrà condiviso insieme alla struttura della frase (la lista) opportunamente criptata.
In fase di decodifica, estratta la combinazione degli id e la struttura, l'algoritmo inverso, elaborerà la frase.
Purtroppo non ci hanno fornito ulteriori spiegazioni, se non uno screenshot, su come lavora il DTE per generare frasi di senso compiuto e su come codificano la lista. Se in futuro gli autori renderanno pubblico il programma o la documentazione, potrebbe rappresentare una svolta nell'ambito della sicurezza informatica. Nel frattempo, basandomi su quest'ultima implementazione (o meglio, sul concetto che ci sta dietro) ho elaborato la mia personale applicazione, argomento della tesi.

Capitolo 3

Progetto - Honey Encryption per messaggi in Linguaggio Naturale

3.1 Motivazione

Le motivazione che mi hanno portato a sviluppare l'ennesima applicazione dell'Honey Encryption per il linguaggio naturale, è semplice: tutte le implementazione sul linguaggio precedentemente presentate, o comunque attualmente disponibili e pubblicate, non hanno reso disponibile l'algoritmo utilizzato, il codice et similia, ma si sono limitate a presentare una soluzione rimanendo sul generis, senza fornire una risposta a "come funziona la generazione dei messaggi", se non quella che dietro c'è un processo automatizzato, basato spesso su servizi di intelligenza artificiale.

Dunque attualmente nessuno ha cercato di schematizzare e strutturare con regole logiche il linguaggio naturale, rispettando l'idea originaria dell'Honey Encryption.

3.2 Descrizione generale

Il mio progetto, che verrà descritto nel dettaglio in questa tesi, consiste in un programma che schematizza rigidamente, con regole predeterminate, una frase inserita in input, successivamente genera un seed numerico che è strettamente legato alle combinazioni delle parole componenti la frase, lo cripta con una chiave inserita dall'utente.
In fase di decodifica, decriptando, si otterrà un seed che, se la password sarà corretta, permetterà di ricostruire la frase di partenza, altrimenti ne costruirà un'altra. La correttezza grammaticale, è soddisfatta grazie ai servizi di natural language processing utilizzati, purtroppo invece la correttezza semantica non è garantita.

Il lavoro, ancora in fase primordiale, attualmente permette di criptare con **HE** frasi semplici, cioè composte da un singolo periodo, in inglese. Ma in futuro, come

spiegato nella sezione successiva riguardante le possibili evoluzioni, seguiranno notevoli progressi.
Verrà anche spiegato come mai sono stati utilizzati determinati servizi, le motivazioni che stanno dietro alle scelte strutturali e i limiti incontrati.

3.2.1 Meccanismo di funzionamento in generale

L'utente inserisce una *frase* e una *password* segreta a sua scelta, il programma restituisce una stringa alfanumerica, che corrisponde al seed criptato con *AES* [30] e codificato in *base64*. Per riottenere il messaggio originale, dovrà essere inserita tale sequenza di caratteri e la password corretta, viceversa, il programma decripterà in maniera diversa, ottenendo un seed diverso (combinazione degli indici delle parole nel dizionario) e una frase diversa.
La correttezza grammaticale è assicurata dal fatto che viene inviata anche una indicazione della struttura grammaticale e dalla presenza di rigide regole. Scritto in

Python, sia per la versatilità garantita da questo linguaggio di programmazione, che per i servizi di nlp disponibili, sfrutta le seguenti librerie: spaCy 3.2.2, inflect 3.2.2, pattern 3.2.2, secrets 3.2.2, Crypto (per AES e PBKDF2)3.2.2 e alcuni moduli di servizio (csv, base64, argparse) 3.2.2.

3.2.2 Librerie utilizzate

Prima di commentare il codice, è opportuno presentare le librerie e come sono state interconnesse tra di loro.

spaCy

SpaCy [21] è una libreria python opensource per l'elaborazione del linguaggio naturale (Natural Language Processing). Questo toolkit permette di analizzare e processare frasi nelle più comuni lingue internazionali (inglese, italiano, portoghese, tedesco, spagnolo, francese e greco): sezionando la frase nelle singole parole, punteggiatura compresa, e riconoscendo il ruolo di ogni termine, il tipo (verbo, nome, articolo ...), le dipendenze fra di essi (soggetto, articolo possessivo . . .) e il lemma A.8.
Viene dunque costruito un albero sintattico-grammaticale della frase, similmente con quanto accade utilizzando lo Standford Parser A.6 , sebbene quest'ultimo sia un software più potente e preciso, ma, visto che per utilizzarlo è necessario lanciare un server Java sulla propria macchina, lo rende particolarmente più delicato da usare. SpaCy, essendo direttamente ottimizzato per Python, garantisce una migliore compatibilità.

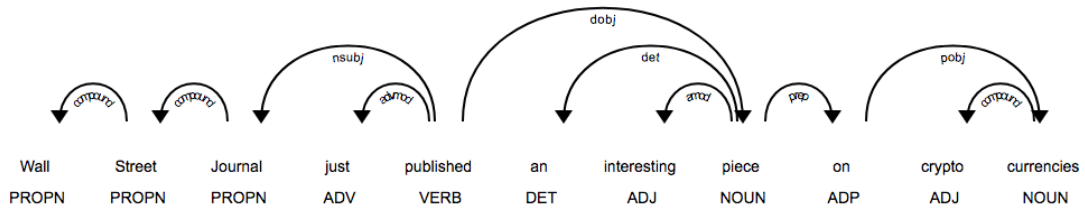

Figura 3.1: Spacy Tree
©nlpforhackers.io

Per riconoscere la struttura delle frasi, spaCy richiede un lavoro di training, inizialmente, in fase di installazione, è possibile utilizzare dei modelli standard, forniti direttamente dagli sviluppatori, già trained.
Le feature e le opzioni disponibili in spaCy sono molteplici, ma come già accennato, per il mio progetto utilizzo solo le funzioni principali: Tokenization, POS tagging, Dependency Parsing e Lemmatization [31].

Insieme a *NLTK* [19], rappresenta il punto di riferimento per chi vuole sfruttare le potenzialità del deep learning e del nlp, sviluppando codice in Python.
Rispetto a *NLTK* presenta però delle importanti differenze, che mi hanno portato a preferirla: la tokenization è orientata agli oggetti e presenta numerosi opzioni aggiuntive che potrebbero rilevarsi fondamentali per estensioni future (Sentence Boundary Detection, Named Entity Recognition ecc.).
Inoltre, *spaCy* è orientato alla produttività e ai consumer, a differenza di nltk che è indirizzato verso il segmento didattico e di ricerca.

Inflect

Come dice il nome stesso, la libreria disponibile per python, inflect [20], permette di flettere e declinare correttamente i nomi e gli articoli (inglesi), partendo dal lemma originario.
Si tratta anche questo di un progetto opensource e gratuito che fornire numerose possibilità di flessione e manipolazione delle parole, quelle usate nel mio progetto sono le seguenti:

- ottenere la declinazione al plurale di un determinato nome (anche se irregolare) e viceversa per il singolare.
- Restituire la corretta versione di "a/an" in base al sostantivo ricevuto in input.

Per generare il corretto output, inflect si basa sulle regole de *"Oxford English Dictionary"* e sulle linee guida presenti nel *"Fowler's Modern English Usage"*.

Pattern

Pattern [32] è una libreria python molto versatile, che può essere utilizzato per il data mining, natural processing language, machine learning, network analysis e canvas visualization.
Nello specifico per il nostro scopo è stato utilizzato il modulo pattern.en, che contiene strumenti nlp per la coniugazione dei verbi, declinazione dei sostantivi e analisi delle frasi. I tools disponibili in questo package sono numerosi, per questo progetto ne sono stati utilizzati solo alcuni:

- article() - Articoli determinativi e indeterminativi
- conjugate() - Coniugazione dei verbi in qualsiasi forma verbale (inclusi i verbi modali).

Altri moduli, che potrebbero risultare utili in futuro sono: implementazione diretta di WordNet, parsing tree delle frasi (come spaCy) e generazione comparativi e superlativi.

Questa libreria è stata usata, sebbene abbia molti strumenti in comune con spaCy e inflect, perchè è l'unica che permette una corretta coniugazione dei verbi.

SECRETS

La nota libreria random, utilizzata da moltissimi linguaggi, in realtà non genera randomicamente, infatti è definita come PRGN Pseudo-random poiché si basa su un seed di partenze che genera dati random predeterminati.
Chiaramente nello sviluppo di un qualsiasi progetto di sicurezza, che prevede la presenza di caratteri alfanumerici random, questo non è tollerabile.
Il modulo secrets rappresenta la soluzione al problema sopra esposto, visto che garantisce la generazione di dati realmente random e difatti viene usato per password, OTP, dati di autenticazione, tokens ecc.
Nel nostro caso, serve per generare una stringa numerica iniziale che risulta essere il nostro seed di partenza, utilizzato in supporto alla nostra frase inserita in fase di criptazione, per costruire seed sempre diversi, anche se la frase è la stessa. Oltre ad essere usata anche per creare un valore di salt casuale.

Python Cryptography Toolkit

Pycrypto [33] contiene una collezione di funzioni hash e algoritmi crittografici, utilizzabili per criptare, e decriptare, i nostri dati. All'interno del mio progetto, la scelta è ricaduta su un noto algoritmo a cifratura a blocchi, *AES* [30], disponibile chiaramente all'interno di questa libreria. Le motivazioni per cui viene utilizzato AES sono

semplici: è un algoritmo già di per se sicuro contro i brute-force attack e soprattutto rappresenta uno standard tra gli algoritmi di cifratura.

Oltre a AES, implementa anche dei utili protocolli di Key Derivation Function, come *PBKDF2* A.10, utilizzato nel nostro caso, che permettono di convertire la nostra password in una chiave di lunghezza fissa (256 bits per AES-256) opportunatamente salata con probabilità uniforme.

Altri moduli

Altre librerie utilizzate per questo progetto sono le seguenti:

- *argparser* per aggiungere opzioni e argomenti alla chiamata dalla riga di comando ("-e" e -"d")
- *csv* per leggere e scrivere su file di tipo .csv, formato scelto per memorizzare i dizionari
- *base64* per codificare e decodificare una stringa in base64

3.2.3 Utilizzo

Il funzionamento generale dello script in Python è abbastanza semplice, sebbene la struttura non sia banale:

Esistono due possibili opzioni lanciabili da linea di comando, **-e** che permette di criptare un messaggio con una password scelta e **-d** che invece è il processo inverso e vuole come argomento il *seed pubblico* (in base 64) e una *password*.

Come lanciare e utilizzare il programma

Dopo aver installato Python e le necessarie libreria, da terminale (il programma è utilizzabile sia da Linux che da Windows) lanciare lo script con il comando **-e** seguito da un **testo** e una **chiave**, per criptare.

```
1 $ python honey.py -e 'Message' 'PASSWORD'
```

Per decifrare, usare l'opzione **-d** seguita dal risultato dell'output precedente (**il seed pubblico**) e la **chiave**.

```
1 $ python honey.py -d 'PublicSeed' 'PASSWORD'
```

3.2.4 Dizionari

Per identificare i sostantivi, gli articoli, i verbi, gli aggettivi ecc. vengono utilizzati dei dizionari in formato *csv*, contenenti le parole più comuni.

Inoltre, anche per memorizzare le strutture è necessario utilizzare un database, perché non è possibile esprimerle matematicalmente, mantenendo la correttezza grammaticale, dunque vengono salvate in *STR.csv* e, in caso l'utente dovesse inserire una frase, la quale struttura grammaticale non è presente nella lista, verrà aggiunta alla nostra lista.

Tabella 3.1: Dizionari

Lista dei dizionari utilizzati dallo script

NAME	TIPO	ESEMPIO
ADJ	Aggettivi	bizarre, curvy, kinf ...
ADP	Pre-Post condizioni	along, before, for, ...
ADV	Avverbi	abnormal, more, narcotic ...
CC	Congiunzioni coordinative	but, and, or ...
DET_PLUR	Articoli determinativi plurali	the, some, any
DET_Q	Articoli determinativi quantificatori	few, little, bit
DET_SING	Articoli determinativi singolare	the, a, this, ...
MD	Verbi modali	can, may, shall, ...
NOUN	Nomi	dog, human, fsociety, ...
PDT	Predeterminanti	such, all, quite, ...
POS	Particella possessivo	', 's
PROPN	Nomi Propri	Zaira, Raphael, Tiffany, ...
PRP_O	Pronomi oggetto	me, you, her, ...
PRP_POSS	Pronomi possessivi	my, yout, their, ...
PRP_S	Pronomi Soggetto	I, You, He ...
RP	Particelle	along, by, roundunder
VERB	Verbi	hack, love, dead, ...
STR	Strutture Grammaticali	"(0, 'PRP', 'nsubj', 1, 'VBP')", ...

Questi database non verranno inviati insieme al messaggio criptato, ma saranno contenuti nella stessa cartella dello script python.

3.3 Funzionamento

3.3.1 Fase di criptazione

L'utente che vuole criptare un messaggio, inserisce in input una password, senza vincoli su lunghezza, caratteri o altro, e il messaggio (che per questa versione dovrà essere di singolo periodo) in lingua inglese.

Attraverso il tool nlp di *spaCy*, il testo viene analizzato, restituendo una lista di token, uno per ogni parola. I token contengono molte informazioni, nonché contengono la struttura intrinseca ad albero della frase, nel nostro caso viene memorizzata:

```
λ python honey.py -e "My little secret" "non dirlo a nessuno"
Struttura
My little secret
Struttura Tokenizzata
[(0, 'PRP$', 'poss', 2, 'JJ'), (1, 'DTQ', 'det', 2, 'JJ'), (2, 'JJ', 'ROOT', 2, 'JJ')]
```

Figura 3.2: Struct Screenshot

Per un maggior approfondimento, consiglio la lettura dell'appendice **"Parsing e Tokenization delle frasi"** A.8.

- la posizione della parola nella frase
- il suo POS TAG
- il tipo di dipendenza
- la posizione della parola da cui dipende
- il POS TAG della parola da cui dipende

Il verbo principale sarà il *ROOT*, da cui a cascata dipenderanno gli altri elementi.
Caricati tutti i dizionari a disposizione (vedi paragrafo 3.2.4, verranno caricati, e salvati in una lista, gli indici delle parole (nel loro lemma) e l'indice della struttura all'interno dell'apposito csv, se la struttura è nuova, verrà aggiunta.
Generato un numero random di 128 numeri attraverso il modulo secrets, a questo verranno tolti i primi quattro numeri (che permetteranno di derivare l'indice della struttura), i restanti 124 verranno divisi in blocchi quasi uguali tramite la funzione *divideNParts* in base al numero delle parole che compongono il testo da criptare.
Sia per l'indice della struttura che per quelli dei vocaboli, verrà eseguita una operazione di mod seguita da operazione di *fill* e sottrazione, tra la n parte del numero random, l'indice e la lunghezza del relativo dizionario per far si che dato il numero numero, se si dovesse fare il modulo con la lunghezza del dizionario, si otterrà il seed originario.
Si otterrà una stringa di 124+4 numeri, questa verrà mappata, cifra per cifra, per ottenere una lista di 128 numeri tra 0 e 250 con nonce, questo procedimento è di fondamentale importanza, perchè garantisce l'Honey Encryption.
Partendo dalla password inserita, con la funzione *PBKDF2*, dopo aver generato con

il modulo secrets un salt di 8 bytes, si ricaverà un key, da utilizzare in *AES*.
Questa lista sarà criptata con *AES* usando la key derivata e encodata in *base64*, ottenendo quindi il seed pubblico (che sarà concatenato con il salt) da condividere per decifrare il messaggio.

3.3.2 Fase di decriptazione

La fase di decrittazione è molto più semplice e veloce rispetto a quella inversa, infatti l'utente inserisce in input il seed pubblico in *base64* e la password a sua conoscenza. Tramite il processo inverso, avviene la decrittazione con *AES* del seed pubblico, ottenendo una lista di 128 numeri che variano tra 0 e 250. Quindi avverrà la mappatura per ottenere una stringa di 128 numeri.
Questo sarà il nostro seed da cui recupereremo la struttura e gli indici dei vocaboli. Utilizzando una password diversa da quella originale, *AES* otterrà chiaramente una lista di numeri diversa da quella originale, che corrisponderanno a un seed diverso: qui si ha l'Honey Encryption, il flow è il medesimo sia per la password corretta che per quella scorretta, semplicemente sfruttando la cifratura a blocchi, si otterrà una diversa combinazione numerica.
Dal seed ottenuto, sappiamo che i primi quattro numeri corrispondono alla struttura grammaticale, che ci indica il tipo e il numero di parole componenti, permettendo di dividere le 124 cifre nell'opportuno numero di parti.
A questo punto, per ogni "parte" verrà fatto il mod con la lunghezza del relativo dizionario, ottenendo l'indice dell'elemento della frase.
Ora, attraverso la struttura tokenizzata e il lemma delle parole, con opportune regole, verrà ricostruita la frase. Quest'ultima parte, insieme a quella iniziale corrispondente, rappresenta il punto più delicato dell'intero progetto.

3.4 Spiegazione del codice

3.4.1 Importazione delle librerie

Ovviamente, le prime righe dello script saranno dedicate all'importazione delle librerie necessarie per il corretto funzionamento del programma.

```
1 import spacy
2 import inflect
3 import argparse, csv
4 from Crypto.Protocol.KDF import PBKDF2
5 from Crypto.Cipher import AES
6 import secrets
7 import base64
```

```
8 from pattern.en import conjugate, article
```

Listing 3.1: Import librerie

Come già precedentemente introdotto, spacy, inflect e pattern.en (con conjugate e article) sono relative al Natural Language Processing. Mentre secrets, AES di Crypto.Cipher, PBKDF2 di Crypto.Protocol.KDF e base64 per la crittazione e la codifica del messaggio con password.

Argparse e CSV sono moduli di sistema per, rispettivamente, aggiungere opzioni da linea di comando attraverso gli argomenti e leggere-scrivere i file di tipo csv.

3.4.2 Variabili globali

Per evitare ridondanza e per fattori di comodità, sono state definite delle variabili globali.

```
10 p = inflect.engine()
11 lengthW = 4
12 nseed = 31*lengthW + lengthW
13 secretsGenerator = secrets.SystemRandom()
```

Listing 3.2: Dichiarazione variabili globali

Alla riga 10 abbiamo la dichiarazione di **p**, che è l'*engine di inflect*, che, grazie alle sue funzioni, sarà responsabile, come vedremo, di pluralizzare gli aggettivi e i sostantivi, data la versione al singolare.

Le righe 11 e 12, invece sono variabili riferite alla lunghezza totale del nostro seed numerico, nel mio caso ipotizzando un numero massimo di trentuno parole per una frase di tipo semplice, ognuna della quale avrà minimo 4 cifre dedicate, ho preferito avere un *seed di 128 numeri*, 4 per l'ID della struttura e 124 per indicare gli id dei vocaboli.

Come per l'altra libreria, anche **secretGenerator** è un *generatore del modulo secrets* necessario per ottenere caratteri random.

Apertura dizionari

Successivamente, avviene l'apertura di tutti dizionario *.csv* e viene creata una variabile globale di supporto, dove salvare il contenuto (sotto forma di lista) per evitare di ripetere richieste di accesso ai file, per ogni operazione.

```
15 #OPEN DICTIONARY
16 with open('ADJ.csv', 'r') as f:
17     reader = csv.reader(f)
18     ADJ = []
19     for row in reader:
```

```
20          ADJ.append(row[0])
```

Listing 3.3: Apertura dizionari .csv

La sequenza di operazioni è la medesima per ogni dizionario, escluso quello delle strutture (mostrato al Listing 3.4). Viene sfruttata l'istruzione di python *"with"* che permette di gestire semplicemente l'accesso ai nostri file, chiudendoli automaticamente al termine dell'operazione. Come accennato, solamente per il dizionario delle strutture, è stato utilizzata una procedura più complessa per la sua gestione.

```
106 with open('STR.csv') as f:
107     reader = csv.reader(f)
108     STR=[]
109     for row in reader:
110         miniSRT = []
111         for col in row:
112             if col != "":
113                 miniSRT.append(eval(col))
114         STR.append(miniSRT)
```

Listing 3.4: Apertura dizionario strutture

Essendo composto a sua volta da liste, è stato necessario aggiungere un ulteriore controllo per il corretto salvataggio all'interno della nostra lista di supporto *STR*

3.4.3 Funzioni

Le funzioni necessarie in fase di criptazione sono:

- NdigitRandomNumber 3.4.3: restituisce un numero random di n numeri.
- getIndex 3.4.3: restituisce l'indice del dizionario della parola ricevuta in input.
- getLenDictionary 3.4.3: ottiene la lunghezza del dizionario richiesto.
- getIndexStr 3.4.3: restituisce l'indice del dizionario di strutture grammaticali della struttura richiesta.
- divideNParts 3.4.3: divide una stringa in n parti uguali.
- generateSeedWords 3.4.3: restituisce il seed composto dagli indici delle parole, da criptare e inviare.

Mentre in fase di decrittazione si andrà ad utilizzare:

- getLemmaByIndex 3.4.3: recupera il lemma dal dizionario in base all'indice inserito.

- getLemma 3.4.3: richiama opportunamente la funzione getLemmaByIndex.
- beConjPast 3.4.3: coniuga il verbo essere (to be) al passato.
- getWord 3.4.3: dalla lista dei lemmi e della struttura grammaticale, restituisce la corretta forma.

NdigitRandomNumber - Generazione numero random di N numeri

La funzione *NdigitRandomNumber* ha il compito di restituire in output un numero random, ricevendo in input il quantitativo di cifre corrispondenti.

```
116 #Get random numner of n string
117 def NdigitRandomNumber(n):
118     i = 0
119     number = ''
120     while i < n:
121         number += str(secretsGenerator.randint(0,9))
122         i = i + 1
123     return number
```

Listing 3.5: Funzione NdigitRandomNumber

Un semplicissimo ciclo while, operato sulla variabile di input **n**, che aggiunge a una stringa, inizialmente vuota, n numeri ottenuti dalla funzione **randInt** in un range 0-9 richiamata dalla variabile di classe **secretsGenerator** precedentemente dichiarata (vedi 3.2), che restituisce realmente numeri random, a differenza della nota libreria omonima, che invece è pseudo-random.

getIndex - Ottenere l'indice del vocabolo all'interno del suo dizionario

Una delle funzioni principali all'interno dello script è *getIndex*, infatti, ricevendo in input il testo tokenizzato della frase (doc) e l'indice i, indicante il termine, recupera da doc il POS TAG e DEP di i che va a indicare il dizionario nel quale cercare il lemma, successivamente effettua la ricerca e restituisce in output l'id indice all'interno dello stesso.

La suddetta funzione, riceve in input **doc**, che non è altro che una *lista di token*, ottenuta dal parsing della frase originale da parte di spaCy. L'indice i indica ovviamente i-esimo elemento della lista.

```
126 def getIndex(doc, i):
127     type = doc[i].tag_
128     dep = doc[i].dep_
129     lemma = doc[i].lemma_
130     text = doc[i].text
```

Listing 3.6: Funzione getIndex - Variabili

Per recuperare il corretto dizionario da quale prelevare l'index della i-esima parola è necessario ricavare dal i-esimo token, alcune informazioni:

- *type* - indica il POS TAG della parola nella frase (es. NOUN).
- *dep* - corrisponde al tipo di dipendenza DEP che sussiste tra il lemma ricercato e il lemma da cui dipende (es. nsubj).
- *lemma* - si riferisce al lemma del termine (es.eating - eat).
- *text* - si riferisce al termine come è presente nel testo (es. eating).

Tra la riga 133 e 199 vengono effettuate le varie scelte, in base alla combinazione di **type**, **dep**, **lemma** e **text**. Se per esempio lemma type indica un verbo, non si cercherà sempre il relativo indice all'interno del dizionario dei verbi.

```
133     if type == 'VB' or type == 'VBD' or type == 'VBG' or
           type == 'VBN'or type == 'VBP' or type == 'VBZ':
134         if dep != "auxpass" and dep != 'aux':
135             return VERB.index(lemma)
136         else:
137             return NdigitRandomNumber(lengthW)
```

Listing 3.7: Funzione getIndex - Condizione if Verbi

Se il verbo è al passivo, dunque sussiste la dipendenza *"aux"* o *"auxpass"* allora si sa che si tratta del verbo essere, in quel caso si restituirà un numero random.

Un'altra casistica è rappresentata dai pronomi personali PRP, infatti in base all'eventuale presenza di una dipendenza che va a indicare un ruolo come soggetto all'interno della frase, si cercherà il lemma all'interno del relativo file

```
164     elif type == 'PRP':
165         if dep == 'nsubj' or dep == 'nsubjpass':
166             return PRP_S.index(text.lower())
167         else:
168             return PRP_O.index(text.lower())
```

Listing 3.8: Funzione getIndex - Condizione if Pronomi Personali

Similmente, questi passaggi avvengono anche per tutti gli altri POS TAG.

```
181     elif type == 'DT':
182         if lemma in DET_PLUR:
183             doc[i].tag_ = 'DTP'
184             return DET_PLUR.index(lemma)
185         elif lemma in DET_SING:
186             if lemma == 'an':
187                 lemma = 'a'
188             doc[i].tag_ = 'DTS'
189             return DET_SFG.index(lemma)
190         return DET.index(lemma)
```

Listing 3.9: Funzione getIndex - Condizione if Articoli Determinativi

Il risultato dell'output return, sarà una stringa numerica, indicante la posizione (l'indice) del termine all'interno della specifica lista precedentemente costruita, partendo dal relativo dizionario (viene usata la funzione standard **index(int i)** delle liste in Python).

getLenDictionary - Ottieni la lunghezza di un dizionario

La funzione *getLenDictionary*, riceve in input una tupla che corrisponde a un elemento della lista della struttura, della quale avete già visto la struttura nella sezione **??**, paragrafo **??**.

```
198 def getLenDictionary(tuple):
199
200     pos_tag = tuple[1]
201     dep = tuple[2]
```

Listing 3.10: Funzione getLenDictionary - Variabili

Similmente a *getIndex*, anche qua vengono usate delle variabili di supporto:

- *pos_tag* - indica il POS TAG della parola nella frase (es. NOUN).
- *dep* - corrisponde al tipo di dipendenza DEP che sussiste tra il lemma ricercato e il lemma da cui dipende (es. nsubj).

Anche per questa funzione, per determinare la lunghezza del dizionari i passaggi e le condizioni di if sono le medesime di *getIndex*.

```
214     elif pos_tag == 'NNU':
215         return len(NOUN_U)
216     elif pos_tag == 'MD':
217         return len(MD)
```

```
218     elif pos_tag == 'CC':
219         return len(CC)
```

Listing 3.11: Funzione getLenDictionary - Condizione if

Per recuperare la lunghezza del dizionario, si applica la funzione python **len**, sulla lista precedentemente dichiarata associata al dizionario.

getLemmaByIndex - Ottenere il lemma dall'indice

Si tratta della funzione complementare di *getIndex*, ma a differenza di quest'ultima, *getLemmaByIndex* risulta molto più semplice.

```
253 def getLemmaByIndex(index, list):
254     return list[int(index)%len(list)]
```

Listing 3.12: Funzione getLemmaByIndex

Riceve in input due argomenti:

- *index* - che è un numero ed è relativo all'indice (posizione) dentro al dizionario del lemma ricercato.
- *list* - è la lista associata al dizionario dove cercare il lemma.

L'output è banalmente l'elemento della lista corrispondente.

L'operazione modulo tra index e la lunghezza della lista è necessaria, perchè index non indica direttamente l'indice, ma rappresenta il numero del seed (convertito in intero, visto che viene condiviso come stringa), che diviso per l'ampiezza del dizionario, restituisce l'indice del termine.

getLemma - Ottenere il lemma dal seed

GetLemma è una delle funzioni principale utilizzata in fase di decriptazione, infatti permette di recuperare i lemma che compongono la frase, partendo dal seed e dalla struttura grammaticale.

```
256 def getLemma(index, type, dep, parentType):
```

Listing 3.13: Funzione getLemma

Gli elementi in input vengono ricavati dal seed decodificato e dalla struttura grammaticale:

- *index* - indice del lemma da ricercare, ricavato dal seed.
- *type* - POS TAG corrispondente al lemma indicizzato, determinato direttamente dalla struttura grammaticale.

- *dep* & *parentType* - vedi sopra

In base alla combinazione dell'**type**, **dep** e **parentType**, verrà ricercato all'interno del corretto dizionario l'elemento corrispondente all'**index**, richiamando la funzione *getLemmaByIndex* 3.4.3.

```
257     if type == 'VB' or type == 'VBD' or type == 'VBG' or
           type == 'VBN'or type == 'VBP' or type == 'VBZ':
258         if dep != "auxpass":
259             if dep != 'aux':
260                 return getLemmaByIndex(index, VERB)
261             elif dep == 'aux':
262                 if parentType == 'VBG':
263                     return 'be'
264                 elif parentType == 'VBN':
265                     return 'have'
266                 else:
267                     return 'do'
268         else:
269             return "be"
270     elif type == 'NNP' or type == 'NNPS':
271         return getLemmaByIndex(index, PROPN)
272     elif type == 'NNU':
273         return getLemmaByIndex(index, NOUN_U)
```

Listing 3.14: Funzione getLemma - Condizione if

Le condizioni di if, seguono a specchio quelle precedentemente illustrate.

beConjPast - Coniugare il verbo essere (to be) al passato

La funzione di supporto *beConjPast* ha il compito di coniugare il verbo essere al passato (was/were) in base al soggetto da cui dipende.

```
316 def beConjPast(struct, list_lemma):
```

Listing 3.15: Funzione beConjPast

Riceve dunque in input due elementi, la struttura e la lista dei lemmi, da cui ricaverà il sostantivo/pronome soggetto.

```
317     nsubj_pos = 0
318     i = 0
319     for tupla in struct:
320         if tupla[1] == 'nsubj' or tupla[1] == 'nsubjpass':
321             nsubj_pos = i
322             break
```

```
323             i = i + 1
324         typeSubj = struct[nsubj_pos][1]
325         lemmaSubj = list_lemma[nsubj_pos]
326         verb = ''
```

Listing 3.16: Funzione beConjPast - Soggetto

Viene innanzitutto effettuato un ciclo di ricerca all'interno della nostra struttura per cercare la tupla, indicante il token, con il ruolo di soggetto e ne salva la posizione all'interno della frase. Successivamente viene recuperato il lemma corrispondente nella relativa lista, grazie alla variabile **nsubj_pos** appena in stanziata.

In seguito, avvengo le condizioni if clause, per determinare se applicare la funzione **conjugate()** di pattern.en usando l'attributo *'1sgp'* (restituisce was) oppure *'p'* (were).

```
328         if (typeSubj == 'PRP' and (lemmaSubj == 'i' or lemmaSubj
               == 'he' or lemmaSubj == 'it' or lemmaSubj == 'she'))
               or (typeSubj == 'NNP' or typeSubj == 'NN'):
329             verb = en.conjugate('be', '1sgp')
330         else:
331             verb = en.conjugate('be', 'p')
332         return verb
```

Listing 3.17: Funzione beConjPast - Condizioni

Ovviamente, se il soggetto è un pronome singolare oppure un nome al singolare, il verbo ottenuto sarà *was*, altrimenti *were*.

getWord - Restituisce il termine correttamente formulato

L'altra fondamentale funzione che entra in gioco nella fase di decrittazione è *getWord*, che lavorando in simbiosi con *getIndex* **??**, restituisce il termine, correttamente declinato o coniugato, da inserire all'interno della frase per ottenere un messaggio grammaticalmente corretto.

```
334 def getWord(lemma, struct, pos, list_lemma):
335     word = lemma
336     token = struct[pos]
337     index_token = struct[pos][0]
338     tag_token = struct[pos][1]
339     dep_token = struct[pos][2]
340     index_parent_token = struct[pos][3]
341     type_parent_token = struct[pos][4]
```

Listing 3.18: Funzione getWord - Variabili

I valori di input sono numerosi, per il semplice fatto che a *getWord* è richiesto di formulare correttamente la parola, in base al ruolo che possiede nella relativa frase.

Le variabili utilizzate diventano quindi le seguenti:

- *word* - corrisponde inizialmente al lemma inserito in input.
- *token* - il token del termine (vedi lemma) all'interno della struttura alla posizione data.
- *index_token* - la posizione corrispondente.
- *tag_token* - il POS TAG relativo al lemma nel token.
- *dep_token* - il tipo di dipendenza che intercorre con il suo token padre, ricavabile dal token.
- *index_parent_token* - la posizione del token padre all'interno della frase.
- *type_parent_token* - il POS TAG del token padre.

In base a questi valori e la loro combinazione, è possibile ricostruire la frase, termine per termine.

```
343     if tag_token == 'NNPS' or tag_token == 'NNS':
344         word = p.plural_noun(lemma)
345     elif tag_token == 'JJ' and (type_parent_token == 'NNPS'
           or type_parent_token == 'NNS'):
346         word = p.plural_adj(lemma)
```

Listing 3.19: Funzione getWord - NOUN e ADJ

Per i sostantivi e gli aggettivi al plurale, per effettuare la corretta pluralizzazione, vengono utilizzate due funzioni dell'engine inflect, **plural_noun** e **plural_adj**, che, come prevedibile, dato il lemma restituiscono la corretta declinazione al plurale.

```
347     elif tag_token == 'DTS' and lemma == 'a':
348         word = article(list_lemma[index_parent_token])
349     elif tag_token == 'DTP' and lemma == 'many':
350         if type_parent_token == 'NNU':
351             word = 'much'
```

Listing 3.20: Funzione getWord - Articoli DET

Invece, per ottenere il corretto articolo Determinativo singolare, viene prelevato il termine da cui dipende, si effettuano dei controlli e viene utilizzata la funzione **article** di pattern.en, per determinare la versione migliore, Mentre, per quelli al plurale, essendoci il "problema" dei sostantivi uncountable, si attuano delle operazioni *manuali*.

Riguardo i verbi, per tutte le opzioni esclusi i verbi al Past Tense (VBD), si applica la funzione conjugate, già presentata, con le corrette opzioni.

```
352     elif tag_token == 'VBG':
353         word = conjugate(lemma, 'part')
354     elif tag_token == 'VBN':
355         word = conjugate(lemma, 'ppart')
356     elif tag_token == 'VB':
357         word = lemma
358     elif tag_token == 'VBD':
359         if lemma == 'be':
360             word = beConjPast(struct, list_lemma)
361         else:
362             word = conjugate(lemma, 'p')
363     elif tag_token == 'VBP':
364         word = conjugate(lemma, '1sg')
365     elif tag_token == 'VBZ':
366         word = conjugate(lemma, '3sg')
367 
368     return word
```

Listing 3.21: Funzione getWord - VERBI

Per i VBD, viene prima effettuato un controllo sulla presenza del verbo essere ed eventualmente viene applicata la funzione *beConjPast* 3.4.3.

Mentre, per tutte le altre opzioni, non vengono effettuate modifiche al lemma, visto che non sono necessarie.

getIndexStr - Ottieni l'indice della struttura

La funzione getIndexStr è la funzione parallela a getIndex 3.4.3 dei vocaboli, ma per le strutture.

Come già accennato, esiste un dizionario delle strutture delle frasi, che a loro volta sono divisibili in "elementi" uno per ogni parola del testo.

```
371 def getIndexStr(list):
```

Listing 3.22: Funzione getIndexStr

La lunghezza del codice è decisamente minore, inoltre, a differenza di quanto accade con *getIndex* dove viene restituito il numero 0, se la struttura non è presente nella lista, questa viene aggiunta e si scrive il file .csv, questo serve per permettere di ampliare il dizionario con nuove strutture.

In input riceve chiaramente solamente un valore, **list**, che corrisponde all'attuale lista delle strutture, dalla quale ricavare l'index.

```
372     if list not in STR:
373         STR.append(list)
374         with open("STR.csv",'a', newline='') as f:
375             wr = csv.writer(f)
376             wr.writerow(list)
```

Listing 3.23: Funzione getIndexStr - Aggiunta delle strutture

Se l'attuale lista non è presente in STR, che rappresenta il dizionario delle strutture, si scrive, con l'opzione *'a' ('append')* per non sovrascrivere completamente, ma solamente aggiungere un elemento, sul nostro file STR.csv.

Successivamente, verrà restituita la posizione che tale lista occupa nella lista, questo sarà il nostro Index.

```
372     return str(STR.index(list)).zfill(lengthW)
```

Listing 3.24: Funzione getIndexStr - Output

Come si nota, è necessario applicare la funzione **.zfill**, che come prevedibile aggiunge '0' alla stringa (da dx vs sx), se l'id non ha la lunghezza richiesta, in questo caso *lenghtW (4)*.

divideNParts - Divisione stringa numerica in n parti

Le successive due funzioni sono indispensabili per la codifica e criptazione del messaggio.

La prima, *divideNParts*, riceve in input una stringa e un numero n, dividendo il testo in n parti uguali, quindi restituendone una lista derivata, composta da n elementi, spezzettando la stringa in n parti.

```
380 def divideNParts(string, n):
381     rest = len(string)%n
382     length_part = int(len(string)/n)
383     parts = []
384 
385     i = 0
386     j = 1
387     start = 0
388     stop = length_part
```

Listing 3.25: Funzione divideNParts - Variabili

Le variabili utilizzate hanno un nome auto-esplicativo:

- *rest* - resto della divisione tra la lunghezza della stringa e n.

- *length_part* - dimensione di una singola parte ni, ottenuta dalla divisione (senza resto) della lunghezza del testo con n.

- *parts* - è la lista, inizialmente vuota, di sotto-stringhe.

- *i, j, start e stop* - sono variabili si supporto per operare la divisione.

Verrà operata un'operazione di divisione, con l'obiettivo di ottenere **n** parti uguali, se questo non sarà possibile, perché la lunghezza della stringa non sarà divisibile per **n**, alcuni parti avranno un carattere in più.

Per eseguire l'operazione di divisione, verrà eseguito un ciclo **n** volte, per ottenere **n** parti.

```
390     while i < n:
391         i = i + 1
```

Listing 3.26: Funzione divideNParts - ciclo

Oltre a incrementare **i**, per il ciclo successivo, viene controllato il valore di **rest**, se questo sarà uguale a 0 (o inferiore), anche j sarà 0.

```
392         if rest <= 0:
393             j = 0
```

Listing 3.27: Funzione divideNParts - Controllo del resto

Questo passaggio è fondamentale, perché, finché **j** non sarà pari a 0, durante la segmentazione di **s**, prenderò un carattere in più rispetto a **lenght_part**.

La variabile stop, che inizialmente è uguale a se stessa sommata con j, indica il limite dove effettuare la substring, attraverso la funzione *[start:stop]*, della nostra stringa.

```
394         stop = stop + j
395         parts.append(string[start:stop])
396         start = stop
397         stop += length_part
398         rest =  rest - 1
```

Listing 3.28: Funzione divideNParts - Substring

Ovviamente, **stop** a ogni ciclo dovrà essere incrementato del suo stesso valore, mentre il nuovo **start** sarà il vecchio **stop**.

Contemporaneamente, **rest** verrà diminuita di un'unità, per 3.27.

generateSeedWords - Generarazione del seed da crittare

L'altra fondamentale funzione è generateSeedWords, sicuramente la più complessa, dal punto di vista logico, dell'intero programma.

Infatti *generateSeedWords* produce il seed, stringa numerica contenente riferimenti alla struttura della frase e ai vocaboli, che verrà successivamente criptata, prima di essere condivisa.

```
403 def generateSeedWords(number, idstruct, list, nelement,
    list_struct):
```

Listing 3.29: Funzione generateSeedWords - Dichiarazione

In input riceve un **number**, che non è altro che il numero random generato, id della struttura e la lista degli id dei termini, il numero di termini che compongono la frase e infine la struttura vera e propria.

```
405     numberword = number[lengthW:]
406     numberstruct = number[0:lengthW]
```

Listing 3.30: Funzione generateSeedWords - Variabili part1

Le prime variabili dichiarate e instanziate, sono quelle relative alla struttura generica:

- *numberword* - stringa estrapolata dal numero random indicante gli id dei vocaboli e relativa ai successivi 124 caratteri, tolti i primi quattro, che saranno specifici della struttura.
- *numberstruct* - primi quattro caratteri numerici indicanti il numero random di partenza per costruire l'index della struttura.

Per determinare l'index relativo alla struttura della frase, sono richieste delle semplici operazioni.

Innanzitutto si ottiene la variabile **restStruct**, cioè il resto ottenuto da **intRandStruct** (conversione di **numberstruct** in intero) con la lunghezza del dizionario STR (similmente a quanto avviene con *getLemma* 3.4.3).

Il significato delle altre variabili è immediato. L'utilità di **restStruct** è dovuta dalla necessità di conoscere l'offset di partenza, dalla qual si ricavera l'**index**.

```
408     intStruct = int(idstruct)
409     intRandStruct = int(numberstruct)
410     structDicLen = len(STR)
411     restStruct = intRandStruct%structDicLen
412     if restStruct != intStruct:
413             dif = intStruct - restStruct
```

```
414             numberstruct = str(intRandStruct+dif)
415             numberstruct = numberstruct.zfill(lengthW)
```

Listing 3.31: Funzione generateSeedWords - Seed Struttura

Se **restStruct** sarà pari all'indice di struttura, non si dovranno effettuare modifiche al numero indicante la struttura, altrimenti, bisognerà sommare a **intRandStruct** la differenza tra questi due valori (e poi eventualmente aggiungere zero fino per arrivare alla lunghezza richiesta).

```
416             if len(numberstruct) > lengthW:
417                 numberstruct = str(int(numberstruct) -
                        structDicLen)
418             numberstruct = numberstruct.zfill(lengthW)
```

Listing 3.32: Funzione generateSeedWords - Seed Struttura #2

Se però, aggiungendo la differenza, il numero ottenuto dovesse superare il limite, allora, per non modificare il meccanismo di resto che restituisce l'index, sarà necessario sottrarre dal seed della struttura, la lunghezza del dizionario, in modo da rendere invariato il risultato della operazione modulo con la stessa.

Anche per **numberword**, che consisterà nella concatenazione degli indici delle parole, le operazioni sono simili a quanto già precedentemente presentato, ma dovranno essere ripetute per tutte le **n** parti, cioè per tutti i termini della frase.

```
420     rest = len(numberword)%nelement
421     length_part = int(len(numberword)/nelement)
422     parts = divideNParts(numberword, nelement)
423
424     i = 0
```

Listing 3.33: Funzione generateSeedWords - Seed Parole #Variabili

Le variabili di supporto hanno il medesimo ruolo, **parts** è la lista ottenuta richiamando la funzione *divideNParts* (3.4.3), passandogli come argomento **numberword** e **nelement**, ottenendo un array di sotto-stringhe di simil dimensione della nostra sequenza numerica random.

```
426     for word in list:
427         length_word_part = len(parts[i])
428         intWord = int(word)
429         intPart = int(parts[i])
430         wordDicLen = int(getLenDictionary(list_struct[i]))
431         restWord = intPart%wordDicLen
432         if restWord != intWord:
433             dif = intWord - restWord
```

```
434             parts[i] = str(intPart+dif)
435             parts[i] = parts[i].zfill(length_word_part)
436             if len(parts[i]) > length_word_part:
437                 parts[i] = str(int(parts[i]) - wordDicLen)
438             parts[i] = parts[i].zfill(length_word_part)
439         i = i + 1
```

Listing 3.34: Funzione generateSeedWords - Seed Parole #Ciclo

I passaggi sono i medesimi, tranne che verranno ripetuti per tutti gli **n** elementi componenti la lista **parts**.

Qua, il riferimento all'id della parola è contenuto nella lista **list**.

```
441     numberwords = ''.join(parts)
442
443     return (numberstruct + numberwords)
```

Listing 3.35: Funzione generateSeedWords - Return

Infine, verranno concatenati e verrà restituita la stringa ottenuta sommando numberseed e numberwords.

encryption

La funzione encryption richiede in input due valori, il testo del messaggio da inviare (**text**) e la password che si vuole utilizzare. Come prevedibile si occupa dell'effettiva codifica e criptazione, dunque della fase numero uno, quella richiesta dal mittente.

```
445 def encrypt(text, password):
```

Listing 3.36: Encryption

A differenza della altre funzioni, non restituisce un valore, ma si "limita" a stampare a video il risultato finale della criptazione, dunque il seed pubblico da condividere con il destinatario (seed composto anche con il valore di salt, utilizzato per derivare la key di AES). La prima operazione che viene eseguita è quella di caricare e preparare il parser spaCy con un dizionario, in questo caso quello quello inglese *"small"*, è necessario poichè questi database non sono altro che modelli linguistici utilizzati come base per riconoscere e parsare le frasi.

```
446     nlp = spacy.load('en_core_web_sm')
447     doc = nlp(text)
```

Listing 3.37: Encryption - Parser testo

Il passaggio seguente, consiste nel parsare il testo ricevuto in input, salvandolo in una variabile denominata **doc**.

Vedesi appendice A.8.
Si dichiarano anche delle variabili di supporto:

- *lista_parole* che conterrà gli index delle parole
- *lista_struttura* che sarà formata di token

```
449     lista_struttura = []
450     lista_parole = []
451     number_element = 0
452     i = 0
453
454     randomnumber = NdigitRandomNumber(nseed)
```

Listing 3.38: Encryption - Variabili

Inoltre, viene generato il numero **randomnumber** con la funzione *NdigitRandomNumber* 3.4.3, che riceverà in input la lunghezza richiesta del seed, cioè 128 cifre.

```
456     for token in doc:
457         number_element = number_element + 1
458         index = getIndex(doc, i)
459         lista_parole.append(index)
460         lista_struttura.append((token.i, token.tag_,
                token.dep_, token.head.i, token.head.tag_))
461         i = i + 1
```

Listing 3.39: Encryption - Tokenizzazione

Successivamente, per ogni token all'interno di **doc**, che come ricordo contiene il testo parsato (e dunque tokenizzato, vedesi appendice A.8 per la spiegazione) verranno estratte alcune informazioni contenute nel token, uno per ogni termine: la posizione all'interno della frase, il POS TAG, il tipo di dipendenza, la posizione dell'elemento padre, il POS TAG dell'elemento padre.
Queste verranno salvate all'interno della **lista_struttura**. Contemporaneamente, per ogni parola, verrà richiamata la funzione *getIndex* 3.4.3 e l'**index** verrà salvato all'interno della lista_parole.

Ora, dopo aver riempito le due liste, ci si occuperà della generazione del seed, ma non prima di aver richiamato la funzione *getIndexStr* 3.4.3 per ottenere anche l'index della struttura.

```
463     idstruttura = getIndexStr(lista_struttura)
464     randomnumber = generateSeedWords(randomnumber,
           idstruttura, lista_parole, number_element,
           lista_struttura)
```

Listing 3.40: Encryption - Generazione seed

Solamente a questo punto, potrà essere richiamata la funzione generate *SeedWords* 3.4.3 con le due liste, il numero random, l'id di struttura e il numero di parole nella frase, per ottenere il nostro seed numerico, che dovrà essere criptato.

Prima di andare a criptare il seed numerico con AES, è opportuno derivare dalla password inserita dall'utente, una **key** tramite una *key derivation function (KDF)* A.10, eseguendo uno key stretching per aumentare la sicurezza della chiave.

```
469     salt = secrets.token_bytes(8)
470     key = PBKDF2(password, salt,32,1000)
```

Listing 3.41: Encryption - PBKDF2

Avviene contemporaneamente una salazione, nel nostro caso con un salt random di 64 bits, al fine di ottenere key diverse data la medesima password, per evitare l'utilizzo di "key" più frequenti. Il valore di salt deve essere pubblico, altrimenti non sarà possibile derivare in fase di decrittazione, verrà infatti convertito in base64 e inviato concatenato al seed.

Per generare il salt, si sfrutta ancora una funzione di secrets, **token_bytes**, per creare un token di 8 bytes random.

```
472     ENC=AES.new(key, AES.MODE_ECB)
473     seedAES = []
```

Listing 3.42: Encryption - AES

Oltre a istanziare **seedAES**, che costituirà il seed criptato, è necessario configurare l'algoritmo di cifrazione di AES, con la nostra **key** e la modalità di permutazione a blocchi (vedi appendice A.9).

Prima di effettuare la vera e propria cifratura con AES, è opportuno, oltre a convertire la stringa numerica in una lista cifra per cifra (salvata nella variabile **seedAES**), effettuare una procedura di "disturbo"-rumore con l'utilizzo di *nonce*, numero pseudo-casuale con un'unico utilizzo.

```
475     for digit in seed:
476         seedAES.append (int(digit))
477
478     seedAES=list(map(lambda x:
           x+10*secrets.randbelow(24),seedAES))
```

Listing 3.43: Encryption - Seed nonce

L'obbiettivo è il medesimo della generazione iniziale di un seed randomico, rimuovere il più possibile legami logici-matematici tra i numeri e impedire attacchi di predizione con l'analisi della variazione dell'output al variare della password.

Per far questo, si applica una funzione lambda su tutti gli elementi di **seedAES** (per questo è stato comodo convertire la stringa in una lista), sommandogli un

numero multiplo di 10 compreso tra 0 e 240 (nella fase di decriptazione verrà eseguita l'operazione modulo 10).

```
483     arr = bytes(seedAES)
484     outText=base64.b64encode(ENC.encrypt(arr)).decode()
485     print("The␣public␣seed␣is:")
486     print(base64.b64encode(salt).decode() + outText)
```

Listing 3.44: Encryption - Criptazione

Infine, si applica sia la funzione di criptazione AES sulla lista (convertita da decimali a bytes) che la codifica in base64, per ottenere il nostro seed pubblico da condividere.
Questo seed viene concatenato con il salt convertito in base64 e stampato a video.

decrypt

L'altra fondamentale funzione è *decrypt*, che come si può intuire, opera la decrittazione e decodifica di un seed, restituendo una frase.

```
489 def decrypt(publicSeed, password):
```

Listing 3.45: Decrypt

Questa funzione, richiamata dall'utente destinatario attraverso l'opzione '-d', riceve in input un **publicSeed** (ottenuto dalla fase di crittazione 3.4.3) e una **key**, la password usata per decrittare la stringa.

Come avviene in criptazione, anche qui è necessario generare l'algoritmo di crittazione, con la password inserita e ovviamente lo stesso metodo di permutazione (MODE_ECB A.9).

Prima chiaramente è opportuno estrapolare dall'input il nostro salt (primi dodici caratteri) e il nostro public seed.

```
491     salt = publicSeed[0:12]
492     pseed = publicSeed[12:]
```

Listing 3.46: Decrypt - Decrittazione salt e public seed

Poi per ottenere la key da usare per istanziare AES, si eseguono i passaggi inversi, richiamando sempre la funzione *PBKDF2* A.10 con il salt convertito in bytes.

```
494     salt = base64.b64decode(salt.encode())
495     key = PBKDF2(password, salt,32,1000)
496
497     ENC = AES.new(key, AES.MODE_ECB)
```

Listing 3.47: Decrypt - Decrittazoone

Successivamente avviene la decrittazione del nostro public seed, dopo essere stato decodificato dalla codifica in base64. Ottenendo come risultato un array contenente, nel nostro caso, 128 numeri.

```
499     arrayseed = list(map(lambda x: x%10,decoded))
500     seed = ''.join(map(str, arrayseed))
```

Listing 3.48: Decrypt - Conversione in seed numerico

Si esegue l'operazione lambda inversa a quella lanciata in fase di criptazione, per ogni elemento della lista, si opera il modulo di 10, e la si converte in una stringa, cosi' da ottenere una sequenza numerica di 128 cifre, dal quale ricaveremo gli indici.

Dalle prime quattro cifre, che come abbiamo visto in fase di generazione del seed, costituiscono l'index della struttura nel relativo database, si recupererà la struttura grammaticale della nostra frase.

```
515     idstruttura = seed[0:lengthW]
516     idstruttura = str(int(idstruttura)%len(STR))
517     struct = STR[int(idstruttura)]
```

Listing 3.49: Decrypt - Struttura

L'id della nostra struttura verrà quindi salvato in una variabile **idstruttura**. A questo punto, dalla lista STR si preleva la corretta struttura tokenizzata.

Procedura simile, ma ovviamente ripetuta per tutte le n parole componenti la frase, verrà seguita per ottenere l'indice dei vocaboli costituenti il nostro testo.

Dopo aver preso i successivi 124 caratteri numerici dalla sequenza, verranno divisi in n parti, come avveniva con le altre funzioni sopra descritte.

```
511     seedparole = seed[lengthW:]
512     parts = divideNParts(seedparole, len(struct))
513
514     lista_lemma = []
515     lista_parole = []
```

Listing 3.50: Decrypt - Indici lemmi

In seguito, dichiarate due liste, la prima che conterrà la lista dei lemmi, la seconda le parole, inizierà il ciclo for sugli **n** elementi di **parts**.

```
516
517     i = 0
518     for indexword in parts:
519         lemma = getLemma(indexword, struct[i][1],
               struct[i][2], struct[i][4])
520         lista_lemma.append(lemma)
521         i = i + 1
```

Listing 3.51: Decrypt - Lista lemmi

Per ogni stringa numerica i **parts**, si richiamerà la funzione *getLemma* 3.4.3 per ottenere il lemma, che sarà salvato nella lista precedentemente dichiarata, **lista_lemma**.

A questo punto, si opererà un ciclo su tutti i lemmi della lista appena istanziata, richiamando la funzione getWord 3.4.3, per declinare e coniugare correttamente la parola.

```
523     i = 0
524     lista_parole = lista_lemma
525     for lemma in lista_lemma:
526         word = getWord(lemma, struct, i, lista_lemma)
527         lista_parole[i] = word
528         i = i + 1
```

Listing 3.52: Decrypt - Lista parole

Infine, da **lista_parole** si genererà e stamperà in output il testo, separando ogni parola con il carattere 'spazio'.

```
530     frase = '␣'.join(lista_parole)
531     frase= frase[:1].upper() + frase[1:]
532
533     print(frase)
```

Listing 3.53: Decrypt - Output

Grazie a questa sequenza di criptazione-decrittazione, inserendo una password diversa da quella originaria, si otterrà comunque un stringa numerica, che, grazie alle regole grammaticali inserite nelle funzioni *getWord* e *getLemma*, determinerà un testo plausibile, corretto grammaticalmente.

3.4.4 Main

Come tutti i programmi, anche questo script presenta un main, nel quale, richiamato il modulo **argParser** 3.2.2,

```
536 if __name__ == '__main__':
537     parser = argparse.ArgumentParser()
538     parser.add_argument("-e", "--encrypt", nargs=2,
539                         type=str,
540                         help="encrypt␣text")
541     parser.add_argument("-d", "--decrypt",
542                         nargs=2, type=str,
543                         help="decrypt␣code")
```

```
544     args = parser.parse_args()
545
546     if args.encrypt:
547         encrypt(args.encrypt[0], args.encrypt[1])
548     elif args.decrypt:
549         decrypt(args.decrypt[0], args.decrypt[1])
```

Listing 3.54: Main

Utilizzando l'opzione **"-d"** verrà chiamata la funzione decrypt 3.4.3, che dovrà essere obbligatoriamente seguita da altri due argomenti di tipo stringa.

Mentre, con **"-e"** si richiamerà encryption 3.4.3, anche questa richiede due stringhe come argomenti aggiuntivi.

3.5 Esempio

Una possibile simulazione d'utilizzo è la seguente: l'utente A vuole inviare un messaggio m all'utente B. L'utente A esegue lo script con l'opzione '-e', aggiungendo il messaggio m e infine la chiave da utilizzare.

```
$ python honey.py -e 'I want to impress those twins on the
   fly' 'this is not a psw'

The original seed is:
767374013218785128188399305981146704564335660383368559072822760
631224344752002059274397507888931681425040420077056864255426468
6875

The secret key is:
this is not a psw

The public seed is:
mgOJcq4L17k=XEdM8btFR6apmq6QCLQxvK5bLdD+wqVkOZ8w2RABvb3wu9vyuu
56Ywd/+tfkbC1rxYtHuKFvHufwfNgtoz8qpk7lVxtL+0nVOe9UQtIdLMTxWudt
A+uNX5P4wqlvxg7uJ0yzFJ/b3usJQu4FrrQBzEy/52CfbSI+wke3nltiHHE=
```

Listing 3.55: Utente A esegue lo script honey.py

Con un protocollo scelto, verrà inviato solamente il seed pubblico all'utente B, che utilizzando la password corretta, potrà generare il messaggio originario.

L'utente B lancerà il comando "-d" seguito dal seed pubblico e dalla chiave.

```
$ python honey.py "-d"
   "mgOJcq4L17k=XEdM8btFR6apmq6QCLQxvK5bLdD+wqVkOZ8w2RABvb3wu9
   vyuu56Ywd/+tfkbC1rxYtHuKFvHufwfNgtoz8qpk7lVxtL+0nVOe9UQtIdL
```

```
   MTxWudtA+uNX5P4wqlvxg7uJ0yzFJ/b3usJQu4FrrQBzEy/52CfbSI+wke3
   nltiHHE="
   "this is not a psw"

The original seed is:
767374013218785128188399305981146704564335660383368559072822760
631224344752002059274397507888931681425040420077056864255426464
6875

I want to impress those twins on the fly
```

Listing 3.56: Descrittazione con key CORRETTA

Utilizzando la chiave corretta, il messaggio sarà logicamente il medesimo *"I want to impress those twins on the fly"*.

Ma, se la chiave dovesse essere anche solo leggermente diversa, l'output sarà completamente diverso. Dunque, se un attacker dovesse recuperare il seed, dovrebbe conoscere la password reale, altrimenti, tramite un brute force attack, otterrebbe risultati diversi, ma plausibili.

```
$ python honey.py "-d"
   "mgOJcq4L17k=XEdM8btFR6apmq6QCLQxvK5bLdD+wqVkOZ8w2RABvb3wu9
   vyuu56Ywd/+tfkbC1rxYtHuKFvHufwfNgtoz8qpk7lVxtL+0nVOe9UQtIdL
   MTxWudtA+uNX5P4wqlvxg7uJ0yzFJ/b3usJQu4FrrQBzEy/52CfbSI+wke3
   nltiHHE="
   "i don't know the right password"

The original seed is:
092597059734149435949309044527570278900607463706325555526547525
590881277565945719810085344571534371592165867935337885096635204
4170

Jonah blows each envelope
```

Listing 3.57: Descrittazione con key ERRATA 1

Anche solo cambiando di poche la password, si ottiene una frase completamente diversa, come si può evincere dal testo.

```
$ python honey.py "-d"
...
"123password456"

The original seed is:
512734904394754255784638647942309129402804801835739485890683910
340274798440569290710542759297005715288482453076002461419486 3
```

```
7241

That exhibition awakes the mountain
```

Listing 3.58: Descrittazione con key ERRATA 2

Non sempre le frasi possiedono un senso compiuto, anzi spesso, ma da un punto di vista grammaticale sono corrette e quasi sempre, si riesce a dare un qualche tipo di significato alla frase in output.

```
$ python honey.py "-d"
...
"ggg3tfu37_KantW&7t5t337-63@m$"

The original seed is:
46523885386595776339144270180066114165271558580927420572166354
87212908130332586930215311915759154532449728577291215079317100
4475

Every hat presets a dysfunction
```

Listing 3.59: Descrittazione con key ERRATA 3

```
$ python honey.py "-d"
...
"qwerty"

The original seed is:
65988019452367491502411855667769891513762132823959418090449061
62963315942148593767705735433473640461567301991740336865268257
5438

They agree to want other chops at the spruce
```

Listing 3.60: Descrittazione con key ERRATA 4

3.6 Repository

Il codice completo è disponibile liberamente sul noto sito GitHub, sul mio profilo personale YanDieg al seguente link:

https://github.com/YanDieg/HoneyEncryptionNLPMessages

L'utilizzo è completamente libero, pure future implementazioni o evoluzione, siete solamente inviatati a citare la fonte.

3.7 Limitazioni

Il primo grosso limite, che emerge immediatamente successivamente alla visione degli esempi, è che la correttezza semantica delle frase ottenuta in seguito a decodificata, non è garantita: spesso le frasi non hanno un senso compiuto, ma è anche vero che la stessa frase originaria potrebbe rientrare tra di queste. Ovviamente, avendo un ampissimo numero di possibili chiavi, si otterrà un altrettanto grande insieme di messaggi e sicuramente tra di queste ci sarà un altrettanto ampio range di testi sensati (tra cui il messaggio originario).
Una soluzione definitiva potrebbe essere quella di utilizzare dei sottodizionari relazionati (da condividere assieme al messaggio), usando per esempio *DataMuse* A.7, contenenti solo parole con un determinata affinità con i termini originali.
Il secondo, meno evidente, ma altrettanto fondamentale problema, è la condivisione della struttura grammaticale.
Salvare in un database e utilizzare l'indicizzazione del dizionario, come meccanismo per identificare la struttura grammaticale, non è la scelta più consona; infatti bisognerebbe sviluppare un algoritmo in grado di generare strutture corrette grammaticalmente, partendo dalla frase originaria o da una frase random. Anche se, eseguendo un lavoro di training con un ampio corpus di frasi, è possibile ampliare questo dizionario con migliaia di strutture.

3.8 Future estensioni

Le possibili estensioni sono molteplici, le tre principali che dovrebbero essere le prime ad essere implementate sono:

- Utilizzare dei sottodizionari relazionati per ottenere frasi più o meno corrette dal punto di vista semantico e di contesto, in riferimento alla frase originaria. Dopo aver parsato il testo da inviare-criptare, viene estrapolato ogni sostantivo, avverbio, aggettivo e verbo e, attraverso le api di DataMuse [22], vengono estrapolati dai nostri dizionari un insieme di parole con un alto indice di affinità con le parole (all'interno di un range randomico).

- Per evitare di utilizzare dizionari salvati in locale, dunque accessibili dall'esterno del programma per analizzare le parole e le strutture, si potrebbe utilizzare un database online, accessibile solo attraverso request interne dal programma.
- Rendere la struttura grammaticale esprimibile aritmeticamente, per evitare di usare un dizionario delle strutture. Per esempio sfruttando il modello n-gram ma applicato alle strutture, oppure generando una frase completamente random, forzando il sistema a generare una struttura grammaticale da questa e successivamente ricostruire l'analisi del messaggio corretto.

Capitolo 4

Conclusione

L'**Honey Encryption** potenzialmente potrebbe rilevarsi uno dei metodi crittografici del futuro più sicuri e versatili, purtroppo attualmente, come mostrato dalle varie applicazioni, oltre alla mia, si trova ancora a un punto di evoluzione primordiale, manca probabilmente un adeguato supporto da parte della community e delle imprese per incrementare lo sviluppo.
Utilizzando dati strutturati, grazie a regole definibili chiaramente, funziona teoricamente molto bene, sebbene non abbia ancora ricevuto abbastanza attenzione, come si vede dalla mancanza di programmi pubblici che la sfruttino, per esempio applicandola alla condivisione di pdf editabili con testi prefissati o cifre, come possono essere i certificati medici o la dichiarazione dei redditi.

Applicata al linguaggio naturale, è ancora un punto interrogativo, probabilmente la tecnologia in supporto non permette di sviluppare applicativi pienamente funzionali, se non in parte come nel caso della proposta del *MIT* per le domande o del mio progetto per messaggi di singolo periodo.

Nel futuro, se adeguatamente applicata, rappresenterà una valida alternativa per proteggere i nostri dati non solo da attacchi brute force, ma anche da attacchi di tipo *dictionary* o *rubber-hose*.

Appendice A

An appendix

A.1 Formula di Luhn

La formula di Luhn, conosciuta anche come Modulo 10, è un algoritmo, utilizzato come checksum per i numeri delle carte di credito, che consente di generare e verificare la validità di vari numeri identificativi.

Nel numero di carta di credito, l'ultima cifra corrisponde alla cifra di controllo di Luhn, che viene calcolata sommando tutte le cifre in posizione pari al doppio della somma di quelle in posizione dispari e, del numero ottenuto, si considera il modulo rispetto a 10; Se il risultato della divisione è:

- 0 (dunque si trattava di un multiplo di 10), la cifra di controllo sarà 0
- altrimenti la cifra di controllo sarà pari alla differenza tra 10 e il risultato. Esempio: 77 % 10 = 7 –> Cifra di Luhn = (10 - 7) = 3

Il processo di verifica, di un numero identificativo contenente la cifra di Luhn, è il seguente:

1. Partendo da destra e spostandosi verso sinistra, moltiplicare per 2 ogni cifra posta in posizione pari
2. Laddove la moltiplicazione ha dato un risultato a due cifre, sommare le due cifre per ottenerne una sola (es. 13 = 1+3 = 4)
3. Sommare tutte le cifre, sia quelle che si trovano in posizione pari, sia quelle che si trovano in posizione dispar

Se la somma complessiva sarà divisibile per 10, dunque senza resto, la carta sarà valida.

A.2 Natural Language Toolkit - NLTK

Natural Language Toolkit [19] è una suite di librerie e strumenti per l'analisi simbolica e statistica per l'elaborazione del linguaggio naturale, abbreviato nlp in inglese, sviluppata principalmente in inglese, anche se ormai si è ampliata e ricopre molti idiomi, per il linguaggio di programmazione noto come Python. Grazie alla sua natura free e opensource, è diventata una colonna portante all'interno del suo settore, soprattutto nel mondo didattico e di ricerca. Permette di interfacciarsi con oltre 100 corpora e risorse lessicali, tra le quali c'è WordNet, il principale database online semantico-lessicale inglese.

E' impossibile elencare e descrivere tutti i servizi offerti, la sua funzione principale è quella di supportare la classificazione, tokenizzazione, derivazione, tagging, parsing e funzionalità di ragionamento semantico.

A.3 Wordnet

Wordnet [23] è un database semantico-lessicale per la lingua inglese che si propone di organizzare, definire e descrivere i concetti espressi dai vocaboli. Usato principalmente per scopi didattici e di ricerca, è stato derivato in molteplici lingue, tra le quali l'italiano, ed è strutturato a synset, cioè aggruppamenti di termini con significato affine, con rigide relazioni definite tra di loro (numerate in base ai possibili collegamenti). Ogni termine viene dunque descritto, definito, esemplificato e inserito all'interno di un apposito synset creando una regolata e complessa organizzazione. Si può considerare WordNet come una combinazione tra un dizionario e un theraurus.

Attualmente contiene più di 150 mila parole, in più di 175 mila synset.

A.4 Statistical Coding Scheme - SCS

Con Statical Coding Scheme, o in italiano schema di codifica statistica, si intende un algoritmo statistico simbolico usato per la compressione di dati.
Alcuni esempi di SCS sono la codifica di Huffman, la codifica aritmetica o Prediction by Partial Matching.
Vengono principalmente utilizzati, nel linguaggio naturale, in combinazione con tecniche di previsione statistica.

A.5 N-Gram

In generale per N-GRAM si intende una sottosequenza di n elementi (parole, caratteri, sillabe, frasi) di un testo.

Un modello n-gram è un tipo di modello linguistico probabilistico che si basa sulla previsione dell'elemento successivo in tale sequenza di n-grammi, seguendo il modello di Markov (che afferma che il futuro stato dipende solamente da quello corrente e non da quelli passati). Viene ampiamente utilizzato nell'ambito del natural language processing, per la generazione di frasi, partendo da un corpus training che permette di costruire lo schema statistico-probabilistico.

A.6 Standford Parser

Un parser del linguaggio naturale, è banalmente un programma che opera su frasi e testi, estraendo la struttura grammaticale delle frasi. Per permettere ciò, devono essere sottoposti a training, cioè gli va insegnato come eseguire l'analisi, attraverso l'inserimento di regole statistiche, ma soprattutto con un parsing manuale di frasi d'esempio, accompagnato ovviamente da un'intelligenza artificiale correttamente sviluppata.
Lo Stanford Parser [34] viene all'unanimità considerato il punto di riferimento per l'analisi delle frasi in lingua inglese (sopre anche altre lingue, come Cinese, Arabo e ovviamente Italiano). Si tratta del principale statistical natural language parser, sviluppato per JAVA (infatti per funzionare in Python necessità della creazione di un virtual Java Server).
Inserendo una frase di input, in output comparirà un albero, rappresentate la struttura grammaticale della frase.

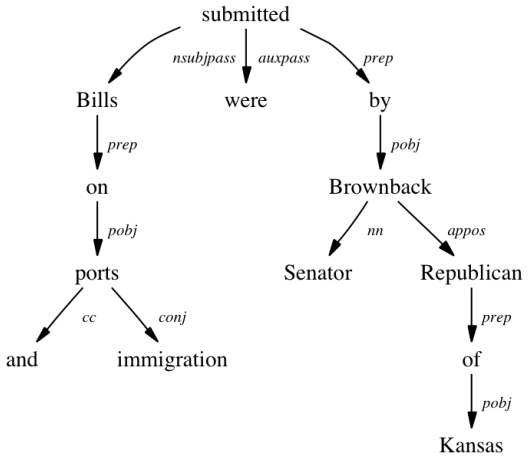

Figura A.1: Basic dependencies - Tree Structure (Standford Parser Output)
©https://nlp.stanford.edu/software/stanford-dependencies/brownback-uncollapsed.png/

A.7 Datamuse

Datamuse [22]è una suite di servizi accessibili tramite api, librerie e sito web, in supporto al natural language processing, analisi dei testi e dei dizionari.
Lo si può considerare in parte simile a WordNet (il servizio OneLook), ma i più, inserendo una parola, mostra tutti i termini (verbi, avverbi, aggettivi e sostantivi) correlati, con un rispetto valore, con questa, in ordine di "affinità" decrescente (RhymeZone).
Inoltre permette di trovare rime, omofonie e molto altro.

A.8 Parsing e Tokenization delle frasi

In generale, quando viene data una stringa che esprime una frase, in pasto a un parser grammaticale come può essere spaCy [21] o lo Standford Parser A.6, il risultato in output consiste ovviamente in una sua analisi, espressa generalmente sotto forma di un albero con le foglie indicanti i termini della frase e i rami la dipendenza fra di essi. Questo albero, in spaCy, viene espresso come una lista di "token", uno per ogni parola. Ovviamente ogni token possiede dei determinati attributi:

Tabella A.1: TOKEN

Tokenizzazione di "My sister loves these beautiful dogs"

TEXT	lemma	index	POS	POS TAG	DEP	Parent_Index
My	my	0	DET	PRP$	poss	1
sister	sister	1	NOUN	NN	nsubj	2
loves	love	2	VERB	VBZ	ROOT	2
these	these	3	DET	DT	det	5
beautiful	beautiful	4	JJ	PRP$	amod	5
dogs	dog	5	NOUN	NNS	dobj	2

Ovviamente, i possibili attributi sono molti di più, per il mio progetto ci siamo limitati a questi mostrati in tabella A.1. Sul sito del software spaCy è presente la lista completa [35].

Sempre sul medesimo sito web, è presente la completa documentazione riguardante i POS, POS TAG (Part-of-speech tagging) [36] e DEP (Dipendenze) [37].

A.9 Electronic code book

ECB [38] è il modo più semplice per criptare con AES, è comunque applicabile in altri sistemi a chiave simmetrica che operano su blocchi di dati di lunghezza fissa

"B". Se si ha un dato con lunghezza maggiore di B, deve essere suddiviso in blocchi di lunghezza B con eventuale padding.

A.10 Password-Based Key Derivation Function 2

Password-Based Key Derivation Function 2 [39], comunemente denominato PBKDF2, è una funzione di derivazione di chiavi crittografice (key derivation function) utilizzata per ridurre la vulnerabili da attacchi di tipo brute force, che data una password qualsiasi inserita dall'utente, ne deriva una chiave della lunghezza necessaria (256 bits se usata con AES-256) composta da caratteri pseudorandom.
Bisognerebbe applicare anche una stringa di salazione (disturbo) per evitare che data la medesima password si ottengano due key uguali.

Bibliografia

[1] Ari Juels and Thomas Ristenpart. *Honey Encryption: Security Beyond the Brute-Force Bound*. 2014.

[2] Debasis Das, U.A. Lanjewar and S.J. Sharm. *The Art of Cryptology: From Ancient Number System to Strange Number System*. 2013.

[3] C. Shannon. *Communication Theory of Secrecy Systems*. 1949.

[4] Joan Boyar, IMADA, University of Southern Denmark. *RSA and Primality Testing*. 1949.

[5] Wombat security. *User Risk Report*. https://info.wombatsecurity.com/hubfs/WombatProofpoint-UserRiskSurveyReport2018_uk.pdf?t=1541521118115. 2018.

[6] Stan Schroeder. «You can now browse through 427 million stolen MySpace passwords». In: *Mashable* (2016). URL: https://mashable.com/2016/07/01/myspace-password-database/?europe=true.

[7] Wikipedia contributors. *Yahoo! data breaches — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-September-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Yahoo!_data_breaches&oldid=906505056.

[8] Samuel Gibbs. «Dropbox hack leads to leaking of 68m user passwords on the internetce passwords». In: *The Guardian* (2016). URL: https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach.

[9] Troy Hunt. «Observations and thoughts on the LinkedIn data breach». In: *troyhunt.com* (2016). URL: {https://www.troyhunt.com/observations-and-thoughts-on-the-linkedin-data-breach/}.

[10] Joseph Bonneau, Computer Laboratory, University of Cambridge. *The science of guessing: analyzing an anonymized corpus of 70 million passwords*. 2012.

[11] The Imperva Application Defense Center (ADC). *Consumer Password Worst Practices*. https://www.imperva.com/docs/gated/WP_Consumer_Password_Worst_Practices.pdf. 2014.

[12] Troy Hunt. *Have I Been Pwned?* https://haveibeenpwned.com.

[13] @GhostProjectME. *Ghost Project*. https://ghostproject.fr/.

[14] Canetti R., Dwork C., Naor M., Ostrovsky R. *Deniable Encryption*. 1997.

[15] Nirvan Tyagi, Daniel Zuo, Jessica Wang, Kevin Wen. *Honey Encryption Applications. Implementation of an encryption scheme resilient to brute-force attacks*. 2015.

[16] Rahul Chatterjee, Anish Athalye, Devdatta Akhawe, Ari Juels, Thomas Ristenpart. *pASSWORD tYPOS and How to Correct Them Securely*. 2016.

[17] Hoyul Choi and Jongmin Jeong and Woo, Simon S. and Kyungtae Kang and Junbeom Hur. «Password typographical error resilience in honey encryption». In: *Computers and Security* (2018). DOI: 10.1016/j.cose.2018.07.020.

[18] Nirvan Tyagi, Daniel Zuo, Jessica Wang, Kevin Wen. *Honey Encryption Applications -EXAMPLES*. https://github.com/danielzuot/honeyencryption.

[19] Steven Bird, Edward Loper, Ewan Klein. *Natural Language Toolkit*. https://www.nltk.org/.

[20] Jazzband. *Inflect*. https://github.com/jazzband/inflect.

[21] Matthew Honnibal. *spaCy*. https://www.spacy.io/.

[22] onelook.com. *Datamuse*. https://www.datamuse.com/.

[23] Princeton University. *WordNet*. https://wordnet.princeton.edu/.

[24] Dr. Santhi Baskaran, S.V.L Sarat Chandra , P.Venkatesh , E.Silambarasan , M.Dinesh. *Implementation of Enhanced Honey Encryption for IoT Security*. 2017.

[25] Edwin Mok, Azman Samsudin, Soo-Fun Tan3. *Implementing the Honey Encryption for Securing Public Cloud Data Storage*. 2017.

[26] Yoon, Ji Won and Kim, Hyoungshick and Jo, Hyun-Ju and Lee, Hyelim and Lee, Kwangsu. *Visual Honey Encryption: Application to Steganography*. 2015.

[27] Rasmita Sahu and Shajid Ansari. «A Secure Framework for Messaging on Android Devices with Honey Encryption». In: *International Journal of Engineering and Computer Science* (2017). URL: http://www.ijecs.in/index.php/ijecs/article/view/2777.

[28] Kim, Joo Im and Yoon, Ji Won. «Honey chatting: A novel instant messaging system robust to eavesdropping over communication». In: *Institute of Electrical and Electronics Engineers Inc.* (2016).

[29] Abiodun Esther Omolara, Aman Jantan, Oludare Isaac Abiodun and Howard Eldon Poston. *A Novel Approach for the Adaptation of Honey Encryption to Support Natural Language Message*. 2018.

[30] Wikipedia contributors. *Advanced Encryption Standard — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-September-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=909852571.

[31] spaCy. *spaCy - Linguistic Features*. https://spacy.io/usage/linguistic-features.

[32] CliPS. *Pattern.en*. https://www.clips.uantwerpen.be/pages/pattern-en.

[33] Darsey Litzenberger. *PyCrypto - The Python Cryptography Toolkit*. https://www.dlitz.net/software/pycrypto/.

[34] Natural Language Processing Group at Stanford University. *The Stanford Parser: A statistical parser*. https://nlp.stanford.edu/software/lex-parser.shtml.

[35] spaCy. *spaCy - Attributes*. https://spacy.io/api/token#attributes.

[36] spaCy. *spaCy - POS TAG*. https://spacy.io/api/annotation#pos-tagging.

[37] spaCy. *spaCy - Syntactic dependency labels*. https://spacy.io/api/annotation#dependency-parsing.

[38] Wikipedia. *Electronic code book — Wikipedia, L'enciclopedia libera*. [Online; in data 5-settembre-2019]. 2017. URL: http://it.wikipedia.org/w/index.php?title=Electronic_code_book&oldid=88897026.

[39] Wikipedia contributors. *PBKDF2 — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-September-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=PBKDF2&oldid=910941103.