



SAPIENZA
UNIVERSITÀ DI ROMA

Tecniche di occultamento dati all'interno di codice macchina

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica
Corso di laurea in Informatica

Michele Donvito
Matricola 1710775

A handwritten signature in black ink, reading 'Michele Donvito'.

Relatore
Massimo Bernaschi

Correlatore

A handwritten signature in black ink, reading 'Massimo Bernaschi'.

A.A. 2019-2020

Sommario

Indice delle figure e delle tabelle	iii
1. Introduzione	4
1.1 Cenni storici	4
1.2 Steganografia moderna	5
1.3 Struttura della tesi	6
2. Steganografia su codice macchina	7
2.1 Definizione	7
3. Teoria di fondo	9
3.1 Crittografia	9
3.2 Formati Eseguibili	12
3.3 Assembly, Assembler e Disassembler	18
3.4 Introduzione al software hidder	20
4. Architettura Software Hidder	22
5. Analisi e conclusioni	32
5.1 Data rate	32
5.2 Lavori futuri	34
5.3 Steganalisi	36
5.4 Conclusioni	40
6. Bibliografia e Sitografia	42

Indice delle figure e delle tabelle

Figura 1 AES.....	10
Figura 2 Funzionamento EBC	11
Figura 3 Funzionamento CBC	11
Figura 4 Formato file ELF.....	12
Figura 5 Formato file PE	14
Figura 6 Registri generali versione 64 bit.....	18
Figura 7 Esempio di offuscamento.....	20
Figura 8 Flowchart del software <i>hidder</i>	22
Figura 9 Rateo di offuscamento negli ELF 64 bit.....	32
Figura 10 Rateo di offuscamento negli ELF 32 bit	33
Figura 11 Rateo di offuscamento nei PE32 e PE 32+	33
Figura 12 Distribuzione delle istruzioni negli ELF 64 bit	36
Figura 13 Distribuzione delle istruzioni negli ELF 32 bit	37
Figura 14 Distribuzione delle istruzioni nei PE32 e PE32+	37
Figura 15 Distribuzione dopo l'utilizzo di <i>hidder</i> negli ELF 32,64 bit	38
Figura 16 Distribuzione dopo l'utilizzo di <i>hidder</i> nei PE32,PE32+.....	38
Figura 17 Avvio hidder tramite bash.....	40
Figura 18 Dimostrazione del corretto funzionamento di <i>ida_copy</i>	41
Figura 19 Avvio decoder e avvio del file <i>hello_copy</i>	41

1.Introduzione

1.1 Cenni storici

La steganografia è una tecnica di occultamento di un messaggio all'interno di un altro dato, non necessariamente digitale, innocuo [15], come una foto, un disegno, una lista o un semplice testo. Chiameremo contenitore colui che trasporta il messaggio nascosto.

Le sue origini non sono note, alcune tracce risalgono addirittura all'antica Grecia, nelle storie di Erodoto¹:

In una queste si narra di Isteo, il quale rasò la testa di un suo servo trascrivendo sul suo scalpo un messaggio. Una volta ricresciuti i capelli gli affidò l'incarico di recarsi munito di una pergamena fittizia da un suo vassallo, Aristagora, riferendo di farsi rasare per consegnare il vero messaggio.

Un'altra storia narra le vicende di Demarato di Sparta, che scoprendo di una possibile ed imminente invasione persiana, scrisse su di una tavoletta tale avvertimento, ricoprendola successivamente di cera. Una volta asciutta scrisse un secondo messaggio innocuo.

Spostandoci più avanti nella storia circa nel 1500, troviamo i trattati di Tritemio:

Steganographia e *Clavis Steganographiae*. All'interno vi sono vari esempi di applicazioni di steganografia, tra le più semplici proposte vi è l'occultamento di un messaggio nelle iniziali di tutte le parole pari. Per esempio, nel testo; "Caro amico, ieri tentando di tenere un esperimento sulla natura del tasso di origine...", si nasconde il messaggio ATTENTO.

Risulta evidente, che tutte le pratiche precedentemente esposte si basano sul nascondere un messaggio in modo da non risultare visibile.

Differentemente dalla crittografia, che punta a rendere il messaggio incomprensibile, la steganografia consente di nascondere il messaggio senza destare sospetti. Purtroppo, questa tecnica porta con sé una falla; i messaggi nascosti possono essere facilmente scoperti se viene effettuata un'analisi attenta del messaggio vettore. A tal proposito rientra in gioco il trattato sopracitato, il quale ha gettato le basi di un modello di steganografia moderna attualmente ancora in uso, applicato anche in questo progetto. Tale modello, suggerisce l'aggiunta di una criptazione al messaggio nascosto.

Riprendendo l'esempio del messaggio nascosto nelle iniziali pari, l'abate Tritemio propone di applicare un cifrario di Cesare² prima di effettuare il camuffamento, rendendo "impossibile" la decifrazione seppur un malintenzionato riesca a scovare il messaggio nascosto.

Questo implica che ambedue gli interlocutori posseggano il cifrario utilizzato.

¹ *Le Storie di Erodoto di Alincarnasso* sono la prima opera completa storiografica nella Letteratura Occidentale, circa 430 a.C.

² Il *cifrario di cesare* o cifrario a scorrimento, è un algoritmo crittografico a sostituzione monoalfabetica in cui ogni lettera del messaggio corrisponde ad un'altra che si trova un numero fisso di posizioni dopo nell'alfabeto.

1.2 Steganografia moderna

La steganografia moderna ha avuto inizio con l'avvento dei Personal Computer e la crescente diffusione dei dati digitali.

Principalmente le tecniche moderne si distinguono in:

- Steganografia ad attaccante passivo o ad attaccante attivo.
- Steganografia generativa o iniettiva.

Nel primo criterio, studiamo il ruolo che svolge un eventuale attaccante.

Con passivo si ipotizza che l'attaccante cerchi semplicemente di determinare la presenza di un messaggio nascosto, analizzando costantemente le comunicazioni. Una prima definizione è formalizzata con il *problema dei prigionieri*³.

Con attaccante attivo invece si ipotizza che esso non solo analizzi costantemente le comunicazioni, ma le alteri cercando di interferire con un possibile messaggio nascosto precedentemente sfuggito all'analisi.

Nel secondo criterio si esamina la costruzione del contenitore.

Con generativo si specifica che il messaggio vettore è per l'appunto generato a partire dal messaggio da nascondere, e quindi costruito su misura.

Nella iniettiva il messaggio vettore è noto a posteriori e viene alterato in modo impercettibile per trasportare il messaggio da nascondere adattandosi ad esso.

Esistono molti algoritmi di offuscamento all'interno di dati digitali come immagini, video, audio ed altri.

La tipologia più diffusa è LSB (dall'inglese *least significant bit*). Questa tecnica è applicabile alle immagini digitali, le quali sono formate da un insieme di pixel.

Ogni pixel è espresso tipicamente in formato RGB ovvero una terna di interi a cui è attribuito un valore da 0 a 255, i quali si riferiscono ai colori base: rosso, verde e blu.

Il bit meno significativo⁴ di ogni colore di ogni pixel influisce sull'immagine in modo quasi impercettibile ad occhio umano. Pertanto, la modifica di quest'ultimo permette di nascondere 3 bit in ogni pixel, senza alterare in maniera visibile l'immagine.

Alcune varianti della tecnica LSB sono applicabili anche ai file video ed audio.

Purtroppo, queste metodologie di occultamento delle informazioni sono spesso identificate come invasive e facili da rilevare poiché seguono *pattern* ben noti.

Pertanto nella steganografia moderna è introdotta una chiave segreta (il cifrario utilizzato da

³ Il *problema dei prigionieri* formulato da Simmons enuncia in breve: avendo due prigionieri divisi in celle diverse, questi ultimi cercano di comunicare tra loro avendo un carceriere che li controlla costantemente, devono pertanto trovare un modo di comunicarsi informazioni segrete all'insaputa del carceriere, il quale se insospettito dell'invio di messaggi sospetti spedisce entrambi i prigionieri in isolamento.

⁴ In informatica i valori vengono spesso trattati in una base diversa da quella decimale, il sistema numerico binario ed esadecimale sono le varianti più gettonate. Con *bit meno significativo* si intende la cifra binaria in posizione zero, la quale può assumere soltanto il valore 0 o 1.

Tritemio) utilizzata in combinazione con qualche tecnica di crittazione con l'unico fine di rendere arduo per un attaccante reperire informazioni segrete.

1.3 Struttura della tesi

Per migliorare la chiarezza espositiva dell'elaborato si propone una divisione in capitoli. Introduremo inizialmente la steganografia su codice macchina dando una precisa definizione.

Di seguito verranno illustrate le teorie di fondo definendo tutti gli aspetti fondamentali necessari per una completa comprensione dell'elaborato.

Verrà presentato poi il software "*Hidden*" scritto e progettato personalmente per sostenere l'idea (la tesi stessa) di fondo. *Hidden* nasconde messaggi in file eseguibili PE32, PE32+, ELF sia 32 che 64 bit, destinati all'architettura x86.

Infine, verranno svolte delle analisi sulle tecniche di offuscamento progettate. Verranno esposte in ulteriore analisi due possibili tecniche applicabili a file eseguibili.

Un'introduzione alla steganalisi concluderà il capitolo.

Completeremo la tesi con delle considerazioni personali, alcuni possibili impieghi futuri ed un esempio di utilizzo.

2.Capitolo 1: Steganografia su codice macchina

2.1 Definizione

In genere quando si parla di messaggi nascosti all'interno di file, si pensa subito alle foto, video ed audio.

In questo elaborato si andrà ad analizzare metodi di steganografia iniettiva su file eseguibili, più precisamente all'interno del codice macchina, il quale verrà modificato secondo regole ben precise:

1. Il funzionamento del file eseguibile contenitore deve rimanere invariato.
2. La dimensione del contenitore non deve essere alterata.

Differentemente dai file digitali sopra citati, nascondere dati all'interno di codice sembra essere un aspetto alquanto trascurato dalla ricerca. Ci sono comunque diverse tecniche interessanti e brevettate, orientate più verso il codice sorgente, ovvero prima della compilazione.

Principalmente vengono nascosti messaggi nel codice per fini di *watermark*⁵, *fingerprint*⁶ o per firmare digitalmente un eseguibile, ognuno di questi utilizzi prefissa delle caratteristiche ben precise sull'algoritmo da utilizzare, anche se spesso questi coincidono. Per esempio, un *watermark* ai fini di *copyright* non deve passare inosservato, mentre una steganografia di un messaggio segreto deve passare inosservata.

Principalmente possiamo dividere le tecniche di offuscamento in due tipi: statico e dinamico. L'offuscamento statico è applicato nel codice stesso, ed è accessibile dal file su disco. Un esempio di *watermark* statico è l'aggiunta di una stringa costante inutilizzata contenente le direttive di copyright. Es.

```
const char c[] = "Copyright license of ....";
```

L'offuscamento dinamico invece è applicato nel codice, ma è accessibile solamente durante l'esecuzione e in presenza di determinati input. Il primo esempio che ci viene in mente sono gli *Easter Eggs*⁷, dove un determinato input fornisce schermate segrete, ricordiamo il simulatore di volo in Excel97, nel quale seguendo una combinazione ben precisa di azioni si accede ad un rudimentale simulatore di volo, nel quale sono presenti i nomi di coloro che hanno contribuito alla realizzazione del prodotto stesso.

⁵ *Watermark* è l'inclusione all'interno di file digitali di informazioni, le quali possono in seguito essere estratte per determinare la provenienza del file. Prende per questo il nome di watermark ovvero filigrana in italiano, utilizzata nello stesso modo nelle banconote per determinarne la provenienza e l'origine.

⁶ *Fingerprint* è un messaggio spesso di lunghezza fissa che identifica un determinato file, il produttore, il destinatario o un codice di brevetto.

⁷ *Easter Eggs* in informatica è un contenuto innocuo inserito all'interno di un prodotto dagli sviluppatori stessi.

Iniziamo a dare alcune definizioni importanti per procedere: *Il Codice macchina*, è il livello di astrazione⁸ più basso, ovvero quello utilizzato direttamente dalla CPU, si tratta di una serie di 0 e 1 che lo rendono quasi illeggibili per noi esseri umani.

Chi effettivamente interpreta ed esegue il codice macchina è quindi il processore (CPU).

Ogni modello di CPU appartiene ad una determinata famiglia di processori, le quali differiscono per caratteristiche architetturali. Alcuni esempi noti di architetture sono ARM, x86, MIPS e così via. All'interno dell'architettura è definito l'ISA (dall'inglese *instruction set architecture*) ovvero l'insieme delle istruzioni base che il processore può svolgere, ciò comporta la variazione del codice macchina tra le varie famiglie di processori⁹.

Il progetto si baserà sull'architettura x86 nelle estensioni i386 (e nelle sue generazioni successive i486 ... i686 e così via) e AMD64.

Per facilitare i programmatori sono stati creati dei linguaggi che aiutano la lettura del codice macchina, detti, linguaggi *assembly*. Ne esistono varie sintassi, tra le più note ci sono MIPS, INTEL, AT&T. Un'istruzione in codice macchina per comodità può essere scritta in notazione esadecimale, es. 48 83 c0 32, l'assembly non è altro che la trasformazione di questo esadecimale in un equivalente mnemonico testuale, in questo caso: ADD RAX, 50, al quale viene attribuito il significato di; sommare 50 a RAX. Specifichiamo che l'operando a sinistra è detto di destinazione e l'operando a destra, sorgente, per cui l'equivalente ad alto livello della precedente istruzione è RAX=RAX+50. Nel progetto si utilizzerà la sintassi INTEL.

A questo punto possiamo dare una definizione di Steganografia su codice macchina:

- La steganografia su codice macchina è una tecnica **iniettiva** di occultamento dati, che utilizza un file eseguibile come contenitore, con la quale modificando le istruzioni in codice macchina senza alterare il comportamento dell'eseguibile stesso, si possono nascondere dati.

⁸ Per i linguaggi di programmazione l'astrazione ne indica il distacco dalle istruzioni in linguaggio macchina. Un codice scritto in Python è molto più vicino al linguaggio dell'uomo che al linguaggio macchina pertanto si dice linguaggio ad alto livello.

⁹ I processori moderni implementano spesso emulatori di ISA di altre famiglie, come per esempio il processore i5-9600k che pur essendo un AMD64 può eseguire istruzioni di tutti gli ISA della famiglia x86. Questa soluzione è generalmente più lenta rispetto ad un ISA implementato in hardware.

3.CAPITOLO 2: Teoria di fondo

Prima di entrare nel dettaglio dell'architettura e realizzazione del software *hiddler*, iniziamo con l'introduzione di tutti i fondamenti teorici di cui abbiamo bisogno.

3.1 Crittografia

La crittografia è lo studio di come mantenere la sicurezza di un messaggio e la sua riservatezza codificandolo in una forma illeggibile (testo cifrato, in inglese *ciphertext*).

La crittografia può essere facilmente espressa dalla seguente formula matematica:

$$ENC(M) = C \text{ (Encryption Process)}$$

$$DEC(C) = M \text{ (Decryption Process)}$$

Esiste una grande varietà di algoritmi di crittografia più o meno efficienti, ogni tecnica si adatta ad un determinato contesto. Andremo a vedere due aspetti fondamentali: la crittazione e la decriptazione, utilizzando la stessa **chiave**.

L'utilizzo di una chiave condivisa sia per la crittazione che decriptazione è comunemente indicato con il nome: a *chiave simmetrica* (a chiave condivisa).

Gli algoritmi simmetrici si dividono in due diverse tipologie: cifratura a flusso (*Stream Cipher*) e cifratura a blocchi (*Block Cipher*).

- 1 Nei *cifrari a flusso*, vengono presi bit o byte di dati e codificati singolarmente per poi essere utilizzati nella codifica del successivo, per questo chiamato anche cifratura a stati.
- 2 Nei *cifrari a blocchi*, il processo di codifica è orientato ad un blocco di dati, tipicamente dai 64 ai 512 bit, ognuno trattato indipendentemente.

Nel paragrafo "G" di [4] Hanna Willa Dhany, viene illustrato il metodo: *Password Based Encryption (PBE)*, il quale si adatta egregiamente al progetto.

PBE è un metodo crittografico simmetrico, il quale utilizza un testo (*password*) passato come input e trasformato in una chiave. Questa chiave viene poi utilizzata per criptare il messaggio in chiaro (*plaintext*) in un testo cifrato (*ciphertext*). La stessa tecnica è utilizzata per decriptare, passando da un testo cifrato ad un testo in chiaro.

Vediamo come è stato adattato PBE ad *hiddler*: come algoritmo di trasformazione da un testo di lunghezza variabile ad una chiave di lunghezza fissa, è stato optato per SHA256. Lo SHA256 è una funzione di *hash* crittografico appartenente al gruppo SHA-2, il suo fine è di trasformare un testo di grandezza variabile in una chiave univoca¹⁰ di dimensione fissa,

¹⁰ Trasformando testi di lunghezza variabile in fissa, è matematicamente impossibile l'univocità. Ciò che si intende per univocità è che la probabilità che testi simili diano lo stesso digest è quasi impossibile.

detto digest, di lunghezza 256 bit, il quale non può essere decriptato e quindi ritornare il suo contenuto testuale.

Come algoritmo di crittografia è stato ritenuto opportuno l'utilizzo dell'AES (*Advanced Encryption Standard*), il formato più diffuso ad oggi e standard del governo americano. In figura 1, possiamo vedere il funzionamento di AES con una chiave di 128bit, utilizzandone una di 256 il numero di xor nel ciclo diventa 14.

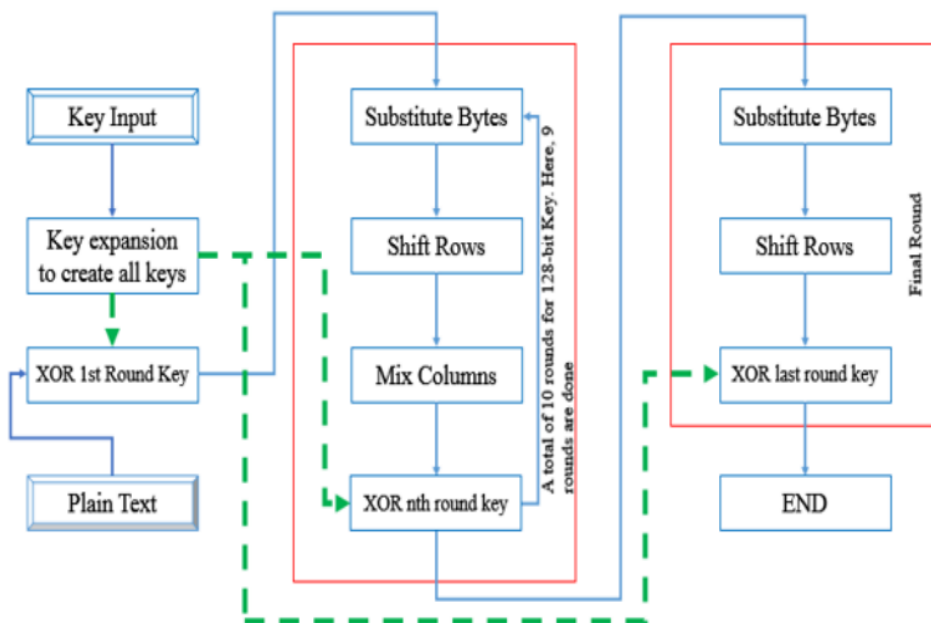


FIGURA 1 AES TRATTA DA [1] NISHA P. SHETTY, NIKHIL RANJAN (2016)

In aggiunta si utilizza una *modalità di funzionamento dei cifrari a blocchi*¹¹ diversa da quanto suggerito in [4]. La modalità ECB suggerita permette di criptare ogni blocco di *plaintext* indipendentemente l'uno dall'altro, Figura 2. Questa modalità, dal punto di vista del nostro progetto, genera più svantaggi che vantaggi.

I vantaggi sono: codice parallelizzabile ed errori localizzati in singoli blocchi, entrambi ininfluenti. Al contrario è favorevole ad attacchi di forza bruta data l'alta frequenza di blocchi uguali in *plaintext* di una certa lunghezza, questi permettono di determinare la chiave utilizzata e pertanto il messaggio di partenza [5] zachgrace.com/posts/attacking-ecb/.

¹¹ In crittografia, la modalità di funzionamento dei cifrari a blocchi è una serie di azioni aggiuntive per un algoritmo a cifratura a blocchi per aumentarne la sicurezza.

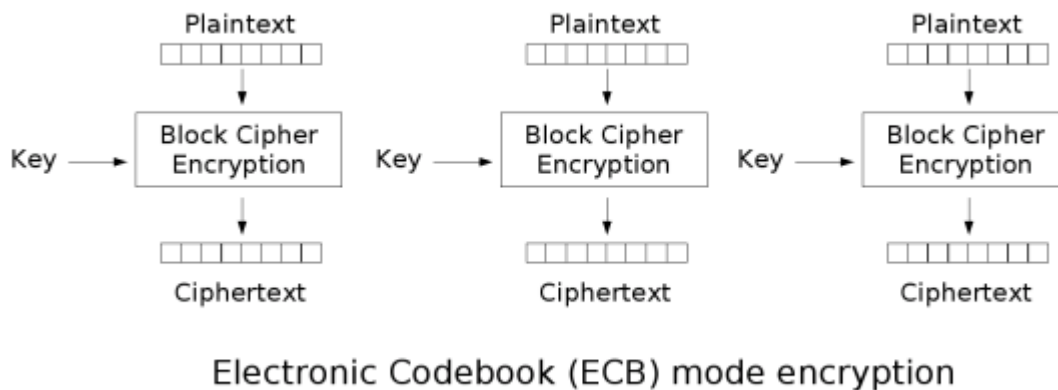


FIGURA 2 FUNZIONAMENTO EBC

Come alternativa viene utilizzata la modalità CBC (*Cypher block chaining*), quest'ultima fondamentalemente inverte i vantaggi con gli svantaggi della modalità ECB sopra elencati.

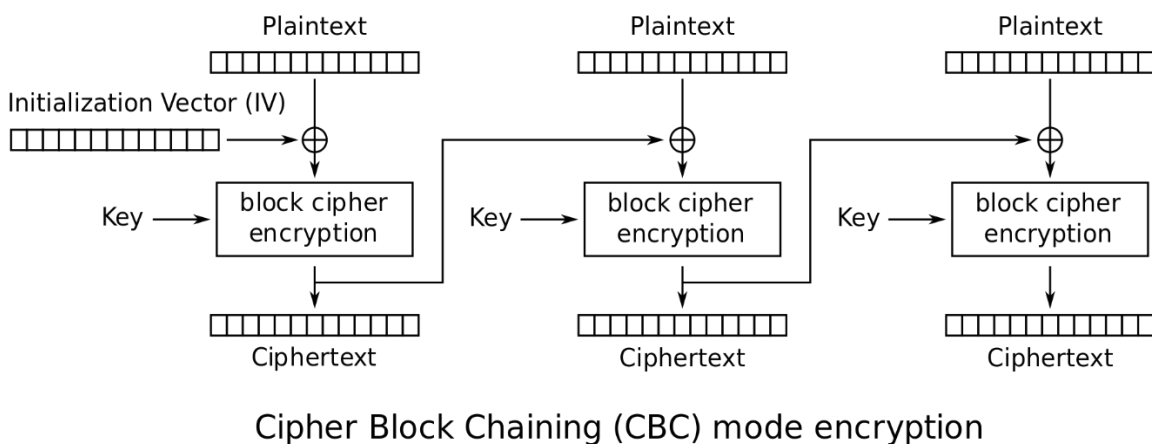


FIGURA 3 FUNZIONAMENTO CBC

Lo stesso blocco di testo in chiaro, se ripetuto, produce blocchi di testo cifrato differenti, questo avviene perché, prima di applicare l'algoritmo di criptazione, il blocco di *plaintext* viene messo in XOR¹² con il *ciphertext* del blocco precedente.

In aggiunta per il primo blocco non avendone uno precedentemente cifrato viene utilizzato un vettore di inizializzazione con cui effettuare lo XOR.

¹² Lo XOR o disgiunzione esclusiva, è un operatore logico che restituisce VERO se entrambi gli ingressi hanno lo stesso valore, altrimenti restituisce FALSO.

3.2 Formati file eseguibili

Un file eseguibile è un file contenente linguaggio macchina, direttamente eseguibile dal computer ed accompagnato da tutte le informazioni per la sua corretta esecuzione. Esistono diverse tipologie di file eseguibili, i quali differenziano nel formato. Prevalentemente si distinguono tre formati: PE, ELF, MACH-O appartenenti ai rispettivi sistemi operativi *Windows*, *Unix*, *Mac OS*. Andremo a vedere nel dettaglio i formati PE ed ELF sia nelle versioni a 32 che 64 bit. Per facilitare la descrizione dei loro campi, verranno utilizzate le *struct*¹³ create all'interno del progetto *hider* per analizzare i file eseguibili.

3.2.1 ELF

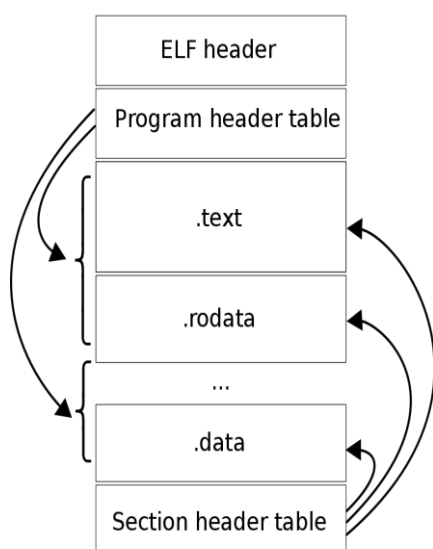


FIGURA 4 FORMATO FILE ELF

Il formato ELF (*Extensible Linking Format*), standard per i file oggetto, librerie condivise e altre tipologie di file, è strutturato come in figura 4.

Troviamo in primo luogo un'intestazione contenente tutte le informazioni riguardo il file: la versione, l'architettura di destinazione, la dimensione del *program header table*, il numero di sezioni del *section header table* e altri campi.

La sua dimensione varia tra le architetture a 32 e 64 bit ma non nel formato, ovvero l'ordine e il fine dei vari campi rimane invariato, semplicemente gli indirizzi sono espressi con 8 bytes invece di 4.

La *program header table* contiene le informazioni utili per la creazione dell'immagine di processo quando questo viene caricato in memoria ed eseguito.

Di seguito troviamo una serie di blocchi di dimensione variabile in cui sono contenuti i dati di ogni sezione: **.text** contiene il codice macchina, **.data** contiene le variabili globali o statiche, **.shstrtab** contiene i nomi di tutte le sezioni sotto forma di array di stringhe. Ci sono varie altre sezioni, non sempre tutte utilizzate.

Infine, vi è la *section header table*, che contiene i *metadati*¹⁴ riguardanti ogni sezione del file, è composta da una serie di righe dove ognuna comprende tutti i metadati di una specifica sezione, come l'inizio del nome all'interno di *.shstrtab*, l'offset di inizio dei dati della sezione e la loro dimensione.

¹³ Le *struct* o meglio strutture in c, sono un insieme di variabili di diverso tipo. Si distinguono dagli *array* che possono contenere solo variabili dello stesso tipo. Alcuni esempi di tipo sono: int, char, double e così via.

¹⁴ I *metadati* sono informazioni che descrivono un insieme di dati. Un esempio è costituito dall'indice di un libro.

Di seguito, sono riportate le strutture dati, dove ogni campo è ordinato secondo la sequenza di dichiarazione all'interno dell'*header*. La descrizione di ogni campo è situata sotto forma di commento adiacente allo stesso.

Iniziamo dall' *ELF header* che contiene i seguenti campi:

```
typedef struct{
    uint8_t e_ident[16]; /* Nei primi 4 bytes, vi è una firma, la quale attesta che si tratti di un file
    ELF ovvero "\x7f","E","L","F", il byte successivo definisce l'architettura di destinazione, vale 1 per
    indicare 32 bit oppure 2 per indicare 64 bit */
    uint16_t e_type; /* Tipo di file: oggetto, eseguibile, libreria dinamica ... */
    uint16_t e_machine; /* Architettura di destinazione: i386, amd64, arm, mips ... */
    uint32_t e_version; /* Indica la versione, 1 per la versione originale */
    uint64_t e_entry; /* Indirizzo di memoria dal quale inizierà l'esecuzione */
    uint64_t e_phoff; /* Offset del program header table */
    uint64_t e_shoff; /* Offset del section header table */
    uint32_t e_flags; /* Flags per specifiche extra, il significato dipende dall'architettura usata */
    uint16_t e_ehsize; /* Dimensione ELF header in bytes */
    uint16_t e_phentsize; /* Dimensione di ogni riga nella program header table */
    uint16_t e_phnum; /* Numero di righe nella program header table */
    uint16_t e_shentsize; /* Dimensione di ogni riga nella section header table */
    uint16_t e_shnum; /* Numero di righe nella section header table */
    uint16_t e_shstrndx; /* Indice della riga nella section header table equivalente al section header
    string table (.shstrtab) */
} Elf64_Ehdr;
```

Passiamo ad analizzare una riga della *section header table*.

```
typedef struct{
    uint32_t sh_name; /* Offset della sezione .shstrndx che punta al section name della sezione
    corrente */
    uint32_t sh_type; /* Tipo di sezione: program data, stringhe, note e altri */
    uint64_t sh_flags; /* Indica gli attributi di una sezione */
    uint64_t sh_addr; /* Indirizzo di memoria virtuale dove verrà locata la sezione corrente una volta
    caricata in memoria */
    uint64_t sh_offset; /* Offset della sezione corrente su disco */
    uint64_t sh_size; /* Dimensione della sezione corrente su disco */
    uint32_t sh_link; /* Indice di un'altra sezione associata alla sezione corrente */
    uint32_t sh_info; /* Alcune informazioni aggiuntive */
    uint64_t sh_addralign; /* Valore di allineamento della sezione */
    uint64_t sh_entsize; /* Dimensione di ogni riga, se la sezione contiene una tabella */
} Elf64_Shdr;
```

Grazie a queste due strutture dati è possibile scrivere un *parser* molto primitivo di file ELF in modo tale da ottenere tutto il codice macchina situato nelle varie sezioni contenenti programma eseguibile.

3.2.2 PE

Per i file PE non è stato possibile individuare una libreria Unix maneggevole e ben strutturata come per i file ELF, che effettuasse un'analisi parziale del file al fine di ottenere il codice macchina della sezione `.text`, pertanto ne ho scritta una basandomi sulla documentazione ufficiale di windows: [5] docs.microsoft.com/en-us/windows/win32/debug/pe-format.

DOS MZ Header
DOS Stub
PE Header
Section Table
Section 1
Section 2
Section ...
Section n

FIGURA 5 FORMATO FILE PE

Il formato PE è strutturato come in figura 5.

DOS MZ Header e DOS Stub si trovano in testa al file e contengono le informazioni di compatibilità per MS-DOS. In DOS MZ Header all'*offset* 0x3c si trova un campo contenente l'*offset* di partenza del PE Header, che rappresenta l'unica informazione utile da queste 2 prime intestazioni.

PE Header è formato da diversi insiemi di campi a seconda del tipo di file. Negli *images file*, ovvero i file eseguibili, ci sono i seguenti elementi:

COFF Header, Optional Header.

Iniziamo ad introdurre le *struct* realizzate in *hidderr* per analizzare i file PE:

COFF Header, è un *header standard* per tutti i tipi di file, sia oggetto che immagine. Ha dimensione e formato fisso ed è il seguente:

```
typedef struct coff_file_header{
    uint8_t Signature[SIGNATURE_SIZE]; /* Contiene la dicitura "PE\x0\x0" ed indica per l'appunto che
    si tratta di un file PE */
    uint16_t Machine; // ID dell'architettura target: i386, amd64...
    uint16_t NumberOfSections; /* indica il numero di sezioni, ci permette di calcolare la dimensione
    della Section Table */
    uint32_t TimeDateStamp; // la data di creazione del file
    uint32_t PointerToSymbolTable; // offset che punta alla COFF symbol table
    uint32_t NumberOfSymbols; // numero di entry nella symbol table
    uint16_t SizeOfOptionalHeader; /* dimensione degli optional header, la section table segue
    sempre gli optional header, pertanto possiamo calcolarci l'offset della section table */
    uint16_t Characteristics; // Flag informativi del file.
}coff_file_header;
```

Optiona Header, è un header utilizzato soltanto nei file immagine. La sua dimensione non è fissa, ma indicata nel *COFF header*. Può essere diviso in 3 distinti *header*:

- 1 Campi Standard (*standard field*)
- 2 *Windows-specific fields*
- 3 *Data directories*

1. Negli *Standard fields*, l'unica differenza tra PE32 e PE32+ sta nel campo *BaseOfData*, il quale non è presente nella versione a 64 bit

```
typedef struct optional_header_32_standard_field {
    uint8_t Magic[MAGIC_SIZE]; /* questo valore determina la versione, contiene "\x0B\x01" se 32 bit
o "\x0B\x02" se 64 bit */
    uint8_t MajorLinkerVersion;
    uint8_t MinorLinkerVersion;
    uint32_t SizeOfCode; /* Questo campo indica la somma della dimensione di tutte le sezioni
contenenti codice */
    uint32_t SizeOfInitializedData;
    uint32_t SizeOfUninitializedData;
    uint32_t AddressOfEntryPoint; /* indirizzo di partenza del programma, relativo
all'image base address, quando il file è caricato in memoria */
    uint32_t BaseOfCode; /* indirizzo di memoria relativo all'inizio della sezione codice una volta
caricata in memoria */
    uint32_t BaseOfData; /* indirizzo relativo all'inizio della sezione dati una volta
caricata in memoria */
}oh32_standard_field;
```

2. *Windows specific field*, contiene informazioni utili al *linker*¹⁵ e al *loader*¹⁶ di windows. Le versioni PE32 e PE32+ di questa intestazione divergono solamente nella dimensione, Alcuni campi sono da 64bit invece di 32, lo scopo e l'ordine delle variabili non cambia.

```
typedef struct optional_header_32_windows_specific {
    uint32_t ImageBase; // image base address del file quando caricato in memoria
    uint32_t SectionAlignment; /* allineamento delle sezioni, in bytes (di solito la grandezza
    dell'allineamento di pagina dell'architettura) */
    uint32_t FileAlignment; // allineamento per i raw data del file.
    // i prossimi 7 valori indicano numeri di versione
    uint16_t MajorOperatingSystemVersion;
    uint16_t MinorOperatingSystemVersion;
    uint16_t MajorImageVersion;
    uint16_t MinorImageVersion;
    uint16_t MajorSubsystemVersion;
    uint16_t MinorSubsystemVersion;
    uint32_t Win32VersionValue;
    uint32_t SizeOfImage; // dimensione in byte dell'immagine (inclusi gli header) in memoria
    uint32_t SizeOfHeaders; // dimensione in byte degli header in memoria
    uint32_t CheckSum; // checksum dell'image file
    // altre variabili non utili per il progetto
    uint16_t Subsystem;
    uint16_t DllCharacteristics;
    uint32_t SizeOfStackReserve;
    uint32_t SizeOfStackCommit;
    uint32_t SizeOfHeapReserve;
    uint32_t SizeOfHeapCommit;
    uint32_t LoaderFlags;

    uint32_t NumberOfRvaAndSizes; /* numero di righe in data directory */
}oh32_windows_specific;
```

¹⁵ Il *Linker* è il software che collega il codice oggetto con i file aggiuntivi come header e crea il file eseguibile.

¹⁶ Il *Loader* prende il file eseguibile fornito dal *linker* e lo carica in memoria principale, allocando lo spazio utile per librerie, programma e altro.

3. *Data directory*, sono righe di 8 bytes che indicano un indirizzo VA¹⁷ e la dimensione di una tabella o una stringa che windows usa. La *data directory* è caricata in memoria per uso a *runtime*. La dimensione non è alterata dalla versione.

```
typedef struct data_directory {
    uint32_t VirtualAddress;
    uint32_t Size;
}data_directory;
```

Gli header di sezione seguono subito dopo gli *optional header* e contengono i *metadati* di ogni sezione.

```
typedef struct image_section_header {
    uint8_t Name[8]; /* contiene il nome identificativo, se di lunghezza inferiore ad 8 caratteri, in
    contrario segue un metodo alternativo per reperire il nome. */
    uint32_t VirtualSize; // dimensione della sezione quando caricata in memoria
    uint32_t VirtualAddress; // indirizzo del primo byte quando caricata in memoria
    uint32_t SizeOfRawData; // dimensione della sezione su disco
    uint32_t PointerToRawData; // offset della sezione su disco
    uint32_t PointerToRelocations;
    uint32_t PointerToLinenumbers;
    uint16_t NumberOfRelocations;
    uint16_t NumberOfLinenumbers;
    uint32_t Characteristics;
}image_section_header;
```

Per il contenuto di ogni sezione (*section data*) è utilizzata la seguente *struct* sia per file di formato ELF 32, 64 bit che PE32, PE32+.

```
typedef struct hdr_section_content
{
    unsigned char* CODE; // contiene i dati della sezione
    int section_size; // dimensione di CODE
    int index; // indice della sezione nella section header table
}hdr_section_c;
```

¹⁷ Gli indirizzi VA sono indirizzi identici agli RVA a cui non è stato sottratto il base address dell'immagine. Ogni processo ha il proprio spazio per gli indirizzi VA, i quali sono separati al livello fisico. Questo rende non prevedibile un VA a differenza degli RVA. RVA è l'indirizzo di un elemento dopo che è stato caricato in memoria, sottratto dell'imageBase address. È quasi sempre diverso dall'indirizzo fisico dell'elemento in memoria file su disco.

3.3 Assembly, assembler e disassembler

Nei paragrafi precedenti sono state già affrontate le tematiche di codice macchina e assembly, dandone una definizione generale. Vediamo ora specifici aspetti utili al progetto: registri, *assembler*, *disassembler*.

In *assembly* i dati possono essere salvati in memoria, ma l'accesso è dispendioso, pertanto sono inseriti registri interni al processore ad accesso molto più rapido [7] www.tutorialspoint.com/assembly_programming/assembly_registers.htm.

Nell'architettura INTEL, esistono 3 categorie di registri: Generali, di controllo e di segmento. I registri generali sono 9 nei processori a 32 bit e 17 nei processori a 64 bit. Ogni registro può essere ridotto per potervi lavorare su una parte di esso come in figura 6, nella versione a 64 bit ci sono altri 8 registri addizionali numerati da R8 a R15.

Main registers (8/16/32/64 bits)				
RAX	EAX	AX	AL	A register
RBX	EBX	BX	BL	B register
RCX	ECX	CX	CL	C register
RDX	EDX	DX	DL	D register
Index registers (16/32/64 bits)				
RSI	ESI	SI		Source Index
RDI	EDI	DI		Destination Index
Stack registers (16/32/64 bits)				
RBP	EBP	BP		Base Pointer
RSP	ESP	SP		Stack Pointer
Instruction pointer (16/32/64 bits)				
RIP	EIP	IP		Instruction Pointer

FIGURA 6 REGISTRI GENERALI VERSIONE 64 BIT

Nella versione a 32 bit sono utilizzati solamente i registri in figura e con dimensione massima 32 bit (EAX).

I registri di controllo sono utilizzati per salvare lo stato del calcolatore in forma di *flag*, da cui il nome EFLAG o registro di stato. Per il progetto vengono analizzati solamente alcuni di questi flag: **Overflow** indica che c'è stato un cambio nel bit più significativo o sinistro, **Carry** contiene il riporto dopo un'operazione aritmetica, **Zero** indica se il risultato di un'operazione è zero, **Parity** indica se il numero di 1-bit nell'ultimo risultato è pari o dispari, **Sign** contiene il segno del risultato dell'ultima operazione.

La trasformazione da *assembly* a codice macchina viene effettuata da un compilatore¹⁸ chiamato *Assembler*. All'interno del progetto è stata utilizzata una libreria, *keystone*, [9] github.com/keystone-engine/keystone che simula il comportamento di un *assembler*, pertanto dato un'istruzione o un set di istruzioni *assembly* ritorna l'insieme di codice macchina equivalente.

¹⁸ Un compilatore è un programma che traduce istruzioni da un linguaggio ad un altro, tipicamente di livello inferiore.

L'inverso di un *assembler* è un *disassembler*, ovvero un decompilatore¹⁹, questo è tra gli strumenti fondamentali per la realizzazione di *hidder*. A tal proposito si utilizza la libreria *capstone* [8] github.com/aquynh/capstone, un valido strumento che fornisce l'*assembly* equivalente al codice macchina in sintassi INTEL inserito in un vettore di *struct cs_insn*²⁰. All'interno di *cs_insn* ci sono altre strutture dati ritenute fondamentali per lo sviluppo del progetto, come la struttura *details* che offre informazioni sui registri e i flags modificati dall'istruzione corrente, dell'indirizzo dell'istruzione ed altre informazioni.

¹⁹ Un decompilatore è un software che dato un codice assembly, riesce ad ottenere un codice funzionalmente equivalente scritto in linguaggio ad alto livello.

²⁰ *cs_insn* è la struttura dati utilizzata da *capstone* per contenere un'istruzione assembly ed i suoi dettagli.

3.4 Introduzione al software

Hidden ha un utilizzo piuttosto semplice: dato un file eseguibile (*input*), un *path*²¹ ad un file esistente o meno (*output*) e un file contenente il messaggio da nascondere (*plaintext*). Verrà inserito in output un file eseguibile funzionalmente identico ad input ma contenente all'interno del codice macchina il messaggio offuscato. Per decodificare bisogna passare come argomento un file eseguibile contenente informazioni nascoste ed un file dove inserire il messaggio che verrà recuperato.

Per nascondere messaggi ci baseremo principalmente sulle inefficienze dei compilatori, che possono essere semplici ridondanze nel codice, o utilizzo di istruzioni poco efficienti.

Per ridondanze nel codice intendiamo, istruzioni diverse ma funzionalmente equivalenti, le quali al più differiscono in piccoli dettagli, come il settaggio del registro EFLAG²².

Prendiamo come esempio l'istruzione di somma `add r/m, imm`; dove viene aggiunto ad un `r/m`²³ il valore contenuto nell'immediato, questa operazione può essere scritta come sottrazione e viceversa, semplicemente invertendo di segno l'immediato; `sub r/m, -imm`.

Usando questa alternativa dell'`add` e `sub`, possiamo nascondere un bit per ogni istruzione, attribuendo ad `add` il bit 0 e a `sub` il bit 1. Vediamo un esempio di camuffamento di 2 bit.

Offuscamento 00	Offuscamento 10
<code>add eax, 20</code>	<code>sub eax, -20</code>
<code>cmp eax, 50</code>	<code>cmp eax, 50</code>
<code>ja LABEL_1</code>	<code>ja LABEL_1</code>
<code>add eax, 10</code>	<code>add eax, 10</code>
Offuscamento 01	Offuscamento 11
<code>add eax, 20</code>	<code>sub eax, -20</code>
<code>cmp eax, 50</code>	<code>cmp eax, 50</code>
<code>ja LABEL_1</code>	<code>ja LABEL_1</code>
<code>sub eax, -10</code>	<code>sub eax, -10</code>

FIGURA 7 ESEMPIO DI OFFUSCAMENTO

²¹ Con *path* in italiano percorso, si intende la posizione specifica di un file all'interno di un file system.

²² In assembly il registro EFLAG è un registro che contiene lo stato corrente del processore, dove ogni bit all'interno del registro equivale ad un flag, i quali sono determinati dall'esecuzione delle istruzioni, che in base al loro risultato impostano i vari flag. Es. un'istruzione di somma imposta il flag OF se avviene un overflow.

²³ `r/m` è un operando di un'istruzione in assembly, indica che può utilizzare un registro o ad una locazione in memoria come operando.

Per definire meglio cosa si intende con “utilizzo di istruzioni poco efficienti”, introduciamo un'altra tecnica basata sempre sulle istruzioni di somma e sottrazione, nella quale camuffiamo bits negli immediati di operazioni aritmetiche consecutive.

Dato il seguente codice:

```
add rax, 40
mov rcx, 1
push rcx
sub esp, 4
add rax, 10
cmp rcx, rax
```

Notiamo che tra l'operazione `add rax, 40` e `sub rax, 10` il registro `rax` non viene mai utilizzato, pertanto basterebbe utilizzare un'unica istruzione: `add rax, 50`.

A ridosso di ciò possiamo utilizzare le due istruzioni per nascondere un elevato numero di bit, in questo caso essendo un registro da 64 bit possiamo nasconderne 63, esclusivamente aggiustando i valori degli immediati, ottenendo al termine delle due operazioni sempre lo stesso valore nel registro `rax` ma nascondendo dei bit. Vediamo come:

```
add rax, MEX
mov rcx, 1
push rcx
sub esp, 4
add rax, 50-MEX
cmp rcx, rax
```

Al termine delle operazioni a `rax` viene effettivamente aggiunto il valore di 50, mentre nell'immediato della prima operazione nascondiamo un messaggio. Si offuscano soltanto 63 bit poiché altrimenti nella seconda operazione applicata a `rax` si potrebbe incappare in un *overflow*, pertanto teniamo il bit di segno per evitare ciò.

Vediamo un altro esempio lavorando sul registro `al` ad 8 bit (range da 127 a -128) invece di 64 per semplicità.

Nel secondo immediato sommato ad `al`, nascondiamo “il valore che effettivamente modifica `al`” sottratto del messaggio inserito nella prima operazione. Se avessimo il valore -50 e dovessimo nascondere il messaggio 96, avremmo da inserire $-50 - 96 = -146$, non realizzabile con 8 bit, verrebbe considerato dal processore come un `add al, 110`. Pertanto, invece di 96 utilizzeremo -96 cambiando l'inutilizzato bit di segno, questo porterà ad avere nella seconda operazione $-50 + 96 = 46$, ovvero un valore realizzabile con soli 8 bit.

Grazie a questa scelta di non utilizzo del bit di segno, è possibile applicare anche lo scambio di istruzioni visto nell'esempio precedente, guadagnando due ulteriori bit. In realtà potremmo recuperare un altro bit scegliendo dove posizionare l'immediato di dimensione maggiore, ma non è stato implementato al momento.

Per analizzare la fattibilità e le capacità della steganografia su codice macchina con l'utilizzo delle precedenti tecniche, abbiamo utilizzato eseguibili di diversi sistemi operativi tra cui (Ubuntu 16.04+, Manjaro e Windows 10), pertanto andremo a contatto sia con gli eseguibili nei formati PE che ELF nelle versioni 32 e 64 bit.

4. CAPITOLO 3: Architettura Software Hidder

Avendo dunque un'idea del comportamento di *hidder* e i fondamenti teorici, passeremo a discutere nel dettaglio i vari passaggi implementativi per realizzare l'offuscamento.

Di seguito il flowchart del software:

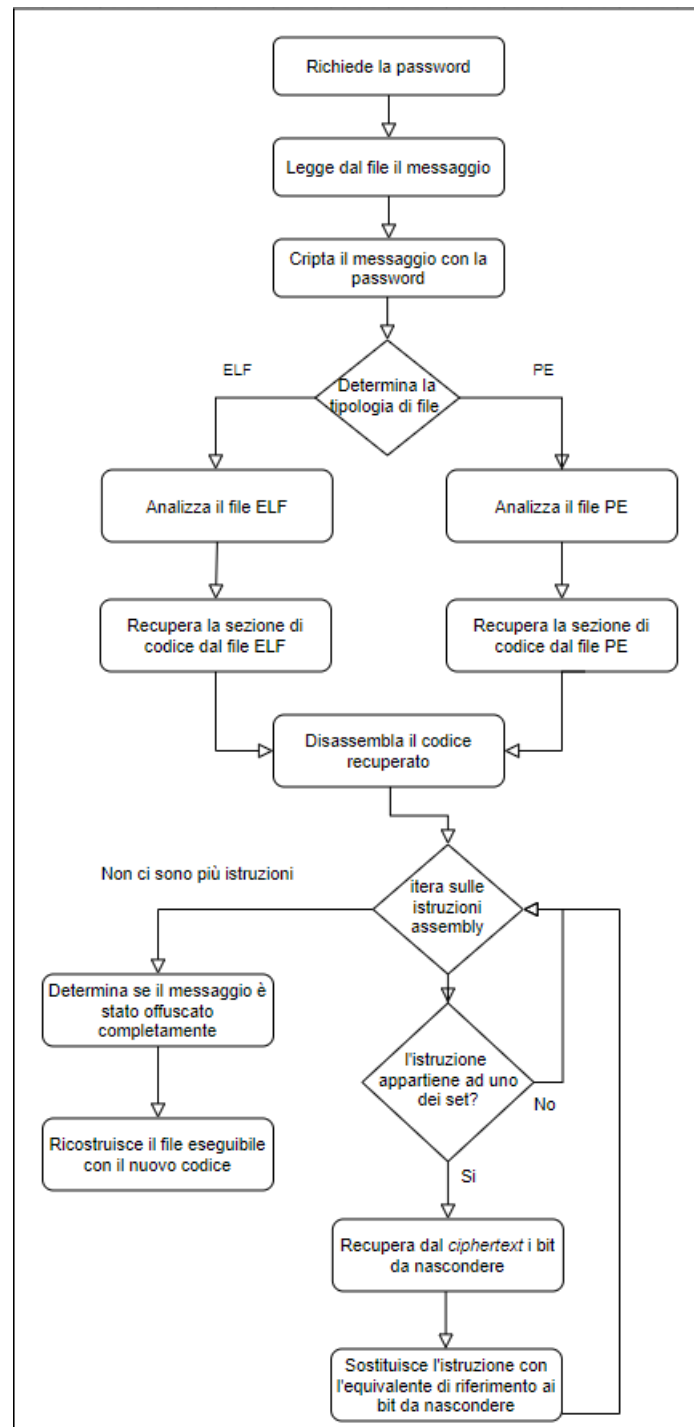


FIGURA 8 FLOWCHART DEL SOFTWARE *HIDDER*

Di seguito è riportata la struttura utilizzata per contenere i dati riguardanti il messaggio da nascondere:

```
typedef struct hdr_data_message
{
    unsigned char* password;
    unsigned char digest[ (DIGEST_LEN/BYTE_LEN) ];
    unsigned char ivec[ IV_LEN ];
    char* plaintext; // testo in chiaro e lunghezza
    int plaintext_len;
    unsigned char* cyphertext; // testo cifrato e lunghezza
    int cyphertext_len;

    unsigned char crypted_plaintext_len[INT_BYTES_LEN];
    // variabili utili per salvare lo stato dei bit nascosti
    unsigned char bit_encoded;
    int byte_encoded;
} hdr_data;
```

Descriviamo in dettaglio la realizzazione di ogni blocco del precedente Flow chart:

1. Per richiedere la password si utilizza la funzione `getpass()` di `<unistd.h>`.
`hdr_data->password = getpass("Inserisci una password: ");`
2. Per leggere il messaggio da nascondere si apre il file passato in `argv[3]`, si recupera la sua dimensione tramite `fstat` per poi leggerne il contenuto.
`read (fd, hdr_data->plaintext, hdr_data->plaintext_len);`
3. Viene eseguita la cifratura del plaintext, utilizzando l'algoritmo di *hash* SHA-256 sulla *password*, implementato nella libreria *OpenSSL*. Impostando il *digest* come *seed*²⁴ per generare numeri casuali viene generato il vettore di inizializzazione IV. Il *digest* e il vettore IV vengono messi in XOR con la lunghezza del *plaintext*,

```
for( i=0; i<(DIGEST_LEN/BYTE_LEN); i+=4 ){
    temp ^=(((unsigned int) (digest[i+0])) << (3*BYTE_LEN) ) |
           (((unsigned int) (digest[i+1])) << (2*BYTE_LEN) ) |
           (((unsigned int) (digest[i+2])) << ( BYTE_LEN ) ) |
           (((unsigned int) (digest[i+3])) ) );
}
hdr_data->crypted_plaintext_len = temp ^ hdr_data->plaintext_len;
```

questa operazione è effettuata dal momento in cui nascondere in chiaro la lunghezza ne potrebbe compromettere la sicurezza, data l'elevata frequenza di 0 nei bit più

²⁴ Il Seed è un valore utilizzato come base per generare numeri pseudo-casuali, fortunatamente lo stesso seed viene generato sempre stessa serie di numeri casuali nella funzione `rand()` linux.die.net/man/3/srand.

significativi. In conclusione, viene utilizzato l'AES sul *plaintext*. I valori ottenuti durante il procedimento vengono memorizzati all'interno di una *struct hdr_data*.

4. Per determinare la tipologia di file, si parte dall'analisi per gli ELF. Vengono letti i primi 5 bytes per determinare con i primi 4 se corrisponde all'intestazione dei file ELF, con il quinto si individua l'architettura di destinazione se a 32 o 64 bit. Se il precedente controllo da esito negativo proviamo l'analisi per i file PE, reperendo il valore salvato in posizione 0x3c che sappiamo essere l'offset di partenza del COFF header. A questo punto viene letto il COFF header ed il campo *magic* del PE header. Comparando il campo *Signature* del COFF header con la stringa identificativa "PE\x00\x00" si determina il formato PE, la versione a 32 o 64 bit è determinata comparando il campo *magic* del PE header con i rispettivi identificativi.
5. L'analisi del file ELF ha avvio con l'inizializzazione della *struct* corrispondente alla versione architetturale. Verrà poi letto l'ELF header del file:

```
read(fd, (void *)elf_header, sizeof(Elf32_Ehdr) 25);
```

Nel ELF Header troviamo la posizione e la dimensione di ogni riga, e il numero di righe del *Section Header Table* pertanto leggiamo l'intera tabella:

```
lseek(fd, (off_t)eh.e_shoff, SEEK_SET);
for(i=0; i<eh.e_shnum; i++) {
    read(fd, (void *)&sh_tbl[i], eh.e_shentsize);
}
```

In egual modo per i file PE viene inizializzata la struct equivalente alla versione a 32 o 64 bit. Vengono letti COFF Header, *Standard field* e *Windows Specific Field*,

```
read(fd, (void *)ms_dos, sizeof(ms_dos_stub));
lseek(fd, (off_t)ms_dos.offset, SEEK_SET);
read(fd, (void *)coff_header, sizeof(COFF_file_header));
read(fd, (void *)standard_field, sizeof(standard_field));
read(fd, (void *)win_field, sizeof(windows_specific_field));
```

viene determinata la lunghezza del Data Directory moltiplicando la lunghezza di un entry (8 bytes)*win_fiels.NumberOfRvaAndSizes. Finalmente abbiamo l'offset di partenza del *Section header table* per cui procediamo a leggere l'intera tabella.

```
lseek(fd, (off_t)win_field.NumOfRva*8, SEEK_CUR);
```

²⁵ La funzione `sizeof(type)` ritorna la dimensione necessaria per archiviare il tipo passato come argomento, pertanto su un puntatore ritornato dalla `malloc` non restituisce il valore passato alla `malloc`, ma ritorna la dimensione del puntatore ovvero 4 sui sistemi 32-bit e 8 sui sistemi 64-bit.

Determiniamo la dimensione del *Section header table* moltiplicando la dimensione di una entry per il numero di sezioni `coff_header.NumberOfSections`.

```
sh_dim = sizeof(section_header_table)*NumberOfSections
sh_tbl = malloc( sh_dim );
read(fd, (void *)sh_tbl, sh_dim );
```

6. Il recupero dei dati della sezione di codice è analogo in qualunque versione, ovvero viene cercata la sezione `.text`.

Per gli ELF andrebbero cercate le sezioni con *type* “SHT_PROGBITS” e *flags* “SHF_EXECINSTR & SHF_ALLOC”, poiché il codice non è salvato unicamente in una sezione. Purtroppo dopo varie analisi si è riscontrato che più del 99% del camuffamento avviene nella sezione `.text`.

Nei file ELF per trovare `.text`, si legge prima la sezione `.shstrndx` che contiene i nomi di tutte le sezioni. Il suo indice è salvato nel ELF header nel campo `eh.e_shstrndx`

```
shstrndx = malloc( sh_tbl[eh.e_shstrndx].sh_size );
lseek(fd, (off_t)sh_tbl[eh.e_shstrndx].sh_offset, SEEK_SET);
read(fd, (void *) shstrndx, sh_tbl[eh.e_shstrndx].sh_size);
```

Di seguito si scorre ogni *entry* di `sh_tbl` finché non viene rilevata quella con nome `.text`. Per determinare il nome basta sommare al puntatore `shstrndx` l’offset salvato all’interno del campo `sh_name` dell’entry di `sh_tbl`, tutti valori reperiti sono salvati nella struct `hdr_section_code`.

```
for( i=0; i<eh.e_shnum ; i++){
    if( strncmp( (shstrndx + sh_tbl[i].sh_name) ,
                CODE_SECTION_NAME, strlen(CODE_SECTION_NAME)) == 0 ){
        lseek(fd, (off_t)sh_tbl[i].sh_offset, SEEK_SET);
        read(fd, (void *)CODE, sh_tbl[i].sh_size);
        section_size = sh_tbl[i].sh_size;
        index = i;
        return true;
    }
}
```

Per i file PE avviene in modo equivalente ma con una semplificazione, il campo `Name` della entry di `sh_tbl` contiene direttamente il nome e non un offset (se il nome contiene meno di 8 caratteri). Pertanto si procede direttamente con il ciclo *for* sostituendo l’espressione dell’*if* con la seguente:

```
strncmp(sh_tbl[i].Name, CODE_SECTION_NAME, strlen(CODE_SECTION_NAME))
```

7. Viene disassemblato il codice macchina tramite la libreria *capstone*, ottenendo un vettore di *struct cs_insn*, dove ogni elemento corrisponde ad un’istruzione. La struttura di *cs_insn* è formata da diversi valori; specifichiamo di seguito i campi utilizzati e pertanto necessitano di essere definiti:

- **Id**, questo è un identificativo per l'istruzione, ad ogni mnemonico equivale un valore es. `X86_INS_ADD` = 8, `X86_INS_SUB` = 333 ...
- **mnemonic**, è una stringa che contiene l'istruzione mnemonica, es. "add".
- **op_str**, è una stringa che contiene gli operandi in formato mnemonico, es. "rax, 0x500". In coppia con *mnemonic* si ottiene l'assembly equivalente all'istruzione in codice macchina, es. `add rax, 0x500`.
- **bytes**, è un vettore di byte che contiene il codice macchina equivalente all'istruzione analizzata.
- **size**, intero che indica la dimensione del campo *bytes* in quella determinata istruzione.
- **address**, è un *offset* che indica l'inizio dell'istruzione all'interno del codice.
- **detail**, è una *struct* che contiene dettagli aggiuntivi sull'istruzione come:
 - **regs_read**, indica quali registri vengono letti implicitamente per la corretta esecuzione dell'istruzione, es. nell'istruzione `jz rax` viene letto il registro *eflag*.
 - **regs_write**, contiene tutti i registri sui quali l'istruzione andrà a scrivere implicitamente, es. `add rax, rcx` andrà a scrivere su *eflag*.
 - **x86.eflags**, *details* a sua volta contiene un'altra *struct*, ma di questa ci interessa solo il campo *eflags*, il quale contiene l'elenco dei flags coinvolti nell'istruzione e in che modo. es. `X86_EFLAGS_MODIFY_OF` o `X86_EFLAGS_RESET_CF`.

8. A questo punto non resta che scorrere il vettore *insn*, se non ci sono più istruzioni si prosegue con il punto 9. Se invece è disponibile un'ulteriore istruzione si passa al punto 11, il quale contiene la descrizione dell'intero ciclo (i passi 11, 12 e 13).
9. Una comparazione tra il numero di byte nascosti e la lunghezza del *ciphertext* determinerà se il messaggio è stato nascosto interamente.
10. Si costruisce il file eseguibile contenitore copiando il file eseguibile di partenza sostituendo i dati della sezione *.text* con il nuovo codice macchina che contiene il *ciphertext*. Vediamo come è stato realizzato per i file PE:

```
fread(&temp, 1, sh_tbl[hdr_code.index].PointerToRawData, f_input);
fwrite(&temp, 1, sh_tbl[hdr_code.index].PointerToRawData, f_output);
fwrite(hdr_code.CODE, hdr_code.section_size, 1, f_output);
fseek(f_input, hdr_code.section_size, SEEK_CUR);
while(true) {
    ret = fread(&temp, 1, 500, f_input);
    if( (ret==0) && feof(f_input) ) break;
    fwrite(&temp, 1, ret, f_output);
}
```

11. Questo punto è il fulcro del progetto, qui vengono esaminati tutti i *set* di istruzioni equivalenti per determinare l'appartenenza ad uno di essi. Vediamo ora i *set* nel dettaglio ed il loro funzionamento. La variabile `index` contiene l'indice di `insn` dell'istruzione attualmente esaminata:

Un primo set di istruzioni equivalenti è stato visto precedentemente ed è l'alternanza tra le operazioni add e sub con immediati. Vediamo come è stato implementato:
Viene prima controllato se id è una corretta operazione.

```
if(( insn[index].id == X86_INS_ADD ) || ( insn[index].id == X86_INS_SUB ))
```

Se sì, verifico che l'operando sorgente sia un immediato. A questo punto vanno controllati i flag che effettivamente sono trattati in modo diverso tra un add ed una sub ovvero OF e CF non causino problemi al corretto funzionamento del file eseguibile.

Per fare ciò ho realizzato la funzione `check_flags(insn, index, flags)`, nella quale viene scorso il vettore `insn`, se individuata un'istruzione di *return*, *leave*, *call* oppure una POPF ovvero l'istruzione equivalente alla normale pop ma che va a sostituire il registro *eflag*, si può dichiarare ininfluente la modifica dei flag, e pertanto è applicabile la sostituzione.

```
if((insn[index].id == X86_INS_RET) || (insn[index].id == X86_INS_LEAVE) ||  
(insn[index].id==X86_INS_POPF) || (insn[index].id==X86_INS_CALL)) return true;
```

Al contrario se si trova un'istruzione di PUSHF ovvero una *push* del registro *eflag* in *stack*, purtroppo ci si deve fermare e rendere l'operazione inutilizzabile. A questo punto se l'istruzione non corrisponde a nessuna delle seguenti si analizzerà la variabile `x86.eflags`, se l'istruzione modifica uno o più flag tra quelli passati da argomento li rimuoviamo dalla lista dei ricercati, se invece l'istruzione utilizza uno dei flag tra quelli ricercati allora anche in questo caso l'istruzione deve essere invalidata. Prima del passaggio all'istruzione successiva si controlla la lista dei flag ricercati, se vuota l'istruzione è utilizzabile.

Nel caso dei jump, non possiamo continuare il flusso sequenziale ma dobbiamo seguire il flusso alternativo. Purtroppo, non è stata realizzata la simulazione del salto con registro o valore in memoria. Pertanto, in caso di JMP invalidiamo l'operazione a prescindere, come per la PUSHF (questa scelta invalida un numero minimo, quasi trascurabile di istruzioni).

In base al risultato di `check_flags`, si procede realizzando l'operazione inversa. Ricordiamo che in questo momento si ha la validità dell'istruzione per la sostituzione.

Dividiamo `op_str` utilizzando ',' come delimitatore, ottenendo così l'operando di destinazione e l'immediato, inverte l'immediato e costruisco l'istruzione inversa.

```
char * dest = strtok( op_str, OPERAND_DELIMITATOR );  
char * imm = strtok(NULL, OPERAND_DELIMITATOR );  
int r_imm = (int)strtol( imm, NULL, 16 );  
r_imm *= (-1);  
sprintf( reverse_insn, "%s %s, 0x%x", strcmp(mnemonic,ADD)==0?SUB:ADD,  
dest, r_imm);
```

Ottenuta l'istruzione inversa all'interno di `reverse_insn` tramite l'*assembler keystone* si ottiene il codice macchina equivalente:

```
ks_asm(ks, reverse_instr, 0, &ks_output, &ks_size, &ks_count);
```

L'output va controllato, poiché può capitare che l'inverso abbia una dimensione in codice macchina più grande rispetto all'istruzione che stiamo sostituendo, es.

```
0: 48 83 eb 80          sub    rbx, -0x80
4: 48 81 c3 80 00 00 00 add    rbx, 0x80
```

Purtroppo, questo è dovuto ad un'inefficienza della libreria *assembler*, in alcuni casi indefiniti non riconosce le istruzioni di tipo ISTR r/m32, imm8 traducendo l'istruzione come se avesse un immediato a 32 bit. Per porvi rimedio va sostituito l'*opcode* 81 con 83, e vanno eliminati gli ultimi 3 *byte* dell'istruzione.

Non resta che prendere un bit dal *ciphertext* per decidere quale operazione utilizzare. La funzione per fare ciò è *take_bits* che ritorna il valore dei bit richiesti.

Il secondo set di istruzioni equivalenti che andremo a considerare è il set TOASXC, formato dalle seguenti istruzioni: test, or, and, add, sub, xor, cmp. Indichiamo per evitare ripetizioni solamente l'insieme ad 8 bit, in realtà sono controllati anche i registri ax, eax, rax.

```
struct table toasxc8_table[] =
{
    { { '\xA8', '\xff' }, NF, "test al, -1" },
    { { '\x0C', '\x00' }, NF, "or  al,  0" },
    { { '\x24', '\xff' }, NF, "and  al, -1" },
    { { '\x04', '\x00' }, NF, "add  al,  0" },
    { { '\x2C', '\x00' }, NF, "sub  al,  0" },
    { { '\x34', '\x00' }, NF, "xor  al,  0" },
    { { '\x3C', '\x00' }, NF, "cmp  al,  0" },
};
```

Il primo valore indica il codice macchina in byte dell'operazione ed è colui che effettivamente andrà a sostituire il codice macchina per il cambio di istruzione. Essendo istruzioni fisse ho ritenuto inefficiente utilizzare l'*assembler* per ogni riscontro.

Il secondo campo indica i *flags* che vanno controllati, in questo caso NF riferisce che i flag sono manipolati da ogni istruzione nello stesso modo. I flags CF e OF sono trattati diversamente dalle operazioni sub, add e cmp. Tuttavia, poiché l'operando immediato è 0 i *flags* vengono puliti da ogni operazione.

Il terzo campo è un identificativo mnemonico dell'operazione.

L'implementazione è banale in quanto si tratta di un semplice *pattern matching*. Se l'istruzione equivale ad una delle precedenti, vengono presi tramite *take_bits* 3 bit, il valore di questi bit viene utilizzato come indice per decidere quale istruzione utilizzare. Purtroppo, con 3 bit arriviamo ad avere 8 possibili indici e l'insieme ne contiene soltanto 7, pertanto utilizziamo gli indici da 0 a 5²⁶ per nascondere 3 bits, ovvero i valori da 000 a 101, la settima istruzione verrà utilizzata solamente in presenza di un valore 110 o 111, e conterrà 2 bit ovvero 11, lasciando la codifica del terzo bit ad un'altra istruzione, così da massimizzare il numero di bit nascosti.

²⁶ Ricordiamo che nei linguaggi di programmazione i vettori di dimensione N sono indicizzati a partire da 0 fino ad N-1.

D'ora in poi i set di istruzioni sono validi solamente se gli operandi di destinazione e sorgente sono lo stesso registro.

L'inversione di due registri identici ne crea una nuova, poiché alcune operazioni possono scrivere gli operandi sia in formato "r/m, reg" che in formato "reg, r/m", quindi invertendo il formato otteniamo la stessa identica istruzione ma con *codice macchina* differente.

Il trattamento dei flag non subirà pertanto alcuna modifica.

Vediamo l'elenco di tutte queste istruzioni. Per convenienza verrà riportato un solo insieme per ogni tipologia di istruzione, ma ognuna di esse è presente nelle versioni dei registri a 8, 16, 32 e 64 bits:

```
struct table toa_8_table[] = {
    { {'\x20'}, NF, "and  r/m8 , r8"    },
    { {'\x84'}, NF, "test r/m8 , r8"    },
    { {'\x08'}, NF, "or   r/m8 , r8"    },
    { {'\x22'}, NF, "and  r8   , r/m8"  },
    { {'\x0A'}, NF, "or   r8   , r/m8"  },
};

struct table xor_sub64_s[] = {
    { {'\x48', '\x31'}, NF, "xor  r/m64, r64"    },
    { {'\x48', '\x33'}, NF, "xor  r64  , r/m64"  },
    { {'\x48', '\x29'}, NF, "sub  r/m64, r64"    },
    { {'\x48', '\x2B'}, NF, "sub  r64  , r/m64"  },
};

struct table add32_s[] = {
    { {'\x01'}, NF, "add  r/m32, r32"    },
    { {'\x03'}, NF, "add  r32  , r/m32"  },
};

struct table adc64_s[] = {
    { {'\x48', '\x11'}, NF, "adc  r/m64, r64"    },
    { {'\x48', '\x13'}, NF, "adc  r64  , r/m64"  },
};

struct table cmp16_s[] = {
    { {'\x66', '\x39'}, NF, "cmp  r/m16, r16"    },
    { {'\x66', '\x3B'}, NF, "cmp  r16  , r/m16"  },
};

struct table mov8_s[] = {
    { {'\x88'}, NF, "mov  r/m8 , r8"    },
    { {'\x8A'}, NF, "mov  r8   , r/m8"  },
};

struct table sbb32_s[] = {
    { {'\x19'}, NF, "sbb  r/m32, r32"    },
    { {'\x1B'}, NF, "sbb  r32  , r/m32"  },
};
```

La ricerca e relativa sostituzione di ognuno degli elementi di questi insiemi all'interno del file eseguibile si svolge esattamente nello stesso modo, l'unica differenza è nel *rate* di bit nascosti. Gli insiemi da 4 o 5 elementi possono nascondere 2 bit rispetto agli altri che ne offuscano uno solo. Vediamo in dettaglio la ricerca e sostituzione di uno solo di essi. Prendiamo `xor_sub_table`:

Per prima operazione si verifica che l'istruzione sia una xor o una sub.

```
if( insn.id != X86_INS_XOR && insn.id != X86_INS_SUB ) return error;
```

Va controllato che gli operandi siano due registri e siano anche lo stesso registro.

```
char * reg1 = strtok( op_str, OPERAND_DELIMITATOR );
char * reg2 = strtok( NULL, OPERAND_DELIMITATOR );
if( ( strcmp( reg1, &reg2[1] ) == 0 ) && ( ret = is_reg( reg1 ) ) ) ...
/* si utilizza &reg2[1] poiché il formato degli operandi è "dest, src",
delimitando per ',' si ottiene uno spazio in testa al secondo operando */
```

`is_reg` è una semplice funzione che utilizza un vettore contenente tutti i registri ordinati e ritorna la tipologia di registro: `REG_8_BIT`, `REG_16_BIT`, `REG_32_BIT`, `REG_64_BIT`. In aggiunta indica se si tratta di un registro aggiuntivo aggiungendo il flag `REG_IS_R_TYPE` (solo per le versioni a 64 bit) al valore di ritorno.

Si controlla il valore di ritorno di `is_reg` con un semplice switch, i registri aggiuntivi per le versioni a 64bit sono trattati diversamente poiché utilizzano in testa un prefisso²⁷ aggiuntivo.

```
switch( ret ) {
    case REG_8_BIT:
        bit = take_bits( data, 2 );
        CODE[insn.address] = xor_sub8_s[bit].code;
        return 1;
    case ...
    ...
    case (REG_8_BIT | REG_R_TYPE):
        bit = take_bits( data, 2 );
        CODE[insn.address] = R_REGS_PREFIX;
        CODE[insn.address+1] = xor_sub8_s[bit].code;
    ...
}
```

²⁷ In codice macchina prima del opcode vi possono essere al più 4 bytes aggiuntivi, chiamati prefissi. Questi prefissi indicano varie cose:

dimensione dell'operando, dimensione dell'indirizzo,

REX.W ovvero indica che stiamo eseguendo un'operazione a 64 bit es. `add eax, 1` è `83 c0 01` ma aggiungendo il prefisso 48 si indica `add rax, 1`,

REX.R,B,X funziona in egual modo a REX.W ma indica il registro aggiuntivo equivalente, `eax=r8 ebx=r9` Quindi per indicare `add r8, 1` si aggiunge il prefisso 49 ad `add eax, 1` ovvero `49 83 c0 01`.

In ultimo luogo vediamo un metodo ideato da me, chiamato CMP_RR. Quest'ultimo si basa sull'enorme utilizzo dell'istruzione `cmp` per determinare se il contenuto di due registri è uguale. Quando si processa una `cmp` la CPU effettua una sottrazione tra i due operandi ma senza modificare il primo come nella `sub`. Pertanto, invertendo i due registri se i contenuti sono uguali il risultato rimane invariato. Non ci resta che determinare se l'operazione `cmp` analizzata è utilizzabile, applicando un *check_flag* di OF, CF e SF ovvero coloro che sono trattati in maniera differente invertendo i registri, possiamo determinarne la validità.

```
char *reg1 = strtok( insn[index].op_str, OPERAND_DELIMITATOR);
char *reg2 = strtok( NULL, OPERAND_DELIMITATOR);
ret2 = is_reg(&reg2[1]);
ret = is_reg(reg1);
if( (ret>=0) && (ret2>=0) ){
    if( !check_flag( insn, (OF|CF|SF), index, num_ists) ) return error;
    // Istruzione valida, segue il camuffamento...
    sprintf( reverse_instr, "%s %s, %s", insn[index].mnemonic,
                                                    &reg2[1], reg1);
    ks_asm(ks, reverse_instr, 0, &ks_output, &ks_size, &ks_count)
    // sostituzione...
}
```

Come viene riconosciuta l'istruzione che indica 0 e quella che indica 1?

Per fare ciò ho attribuito ad ogni registro un valore intero, reperibile grazie alla funzione *which_reg* che diversamente da *is_reg* la quale ritorna il gruppo di appartenenza (8, 16, 32, 64 bit), restituisce il valore del registro, -1 altrimenti.

Non resta che determinare quale registro sia il maggiore e posizionarlo a sinistra per nascondere il bit 0, a destra per nascondere 1.

Abbiamo così analizzato tutte le tecniche utilizzate da *hidder*, pertanto possiamo ritorniamo al punto 8, per proseguire con l'iterazione.

Avendo così descritto il funzionamento delle parti principali del nostro *hidder*, passiamo ora ad un'analisi approfondita per trarne delle conclusioni.

5. CAPITOLO 4: Analisi e conclusioni

5.1 Data rate

Il data rate di *hider* è di 1 bit offuscato ogni 120 di codice ($\frac{1}{120}$), attualmente sono state implementate le tecniche di sostituzione di istruzioni equivalenti, *cmp_rr* e sostituzione di immediati in operazioni aritmetiche consecutive.

Nelle figure di seguito sono riportati in blu il data rate utilizzando sia le istruzioni equivalenti, *cmp_rr* e le operazioni consecutive, in arancione utilizzando soltanto le istruzioni equivalenti. Nelle seguenti tabelle sono indicati soltanto alcuni file campione presi dall'insieme dei file utilizzati per eseguire i test.

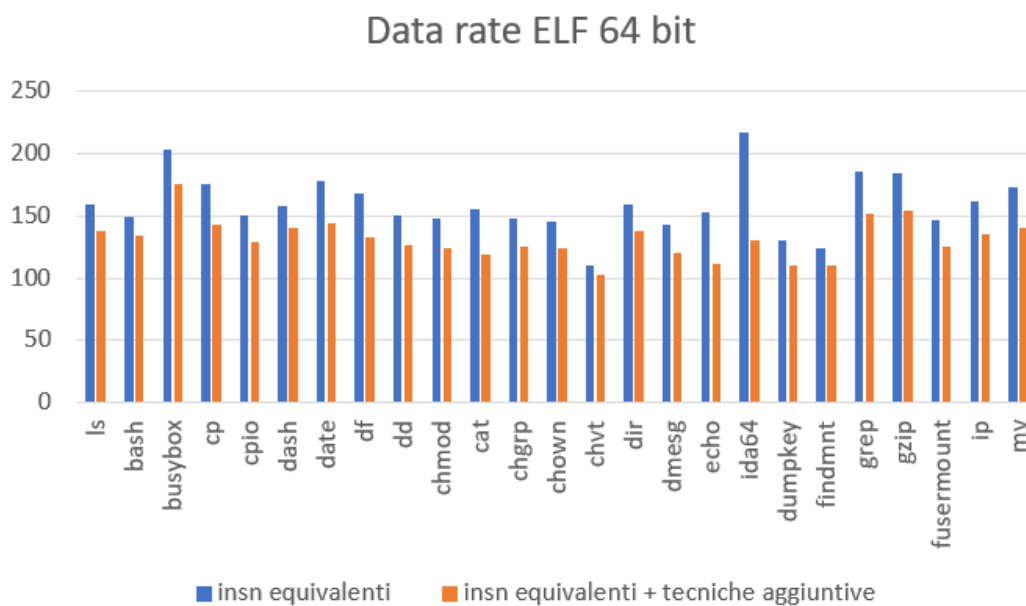


FIGURA 9 RATEO DI OFFUSCAMENTO NEGLI ELF 64 BIT

Nei file ELF 64 bit si è ottenuto un rateo di $\frac{1}{122}$ (arancione) applicando istruzioni equivalenti, *cmp_rr* e l'offuscamento negli immediati di operazioni consecutive, e $\frac{1}{159}$ usufruendo solamente delle istruzioni equivalenti.

Data rate ELF 32 bit

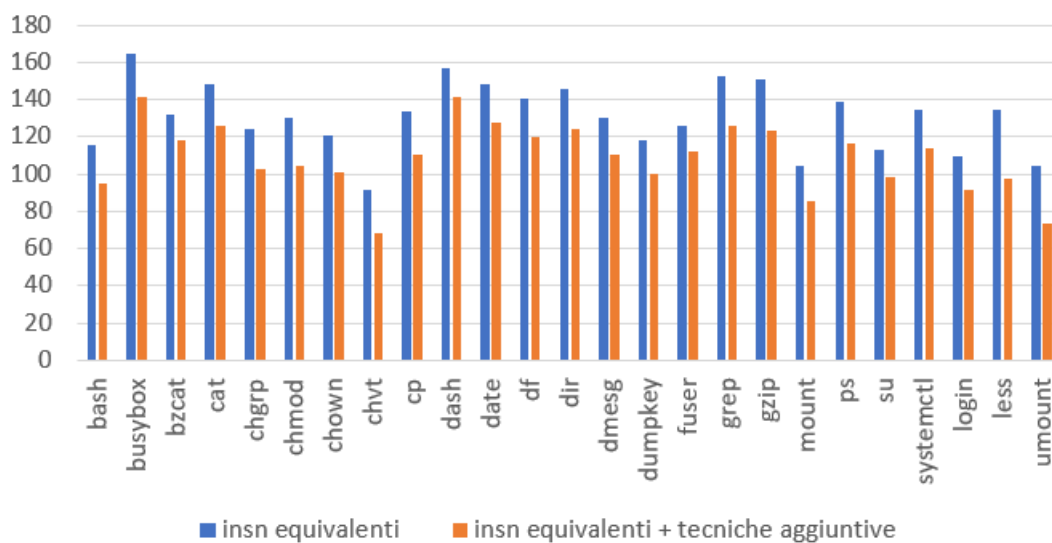


FIGURA 10 RATEO DI OFFUSCAMENTO NEGLI ELF 32 BIT

Nei file ELF 32 bit si è ottenuto un rateo di $\frac{1}{109}$ (arancione) e $\frac{1}{130}$ (blu).

Data rate PE32, PE32+

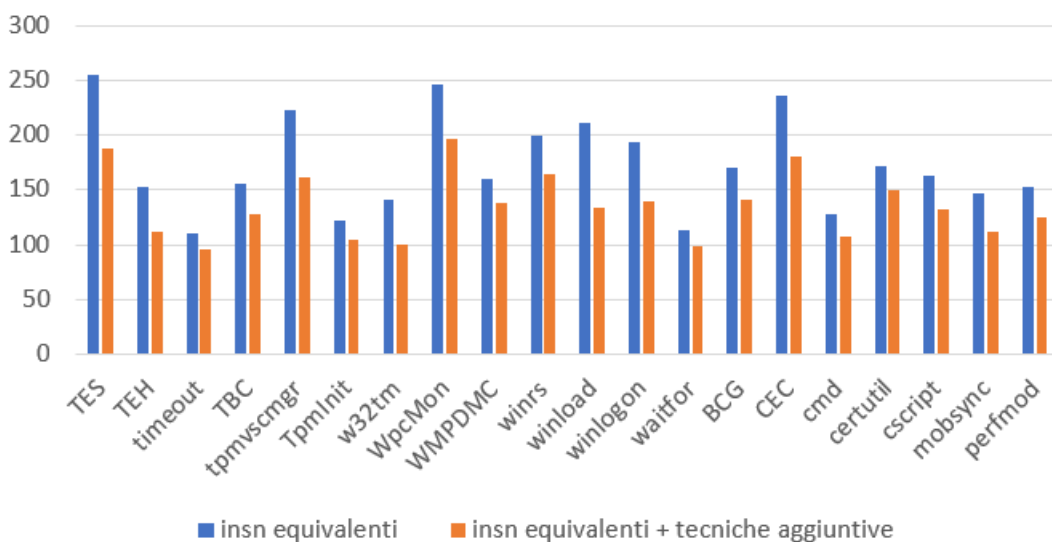


FIGURA 11 RATEO DI OFFUSCAMENTO NEI PE32 E PE 32+

Nei file PE si è ottenuto un rateo di $\frac{1}{130}$ (arancione) e $\frac{1}{170}$ (blu).

5.2 Future prospettive di lavoro

Abbiamo visto fino ad ora l'utilizzo di due tecniche per nascondere bit: Alternanza di istruzioni funzionalmente equivalenti e sostituzione degli immediati in operazioni aritmetiche consecutive.

In realtà esistono varie altre tecniche applicabili mantenendo le due regole fondamentali di *hidder*.

- Ottenere un file eseguibile funzionalmente identico al file di partenza.
- Mantenere la lunghezza invariata.

Se non si vuole mantenere il punto sulla lunghezza invariata del file eseguibile, esistono un'infinità di tecniche ed anche qualche software di camuffamento già esistenti, per esempio [1] Nisha P. Shetty, Nikhil Ranjan (2016), definisce uno di essi, nascondendo il messaggio alla fine di un file eseguibile. Altri nascondono il messaggio in sezioni fittizie, create solamente per contenere dati, oppure nella sezione *rsrc*.

Purtroppo queste tecniche non sono minimamente sicure, anche un'analisi superficiale potrebbe trovare immediatamente questi messaggi ed anche qualche antivirus, come descritto in [10] Michael Sikorski and Andrew Honig (2012), *practical malware analysis*.

Tra le prime tecniche di analisi statica di malware si controllano *header*, dimensione file e librerie linkate, per poi passare al controllo delle sezioni con software come *Resource hacker* o *PE view*.

Passiamo ora ad analizzare altre due tecniche potenzialmente sicure:

Il rimpiazzo di *dead-code*²⁸ con istruzioni appositamente create per nascondere bit. Andando ad eliminare codice, la ricerca del *dead-code* deve risultare accurata al 100% per non corrompere il funzionamento del file eseguibile. Una semplice implementazione potrebbe essere quella di simulare il flusso di esecuzione del programma seguendo ricorsivamente ogni jump e chiama a funzione, mettendo un check ad ogni istruzione che viene eseguita. Infine, si sostituisce un minimo numero di codice sicuramente inutilizzato.

In [11] Denys Vlasenko, (2010), viene mostrato come eliminare funzioni non raggiungibili dal programma e dati inutilizzati per un totale variabile dal 2% fino ad un massimo del 10% del codice e dei dati. In media circa il 50% delle librerie linkate staticamente non vengono utilizzate, queste sostituzioni porterebbero un incremento sostanziale al *data rate* di *hidder*.

La seconda tecnica è quella suggerita ed analizzata da [2] El-Khalil, R.: Hydan (2004), la quale consiste nel riordino dei blocchi di codice funzionalmente indipendenti, come ad esempio le funzioni. Per fare ciò, un possibile metodo consiste nel riordinare i blocchi secondo un ragionamento fisso, ottenendo dallo stesso insieme di blocchi sempre lo stesso ordine. Indichiamo con N l'insieme da riordinare e con n il numero di blocchi, sappiamo che il numero di possibili ordinamenti è $n * (n - 1) * (n - 2) * ... * 1$ ovvero $n!$, se attribuiamo ad

²⁸ Il *dead-code* o codice morto è una parte del codice di un programma che non verrà mai raggiunto da nessun flusso di esecuzione e pertanto eseguito.

ogni ordinamento un valore possiamo nascondere $N_{bit} = \log_2(n!)$ arrotondato per difetto. Es. $n = 10 \rightarrow N_{bit} = 21$, dobbiamo pertanto determinare una funzione biunivoca²⁹ tra l'ordine degli N blocchi e il valore corrispondente.

Un possibile algoritmo è il seguente pseudocodice:

Input: N, n, VAL // valore da nascondere

Output: N^* // ovvero una copia di N ordinata secondo VAL

if $\log_2 VAL > \log_2 n!$ **then:** // entrambi i logaritmi arrotondati per difetto

return "impossibile nascondere il valore" ;

Ordiniamo i blocchi di N secondo un ordinamento logico; // es grandezza

Creiamo una copia N^* di N ;

Inizializziamo $i=0$;

Scomponiamo VAL in fattoriali ottenendo $k * (n-1)! + k^1(n-2)! + \dots + k^{n-2} * 1!$;

foreach K **in** $(k, k^1..k^{N-2})$ **then:**

$N^*[i] = N[K]$;

 riordino N senza $N[K]$; // semplicemente spostato i valori da $N[K+1]$ in poi a sinistra di uno
 $i = i + 1$;

end

return N^* ;

Vediamo un esempio esplicativo: se $n = 5$ allora $N_{bit} = 6$ se $VAL = 110110$ (54), la sua scomposizione è: $2 * 4! + 1 * 3! + 0 * 2! + 0 * 1!$. Fingiamo che N riordinato sia una stringa ed ogni blocco una lettera: "abcde", i K sono 2, 1, 0, 0.

Come prima operazione si mette $N[2]$ in prima posizione, ottenendo $N^* = "c"$ e $N = "abde"$.

Nella seconda iterazione $N[1]$ va in seconda posizione, ottenendo $N^* = "cb"$ e $N = "ade"$..

Infine avendo due 0 si aggiunge N in coda ad N^* ottenendo il riordino $N^* = "cbade"$.

Il procedimento inverso è simile, prendendo il messaggio N^* criptato, si riordina ottenendo N , una volta in possesso di N e N^* ricaviamo i valori di K della scomposizione, e di conseguenza VAL .

A questo punto non ci resta che trovare ciò che può essere riordinato e analizzarlo. In [2] El-Khalil, R.: Hydan, hanno effettuato questo lavoro per noi, misurando che in media in un programma ci sono circa 315 funzioni, riordinando .got .plt .data .bss .ctors .dtors la sh_tbl (tutti questi campi hanno un equivalente anche per il formato PE), secondo *hydan* si riesce a quadruplicare il rateo di bit nascosti in un file. Utilizzando le istruzioni equivalenti e il riordino dei file si arriva ad un rateo di $\frac{1}{36}$ ovvero 1 bit nascosto ogni 36 bit di codice.

²⁹ Una corrispondenza biunivoca tra due insiemi è una funzione $F : A \rightarrow B \mid F(a) = b \ \forall a \in A \text{ e } b \in B$ che associa ad ogni elemento di A un elemento di B e viceversa.

5.3 Steganalisi

Come per la crittografia esiste il requisito di essere impossibilitati a leggere il messaggio senza il possesso della chiave, anche per la steganografia esiste lo stesso requisito, ovvero essere impossibilitati di comprendere se il contenitore nasconda un messaggio o meno senza la conoscenza della chiave.

L'analisi per determinare se ad un messaggio è stato applicato qualche algoritmo di steganografia è chiamata steganalisi, il suo scopo è soltanto quello di determinare o meno l'offuscamento e non di decifrare in chiaro il messaggio nascosto.

Vedremo che è possibile scovare l'utilizzo del *software hider* su di un file e come evitare che ciò avvenga.

Come sappiamo *hider* si basa su diversi insiemi di istruzioni equivalenti: TOA, xor_sub, add_sub e altri. Vediamo nelle seguenti figure la distribuzione di queste istruzioni all'interno di file ELF 64,32 bit e PE32,32+. Per eseguire queste analisi sono stati utilizzati circa 100 file eseguibili per ogni tipologia di file e per ogni versione.

Troviamo la distribuzione all'interno dei file ELF 64 bit nella **Figura 10**, notiamo che la distribuzione vede un'affluenza totalitaria delle istruzioni XOR e TEST nei rispettivi insiemi xor_sub e TOA. Le istruzioni dei restanti insiemi invece sono sufficientemente bilanciate.



Figura 12 Distribuzione delle istruzioni negli ELF 64 bit

Troviamo la distribuzione delle istruzioni dei file ELF32 bit in **Figura 11**, notiamo che le istruzioni XOR e TEST dei rispettivi insiemi xor_sub e TOA ricoprono il 99,9% della distribuzione dei loro rispettivi insiemi.

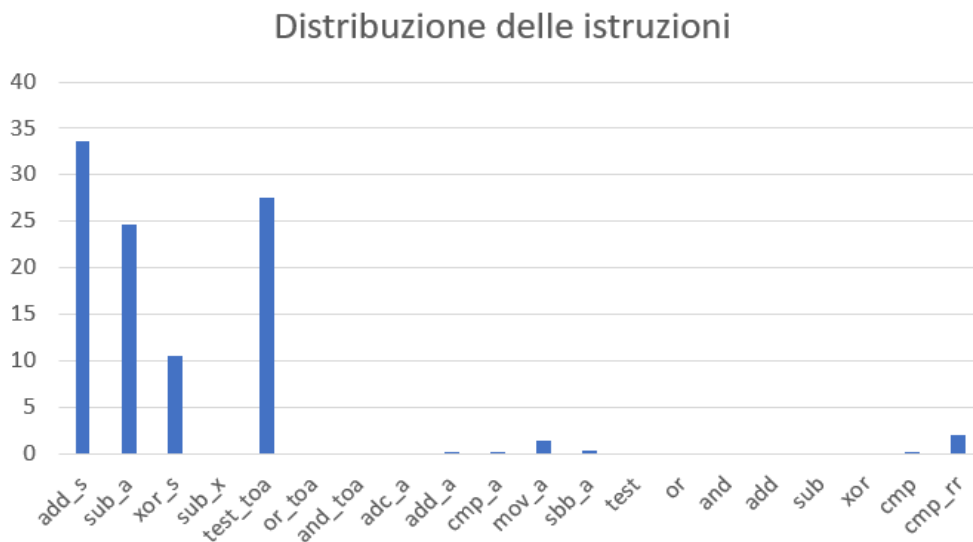


FIGURA 13 DISTRIBUZIONE DELLE ISTRUZIONI NEGLI ELF 32 BIT

Le distribuzioni delle istruzioni all'interno dei file PE32 e PE32+ in **Figura 12**, sono state riportate in un'unica tabella poiché i risultati si sono rilevati analoghi. Vediamo anche qui un'elevata predominanza delle istruzioni XOR e TEST dei loro rispettivi set di istruzioni equivalenti.

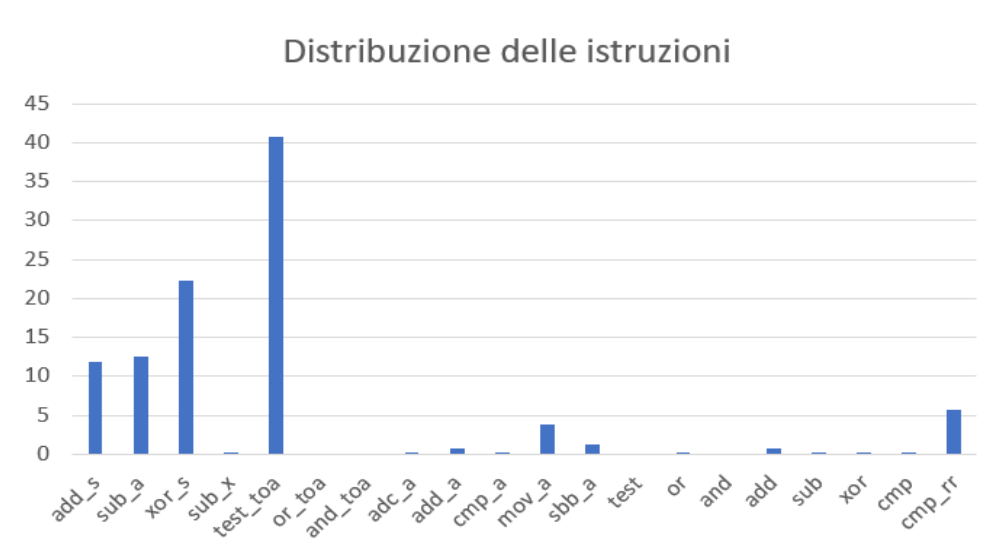


FIGURA 14 DISTRIBUZIONE DELLE ISTRUZIONI NEI PE32 E PE32+

L'utilizzo di *hider* su questi file genera nel codice modifiche di diverse proprietà del file eseguibile di partenza. Non viene cambiata la dimensione ma bensì la distribuzione delle istruzioni, vediamo nelle seguenti tabelle la distribuzione dei file applicando un offuscamento totale. Ovvero se il contenitore può nascondere 52 bytes offuscheremo un file di 48 bytes poiché la dimensione del *ciphertext* deve essere multiplo del *block size* di AES (16 bytes).



FIGURA 15 DISTRIBUZIONE DOPO L'UTILIZZO DI *HIDDER* NEGLI ELF 64 BIT

La precedente distribuzione (Figura 15) si riferisce ai file ELF, notiamo fin da subito che la distribuzione viene divisa quasi equamente tra le varie istruzioni di ogni insieme.



FIGURA 16 DISTRIBUZIONE DOPO L'UTILIZZO DI *HIDDER* NEI PE32, PE32+

Come per i file ELF anche nei file PE le istruzioni vengono ripartite quasi equamente. Vediamo che l'istruzione AND di TOA viene utilizzata più frequentemente, ma questo dipende solamente dal *ciphertext* ottenuto.

Possiamo trarre in conclusione che la distribuzione delle istruzioni nei file dopo l'utilizzo di *hidder* è chiaramente diversa da quella di partenza. Questo comporta che l'elusività di *hidder* dipende molto dalla grandezza del messaggio da nascondere.

Eseguendo dei test sulla distribuzione delle istruzioni di un file è facilmente individuabile l'utilizzo di un *software* di steganografia su file eseguibile utilizzando la tecnica della sostituzione di istruzioni equivalenti, quest'analisi comporta la scoperta di codice nascosto con rateo del 100% come affermato da [3] Jorge Blasco(2009).

Una metodologia per evitare questa individualizzazione è quella di mantenere la distribuzione del file di partenza e quindi utilizzare prima le istruzioni bilanciate come TOAASXC, AACMS e add_sub. Notiamo che la metodologia CMP_RR non ha modo di essere determinata poiché l'ordine dei registri usati cambia da compilatore a compilatore.

Un altro metodo proposto in [16] suggerisce di creare un "*random walk*" tradotto letteralmente in camminata casuale, prevede di prendere solo alcune delle istruzioni realmente utilizzabili, seguendo un *pattern*. Nel nostro caso potremmo utilizzare il *digest* come *seed* in modo tale da ottenere la stessa camminata casuale sia nel camuffamento che nel deoffuscamento. Si consiglia di utilizzare un range $[0, 2 * \frac{C_{Len}}{P_{Len}}]$, dove con C_{Len} indichiamo il numero di *bit* ancora offuscabili nel contenitore e con P_{Len} il numero di *bit* del *plaintext* ancora da nascondere. Una volta calcolato si genera casualmente un numero N compreso nel range e si "saltano" N *bit* prima di utilizzare un'istruzione per il camuffamento.

Vediamo come si comportano le altre metodologie di offuscamento rispetto ad una steganalisi:

Ogni forma di riordino di blocchi indipendenti come descritto precedentemente, mantiene un alto grado di furtività, sono pochi i controlli di cui si deve tenere traccia per non essere individuati. Un esempio è l'ordine delle *push* e *pop*, molti produttori come IBM utilizzano un ordine specifico di *push* e *pop* per evitare pirateria del *software* [17]. Per il resto l'ordine dipende molto dal codice sorgente pertanto il livello di casualità è molto elevato e difficile da analizzare per poter determinare una steganografia.

Per quanto riguarda il camuffamento del *dead-code*, il tasso di furtività è deciso totalmente del *software* che genererà il sostituto per il *dead-code*. Progettando il codice sostitutivo a partire dal messaggio secondo un ragionamento logico, ovvero seguendo un *pattern* e non posizionando una serie di istruzioni casuali; utilizzando istruzioni con alta frequenza che risultino non sospette, ma soprattutto istruzioni che si combinino fra loro simulando un codice che effettivamente segue un ragionamento. Si ottiene così un codice pulito ed innocuo ma soprattutto che contiene bit nascosti.

5.4 Conclusioni

Data l'esautiva analisi delle varie tecniche di offuscamento di dati all'interno di file eseguibili è possibile riscontrare tale metodologia come una tecnica con chiari vantaggi. Attraverso la realizzazione di un software che implementi tutte e 4 le tecniche esposte, ovvero:

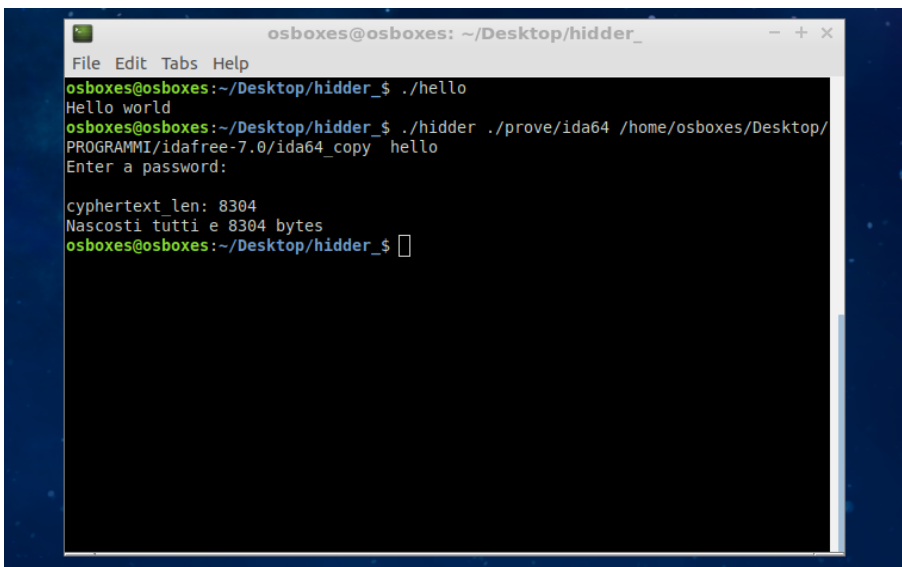
Sostituzione di istruzioni equivalenti, offuscamento negli immediati di operazioni aritmetiche consecutive, sovrascrittura del dead-code ed infine il riordino di blocchi funzionalmente indipendenti.

Si può calcolare con delle stime approssimative un software che nasconda informazioni in file eseguibili in modo furtivo con un rateo di circa 1 bit ogni 19 di codice $\frac{1}{19}$, che può essere ulteriormente migliorato se aggiunte altre sezioni da riordinare o altri insiemi di istruzioni equivalenti.

Il rateo medio di bit nascosti all'interno di un JPEG è di $\frac{1}{17}$ [2], considerando gli innumerevoli studi della steganografia sulle immagini e grazie all'enorme ridondanza dei bit in esse, possiamo ritenerci soddisfatti dei risultati ottenuti ed affermare che la steganografia su file eseguibili, senza modificare le relative dimensioni è possibile, e seguendo l'iter ideato nella tesi è possibile ottenerlo anche con ottimi risultati.

Ricordiamo inoltre che per messaggi di dimensioni ridotte il tasso di furtività è molto elevato, pertanto queste tecniche potrebbero essere molto utili per evitare la pirateria, attribuendo *fingerprint* e *watermark* al file eseguibile. Per esempio offuscando i codici di licenza all'interno di un software, riordinando determinati blocchi indipendenti come l'ordine di richiesta di alcune query ad un server si può fornire al server il proprio codice di licenza.

Di seguito vediamo una serie di immagini auto esplicative di un'esecuzione molto semplice di *hider* utilizzando come *plaintext* un semplice binario *hello world*, "hello", già compilato, e come contenitore un file eseguibile, in questo caso la versione 64 bit di IDA.



```
osboxes@osboxes: ~/Desktop/hider_
File Edit Tabs Help
osboxes@osboxes:~/Desktop/hider_$ ./hello
Hello world
osboxes@osboxes:~/Desktop/hider_$ ./hider ./prove/ida64 /home/osboxes/Desktop/
PROGRAMMI/idafree-7.0/ida64_copy hello
Enter a password:

cyphertext_len: 8304
Nascosti tutti e 8304 bytes
osboxes@osboxes:~/Desktop/hider_$
```

FIGURA 17 AVVIO HIDER TRAMITE BASH

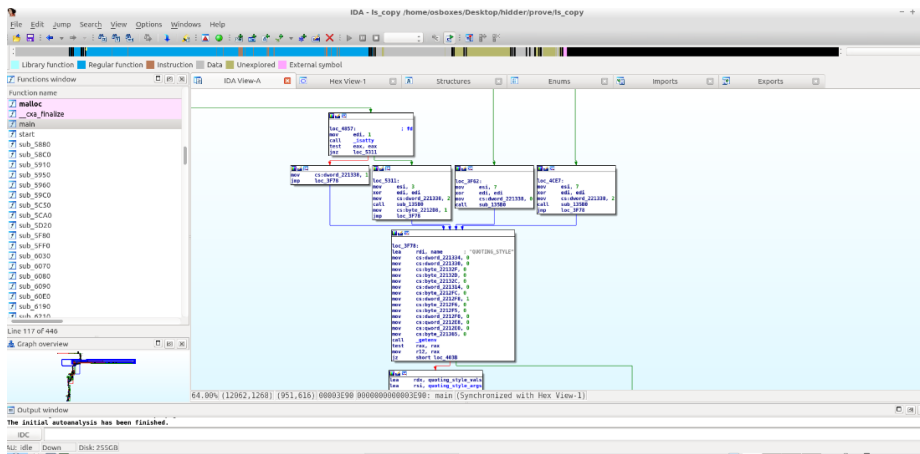


FIGURA 18 DIMOSTRAZIONE DEL CORRETTO FUNZIONAMENTO DI IDA_COPY

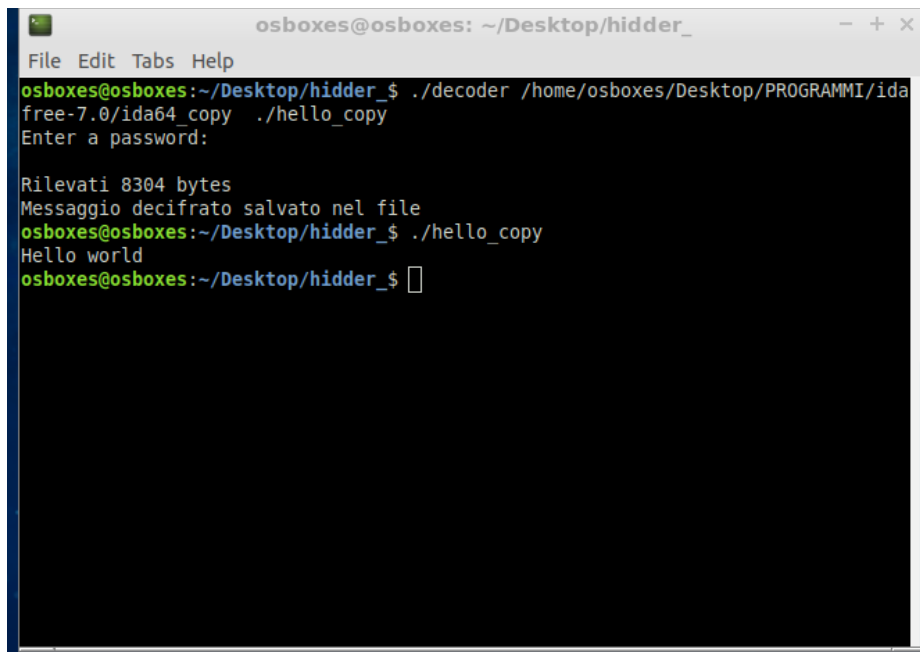


FIGURA 19 AVVIO DECODER E AVVIO DEL FILE HELLO_COPY

6. Bibliografia e Sitografia

- [1] Nisha P. Shetty, Nikhil Ranjan (2016), Using Steganography & Cryptography to hide data in EXE files, International Journal of Engineering & Technology.
- [2] El-Khalil, R.: Hydan: Hiding Information in Program Binaries (2003). Lecture Notes in Computer Science 3269, 187–199 (2004) <http://crazyboy.com/hydan/>. Cited 20 Oct 2008
- [3] Jorge Blasco, Julio C. Hernandez-Castro, Juan M.E. Tapiador, Arturo Ribagorda and Miguel A. Orellana-Quiros (2009), Steganalysis of Hydan
- [4] Hanna Willa Dhany, Fahmi Izhari, Hasanul Fahmi, Tulus, Sutarman (2017), Encryption and Decryption using Password Based Encryption, MD5, and DES
- [5] <https://zachgrace.com/posts/attacking-ecb/> , Attacking ECB, ultimo accesso 21/09/2020
- [6] <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format> , PE Format, ultimo accesso 21/09/2020
- [7] https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm , Assembly-Registers, 21/09/2020
- [8] <https://github.com/aquynh/capstone> , CAPSTONE, ultimo accesso 19/08/2020
- [9] <https://github.com/keystone-engine/keystone> , KEYSTONE, ultimo accesso 19/08/2020
- [10] Michael Sikorski and Andrew Honig (2012), PRACTICAL MALWARE ANALYSIS The Hands-On Guide to Dissecting Malicious Software
- [11] Denys Vlasenko, (2010) Link time dead code and data elimination using GNU toolchain
- [12] Kwan, M.: gifshuffle. <http://www.darkside.com.au/gifshuffle/> (2003)
- [13] A.A.Zaidan, B.B.Zaidan, M.M.Abdulrazzaq, R.Z.Raji and S.M.Mohammed, "Implementation Stage for High Securing CoverFile of Hidden Data Using Computation between Cryptography and Steganography". International Association of Computer Science and Information Technology (IACSIT), indexing by Nielsen, Thomson ISI (ISTP), IACSIT Database, British Library and EI Compendex, Volume 20, 2009, Manila, Philippines.
- [14] Bertrand Anckaert, Bjorn De Sutter, Dominique Chagnet and Koen De Bosschere, Steganography for Executables and Code Transformation Signatures. Information Security and Cryptology - ICISC 2004. Springer-Verlag Berlin Heidelberg. Lecture Notes in Computer Science. Vol. 3506 (3506). 2005. pp. 425-439
- [15] Johnson N.F., Jajodia S.: Exploring steganography: Seeing the unseen. Computer 31(2), 26–34 (1998).

- [16] Provos, N.: Defending Against Statistical Steganalysis. In: Proceedings of the 10th USENIX Security Symposium. (2001)
- [17] Council for IBM Corporation: Software birthmarks. Talk to BCS Technology of Software Protection Special Interest Group (1985)