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

SAPIENZA
UNIVERSITÀ DI ROMA

Michele Donvito
Matricola 1710775

Relatore
Massimo Bernaschi

Introduzione

- L’avvento dei dati digitali ha permesso una ripresa dello sviluppo delle tecniche di steganografia.
- Principalmente si effettua su foto, video ed audio.
- Il nostro scopo: determinare la possibilità di applicare steganografia all’interno di altri dati digitali, come i file eseguibili ed in particolare nel codice macchina.
- Partiremo come base da *Hydan,* il primo progetto funzionante che ha proposto tecniche per offuscare bit all’interno del codice macchina in file eseguibili a 32 bit.

Introduzione

- Ho progettato ed implementato un software, chiamato *hidder*, il quale dato un file eseguibile ed un messaggio, produce un secondo file eseguibile funzionalmente identico al primo ma contenente il messaggio offuscato all'interno del codice macchina.

- *Hidder* nasce con l'idea di verificare la portabilità di *hydan* sulle versioni a 64 bit e di determinare ulteriori tecniche di camuffamento all'interno del codice macchina.

- El-Khalil, R.: Hydan: Hiding Information in Program Binaries (2003). http://crazyboy.com/hydan/

Teoria di fondo

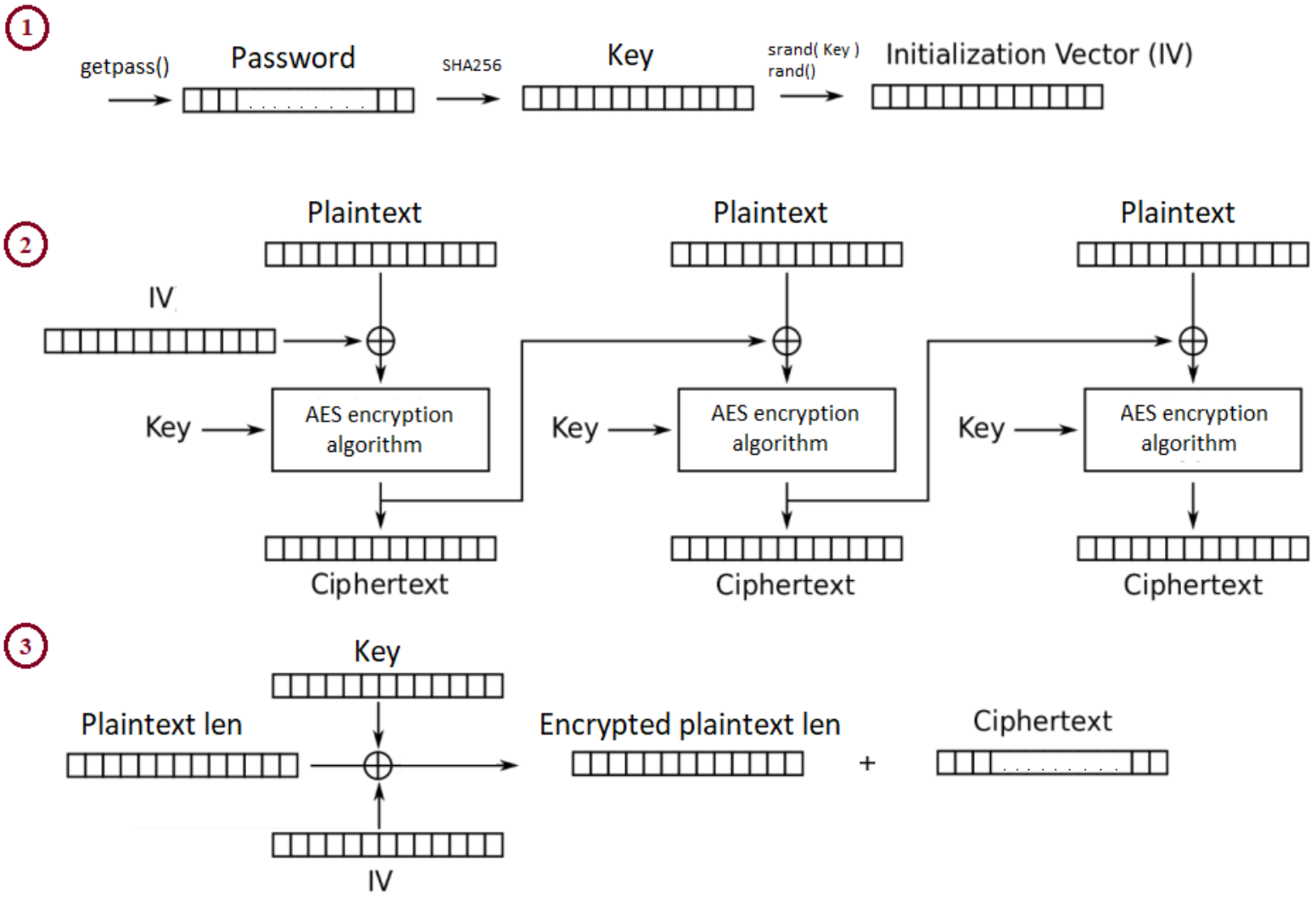
- Per criptare il testo in chiaro viene applicato il metodo classico *Password Based Encryption (PBE).*

- Il PBE è un metodo crittografico a chiave simmetrica, il quale utilizza un testo detto password come chiave ottenuta tramite un algoritmo di *hash*. Questa chiave viene poi utilizzata per criptare il testo in chiaro in un testo cifrato.

- La stessa tecnica è utilizzata per decriptare.

- Lo SHA256 è l'algoritmo di *hash* crittografico utilizzato.
  L'AES è l'algoritmo di cifratura utilizzato.
  La modalità di funzionamento dei cifrari utilizzata è la CBC.

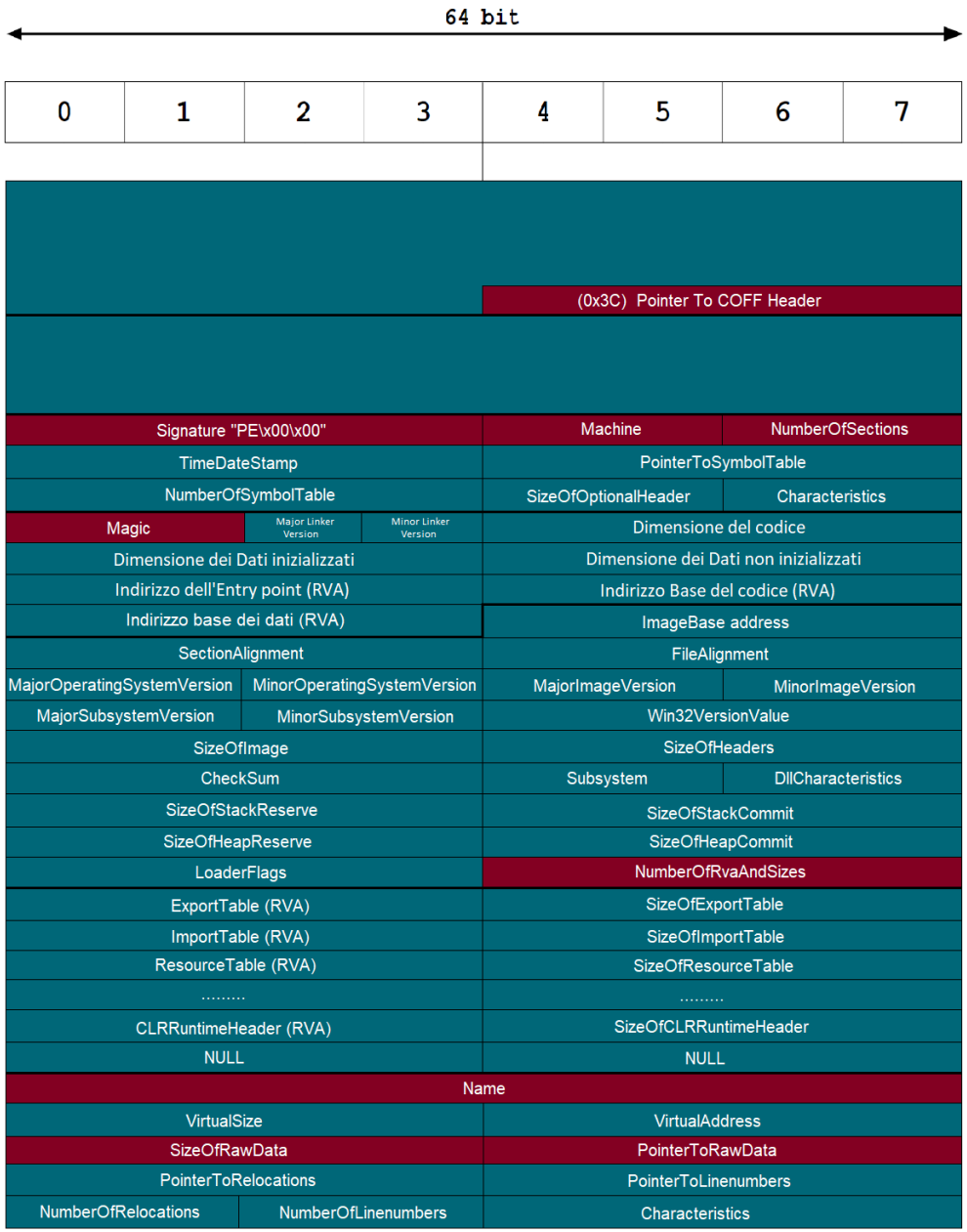
Teoria di fondo



Teoria di fondo

- *Hidder* può offuscare dati all'interno di file eseguibili dei Sistemi Operativi *Windows* e *Linux*.
- I formati dei file eseguibili *Windows* e *Linux* sono rispettivamente PE ed ELF entrambi nelle versioni a 32 e 64 bit.
- Come estrarre il codice macchina da un file eseguibile?
- Entrambi i formati sono basati su una divisione in sezioni, lo scopo è di reperire la sezione contenente il codice macchina.

PE File Format



Introduzione Tecniche di offuscamento

Principalmente utilizziamo tre tecniche di sostituzione:

- Di istruzioni distinte ma funzionalmente identiche.
- Di istruzioni aventi due formati di scrittura.
- Degli immediati di istruzioni aritmetiche consecutive.

Sostituzione Istr. equivalenti

- Questo metodo utilizza la ridondanza all'interno dell'instruction set dell'architettura x86.
- Per esempio, per sommare il valore 50 a RAX possiamo utilizzare sia l'istruzione ADD rax, 50 che SUB rax, -50.
- Utilizzando queste alternative di scrittura possiamo codificare un bit di informazioni, attribuendo ad ADD il valore 0 e a SUB il valore 1.

Sostituzione Istr. equivalenti

Offuscamento 00	Offuscamento 10

add eax, 20
cmp eax, 50
ja LABEL_1
add eax, 10
	sub eax, -20
	cmp eax, 50
	ja LABEL_1
	add eax, 10

Offuscamento 01	**Offuscamento 11**

add eax, 20
cmp eax, 50
ja LABEL_1
sub eax, -10
	sub eax, -20
	cmp eax, 50
	ja LABEL_1
	sub eax, -10

Sostituzione Istr. equivalenti

- Le istruzioni equivalenti possono essere diverse in piccoli dettagli, come l'interazione con il registro EFLAG.
- Nelle istruzioni ADD e SUB il registro EFLAG può essere modificato diversamente nei flag di *Overflow (OF)* e *Carry (CF)*.
- Cosa possiamo determinare la validità?

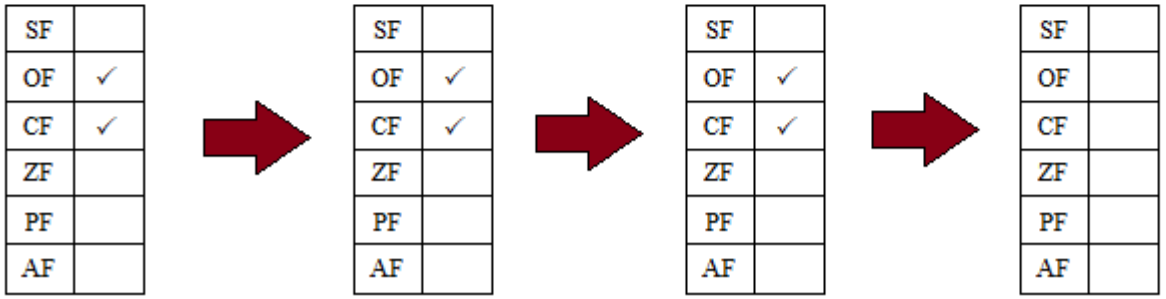
Sostituzione Istr. equivalenti

- check OF
- check CF

- reset CF
- reset OF
- mod SF
- mod ZF
- ind AF

- test ZF
- test OF
- test SF

ADD RAX, 20	MOV RCX, RAX	LEA RAX,[R8]	TEST RAX, RCX	JG LABEL_



Sostituzione del Formato

- In questo metodo le istruzioni che discuteremo sono utilizzabili solamente se gli operandi di destinazione e sorgente sono lo stesso registro.
- Prendiamo per esempio l'istruzione XOR eax, eax, questa può essere indicata sia nel formato XOR r32, r/m32 che XOR r/m32, r32.
- Il cambio di formato è valido per ogni dimensione del registro.
- Le istruzioni sulle quali possiamo applicare questo metodo sono molteplici, come ADD, SUB, CMP, MOV ed altre.

Sostituzione istruzione e Formato

- Ci sono anche insiemi di istruzioni equivalenti che utilizzano i molteplici formati come alternative.

- Un esempio sono i seguenti insiemi *TOA e XOR_SUB:*

```
test    r/m , r
or      r/m , r
or      r , r/m
and     r/m , r
and     r , r/m
```

```
xor     r/m , r
xor     r , r/m
sub     r/m , r
sub     r , r/m
```

Sostituzione istr. Aritmetiche consecutive

- Questa tecnica si basa su un'inefficienza, ovvero l'utilizzo di più istruzioni quando è sufficiente una sola.

- Riprendiamo le nostre istruzioni di ADD e SUB e vediamo il seguente codice:

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

```
add     rax, 50
```

Sostituzione istr. Aritmetiche consecutive

- Alterando gli immediati di queste operazioni, possiamo ottenere in rax lo stesso risultato ed allo stesso tempo nascondere N-1 bit di informazioni, dove N è la grandezza del registro in questione.

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

- Quali controlli vanno effettuati per validare queste due istruzioni?

Sostituzione istr. Aritmetiche consecutive

- Il primo controllo in assoluto è che il registro non sia utilizzato nell'intervallo tra le due operazioni.
- Un secondo controllo è che non vi siano LABEL nell'intervallo.
- Il terzo controllo è l'utilizzo del registro EFLAG nelle istruzioni che seguono le due operazioni.

```
        add     rax, MEX
        mov     rcx, 1
        push    rcx
        sub     esp, 4
LABEL:
        add     rax, 50-MEX
        jmp     LABEL_
```

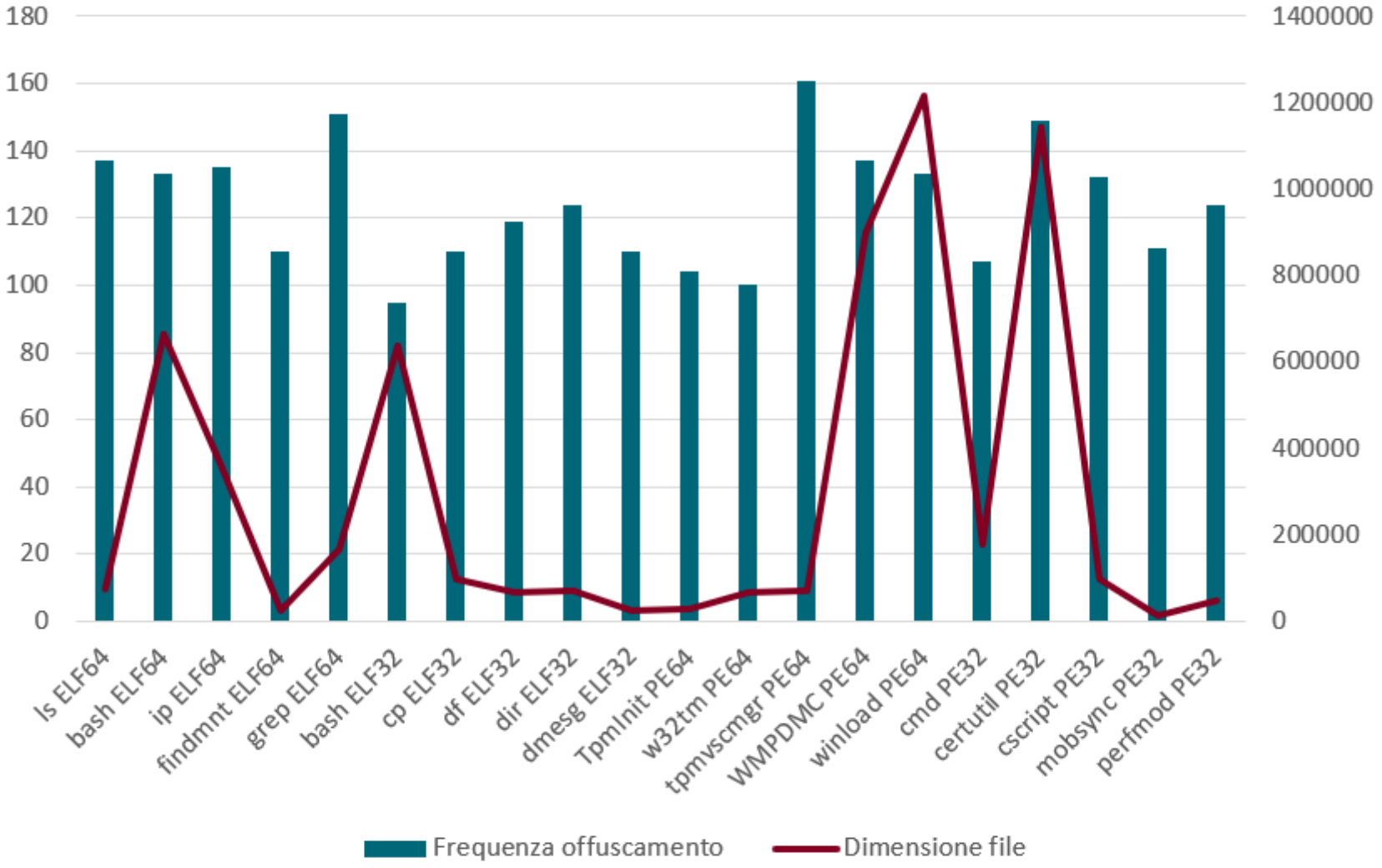
Sostituzione CMP tra registri

- Presentiamo CMP_RR, un nuovo metodo alternativo di offuscamento progettato da me.
- L’istruzione CMP utilizzata su due registri per verificarne l’uguaglianza è di uso comune.
- In questo utilizzo possiamo invertire gli operandi ottenendo lo stesso risultato.
- Per validare l’inversione dei registri verifichiamo con il metodo precedentemente esposto impostando il vettore dei flag da ricercare con *Overflow, Carry e Sign*.

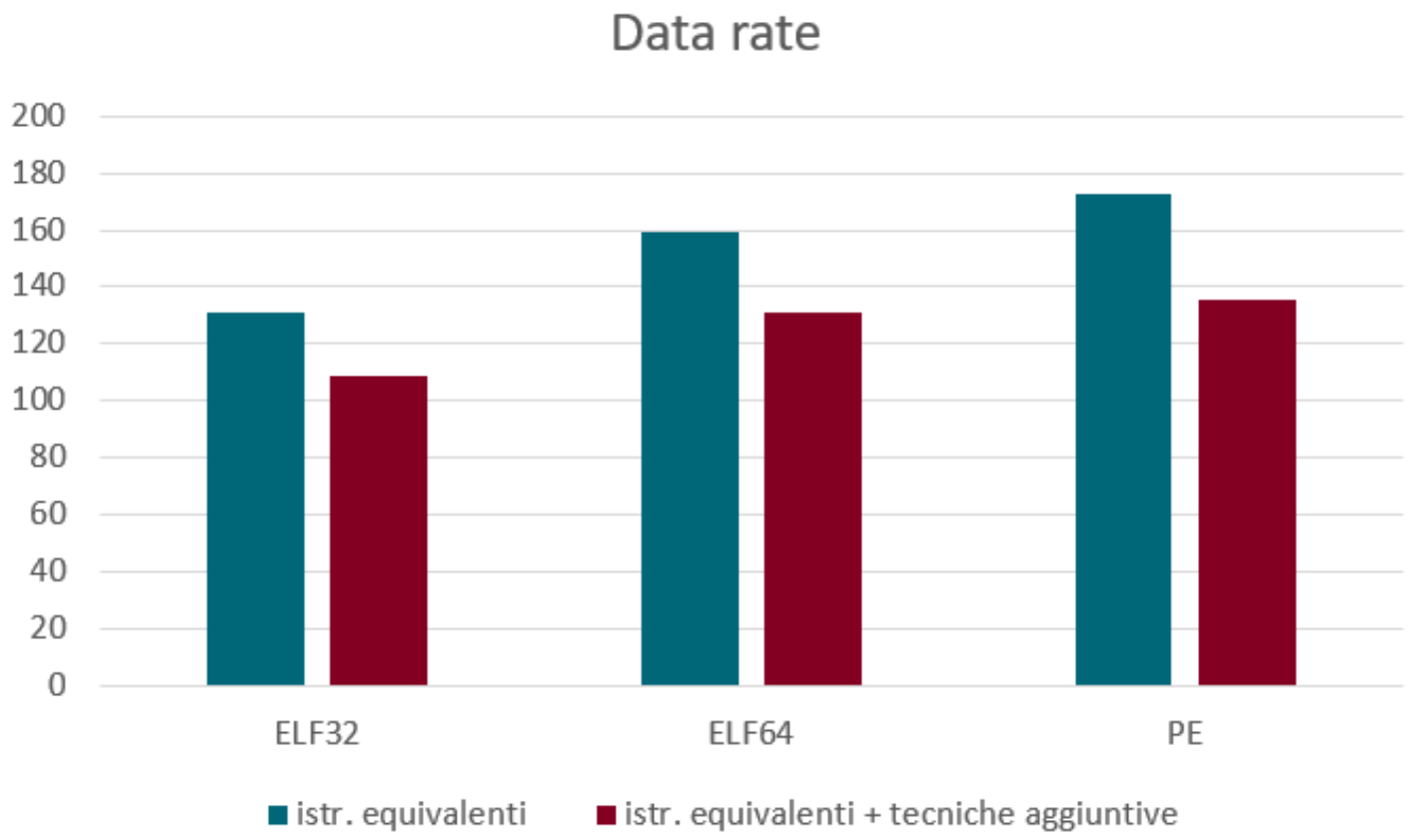
Frequenza di offuscamento

- La frequenza di offuscamento di *hidder* è di 1 bit ogni 120 di codice.
- Per future ricerche, sarebbe possibile ricercare ulteriori possibili insiemi di istruzioni equivalenti, riuscendo così a migliorare le prestazioni in termini di frequenza di offuscamento.
- I test sulle tecniche implementate sono stati eseguiti su file eseguibili all'interno dei sistemi operativi: Ubuntu 18.04, Manjaro e Windows10.
- Sono stati presi in considerazione circa 100 file per ogni formato, con dimensione variabile dai 10KB ai 6MB.

Frequenza di offuscamento e dimensione file



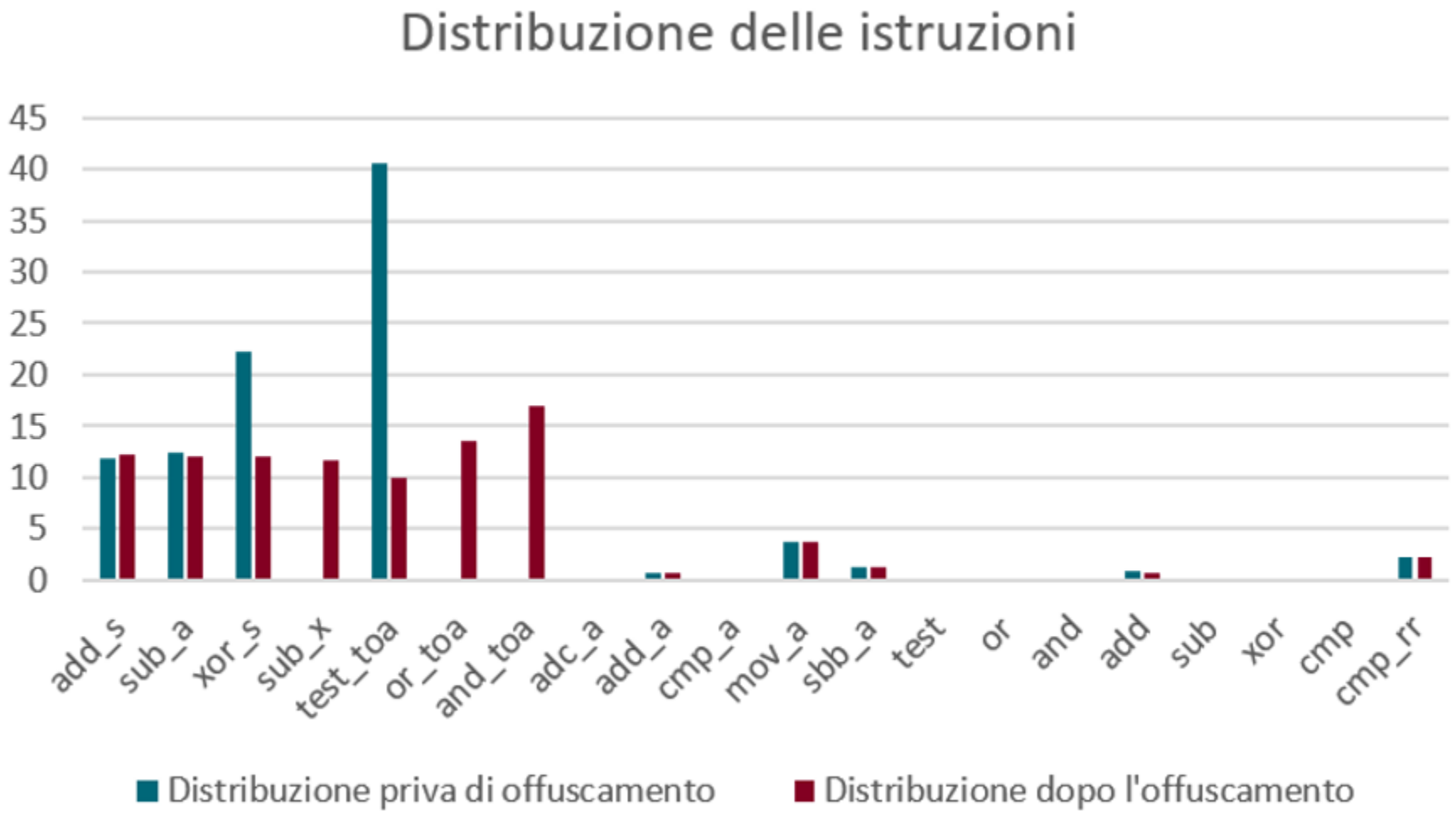
Frequenza di offuscamento



Steganalisi

- La Steganalisi è lo studio di tecniche per determinare se ad un messaggio sia stata applicata una steganografia.
- La steganografia ha pertanto come ulteriore requisito: rendere il più possibile difficile comprendere se il contenitore nasconda un messaggio, se non si è in possesso della chiave.
- Esiste un metodo per determinare l'utilizzo di *hidder?*

Steganalisi



Lavori futuri

- Fino ad ora abbiamo visto alcune tecniche di offuscamento, ma si può pensare ad altre possibili tecniche.
- Rimpiazzo di *dead-code* con istruzioni appositamente create per nascondere bit.
- Riordino dei blocchi funzionalmente indipendenti.

Conclusioni

- Seguendo le varie analisi abbiamo stabilito una frequenza di offuscamento di *hidder* di 1 bit ogni 120 di codice.
- In confronto, le tecniche di steganografia che utilizzano un'immagine come contenitore riescono ad ottenere un rateo medio di 1 bit ogni 17 di dimensione del file.
- Tuttavia, attraverso l'aggiunta ad *hidder* delle nuove tecniche si stima il raggiungimento di 1 bit offuscato ogni 19 di codice, rendendo la steganografia su file eseguibile una valida alternativa.
- Per il futuro vorrei implementare le restanti tecniche ottenendo il raggiungimento delle stime previste.

Ringraziamenti

Ringrazio tutti i presenti per l’attenzione,
Domande?

Bibliografia e Sitografia

- El-Khalil, R.: Hydan: Hiding Information in Program Binaries (2003).
- http://www.capstone-engine.org/, CAPSTONE, https://github.com/aquynh/capstone, ultimo accesso 19/08/2020
- https://www.keystone-engine.org/, KEYSTONE, https://github.com/keystone-engine/keystone, ultimo accesso 19/08/2020

Storia della steganografia

- La *Steganografia* è una tecnica di occultamento di un messaggio all'interno di un altro dato, non necessariamente digitale, innocuo.
- Alcune tracce della sua origine risalgono addirittura all'antica Grecia, nelle storie di Erodoto.
- Il primo utilizzo del conio Steganografia risale intorno al 1500 d.C. nei trattati di Tritemio, *Steganographia* e *Clavis Steganographiae*.
- All'interno di questi testi vi sono vari esempi di applicazioni di Steganografia, tra le quali troviamo le basi della Steganografia moderna.

Disassemblatore

- Data la natura binaria del codice macchina, sono stati ideati linguaggi che ne facilitano la lettura, chiamati *assembly.*
- L'*assembly* è un semplice equivalente mnemonico testuale delle varie istruzioni in codice macchina.
- La traduzione da codice macchina ad *assembly* viene effettuata da un disassemblatore.
- All'interno di *hidder* per facilitare le tecniche di offuscamento è stato utilizzato un disassemblatore esterno chiamato ***capstone***.
- *Capstone* riceve un vettore di byte (codice macchina), e ritorna un vettore ordinato, dove ogni elemento indica un'istruzione *assembly*.

Sostituzione Istr. equivalenti

- Abbiamo altri insiemi di istruzioni equivalenti, come il seguente, chiamato TOASXC

```
A8 FF       test    al, -1
0C 00       or      al, 0
24 FF       and     al, -1
04 00       add     al, 0
2C 00       sub     al, 0
34 00       xor     al, 0
3C 00       cmp     al, 0
```

Sostituzione Istr. equivalenti

- Quanti bit possiamo offuscare in un insieme da N istruzioni?

- Possiamo nascondere $\log_2 N$ bit se N è potenza di 2 e arrotondando per difetto quando non lo è.
  Ma possiamo migliorare recuperando qualche bit. Vediamo:

A8 FF	test	al, -1	→	000
0C 00	or	al, 0	→	001
24 FF	and	al, -1	→	010
04 00	add	al, 0	→	011
2C 00	sub	al, 0	→	100
34 00	xor	al, 0	→	101
3C 00	cmp	al, 0	→	11

Sostituzione istr. Aritmetiche consecutive

- Vediamo un altro esempio per capire perché prendiamo N-1 bit, per semplicità utilizziamo un registro ad 8 bit (range da 127 a -128).

- Fingiamo che il messaggio da nascondere sia 96 e che al registro al sia effettivamente sottratto 50 al termine delle due operazioni.

```
add     al, 96
mov     rcx, 1
push    rcx
sub     esp, 4
add     al, -50-96
jmp     LABEL_
```

```
add     al, -96
mov     rcx, 1
push    rcx
sub     esp, 4
add     al, -50+96
jmp     LABEL_
```

Rimpiazzo del *dead-code*

- Effettuando un rimpiazzo il risultato della ricerca deve essere accurata e sicura al 100% per non corrompere il funzionamento dell'eseguibile.

- Una semplice implementazione potrebbe essere:
  Seguire ricorsivamente il flusso di esecuzione del programma segnando ogni istruzione che viene eseguita.

- Al termine di questa ricerca le istruzioni mai visitate possono essere considerate *dead-code*.

Riordino dei blocchi indipendenti

- Questo metodo consiste nel determinare blocchi funzionalmente indipendenti e riordinarli per nascondere informazioni.
- Un esempio di blocchi indipendenti sono le varie funzioni del programma o le sezioni del file eseguibile.
- Un possibile metodo di implementazione consiste nell'ordinare i blocchi seguendo un ordine fisso.
- I possibili ordinamenti di N blocchi sono:
  $N * (N - 1) * (N - 2) * \ldots * 1$ ovvero $N!$
  Se attribuiamo ad ognuno di questi possibili ordinamenti un valore, possiamo nascondere $N_{bit} = \log_2 N!$ utilizzando quel determinato riordino.

Riordino dei blocchi indipendenti

- Non resta che determinare una funzione biunivoca che associ ad ogni sequenza di blocchi un valore. Vediamo un algoritmo:

Input: N, VAL // valore da nascondere
Output: N* // ovvero una copia di N ordinata secondo VAL
if $\log_2 VAL > \log_2 N!$ then:
    return “impossibile nascondere il valore” ;
Ordiniamo i blocchi di N secondo un ordinamento logico; // es grandezza
Creiamo una copia N* di N ed Inizializziamo i=0;
Scomponiamo VAL in fattoriali ottenendo:
$k * (sizeof(N) - 1)! + k^1 * (sizeof(N) - 2)! + \ \ldots + k^{sizeof(N)-2} * 1!$;
foreach K in ($k, k^1, \ldots, k^{sizeof(N)-2}$) then:
    N*[i] = N[K] ;
    riordino N senza N[K] ; //shift sinistro di uno a partire da N[K+1] in poi
    i = i + 1;
end
return N*;