

UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA



Corso di Laurea Magistrale in Informatica

Curriculum Sicurezza Informatica

Offuscamento del Codice tra Teoria e Pratica: la Piattaforma ObfuSee

Relatori:

**Ch.mo Prof.
Paolo D'Arco**

Candidata:

**Marta Coiro
Matr. 0522501611**

**Ch.ma Prof.ssa
Marialaura Noce**

ANNO ACCADEMICO 2024/2025

*A nonna Lina,
che continua a vivere nei miei ricordi,
come esempio di forza, saggezza e amore.*

*A mamma, papà e Carmen,
per il sostegno, l'incoraggiamento e la presenza costante.*

ABSTRACT

Nella continua battaglia tra sviluppatori e avversari, la protezione della proprietà intellettuale del software rappresenta una frontiera critica. Per decenni, l’offuscamento del codice è stato percepito come l’unica soluzione possibile, con l’obiettivo utopico di trasformare un programma in una “scatola nera virtuale” (Virtual Black Box), inaccessibile a qualsiasi tentativo di reverse engineering. Tuttavia, la ricerca teorica ha infranto questo sogno, dimostrando in modo rigoroso l’impossibilità di raggiungere tale livello di sicurezza assoluta.

Questa tesi si immerge in questa apparente contraddizione. Partendo da un’analisi rigorosa degli importantissimi limiti crittografici che ne decretano l’impossibilità teorica, il lavoro esplora come il campo si sia evoluto da una ricerca della perfezione a un’arte della resilienza pratica, infatti vengono esaminate le moderne strategie di offuscamento (lessicale, di controllo, sui dati e di anti-debugging). Il percorso analizza framework all’avanguardia come Khaos che semplifica approcci ibridi e multi-livello per massimizzare la confusione dell’analista. Il contributo originale di questa ricerca consiste nella progettazione e implementazione di una pipeline di offuscamento ibrida, che combina sinergicamente diverse tecniche in un flusso dimostrativo applicato a codice reale. I risultati evidenziano come un approccio stratificato possa trasformare un semplice programma in un complesso labirinto di codice, scoraggiando l’analista.

INDICE

1	Introduzione	6
1.1	Il Contesto: Protezione del Software e Reverse Engineering	7
1.2	Panoramica sull’offuscamento del codice	8
1.3	Come funziona l’offuscamento del codice?	9
1.4	Diversi tipi di offuscamento del codice	10
1.5	Come offuscare il codice: Best Practices	10
1.6	Principali applicazioni degli Offuscatori	11
1.7	Il limite teorico: La ”Virtual Black Box” e la sua Impossibilità .	12
1.8	Offuscamento come esercizio creativo e culturale	13
1.9	Obiettivi e Struttura della tesi	15
2	La Prova Fondamentale: Teoria e Limiti dell’Offuscamento Software	16
2.1	Il Principio Fondamentale: Il Programma come Messaggio	17
2.2	La Definizione Formale: Cos’è un Offuscatore VBB?	17
2.2.1	Costruzione del Controesempio	19
2.2.2	Come l’Avversario Svela l’Inganno	20
2.2.3	La Prova Definitiva: Il Fallimento del Simulatore	21
2.3	Approfondimento: Le Implicazioni Reali della Prova	21
2.4	Conclusioni del Capitolo	22

INDICE

3 Lo Stato dell'Arte dell'Offuscamento Pratico	24
3.1 L'Approccio Pragmatico: Il Lavoro di Collberg e Thomborson	25
3.2 Le Tecniche di Offuscamento: Un Arsenale per la Difesa	26
3.2.1 Offuscamento Lessicale	26
3.2.2 Offuscamento del Flusso di Controllo	26
3.2.3 Trasformazioni sui Dati	27
3.2.4 Tecniche di Anti-Debugging	27
3.2.5 Physical Unclonable Functions (PUF)	27
3.3 Lo Stato dell'Arte: Esempi di Framework Moderni	29
3.3.1 Khaos: Offuscamento Inter-Procedurale	29
3.3.2 ERIC: Un Approccio Ibrido con Protezione Hardware	30
3.3.3 White-Box Cryptography: Un Caso di Studio sui Limiti	30
3.3.4 Saxena e Wyseur: La Formalizzazione dei Limiti	31
3.4 Conclusioni del Capitolo	31
4 Progettazione e Sperimentazione di una Pipeline Ibrida	33
4.1 Obiettivi della fase sperimentale	33
4.2 Descrizione della piattaforma ObfuSee	34
4.3 Gli step di offuscamento	37
4.3.1 Step 2 – Offuscamento del flusso di controllo	38
4.3.2 Step 3 – Trasformazione dei dati	39
4.3.3 Step 4 – Tecniche Anti-Debugging	39
4.3.4 Step 5 - Codifica Base64	40
4.3.5 Step 6 - Cifratura AEAD	43
4.3.6 Perchè AEAD ?	43
4.4 Esempio pratico: Offuscamento di una calcolatrice	46
4.4.1 Introduzione all'esempio	46
4.4.2 Codice Originale	46
4.4.3 Processo di offuscamento passo per passo	47
4.5 Codice originale VS Codice Offuscato	51
4.6 Considerazioni finali	51

INDICE

5 Analisi dei risultati ottenuti	53
5.1 Comportamento in esecuzione — variante Base64 (Step 5)	53
5.2 Comportamento in esecuzione — variante AEAD (Step 6)	54
5.3 Analisi con Debugger	55
5.4 Reverse Engineering con <code>strings</code> - Codifica Base64 (Step 5) . .	56
5.5 Reverse Engineering con <code>strings</code> - Variante AEAD (Step 6)	59
5.6 Valutazione dell’Efficacia	60
6 Conclusioni	61
6.1 Lavori futuri	62
Ringraziamenti	64
References	70
Elenco delle figure	74
Elenco delle tabelle	75

CAPITOLO 1

INTRODUZIONE

Scrivere un programma è un atto di creazione che porta con sé una vulnerabilità intrinseca: per esistere, un’idea deve essere tradotta in codice, e quel codice, una volta eseguito, può essere svelato. L’offuscamento è da sempre la risposta a questa sfida: l’arte di velare la logica, di trasformare la chiarezza di un algoritmo in un labirinto dal quale è molto difficile trovare un’uscita.

A definire la direzione di questa ricerca è stata una verità tanto elegante nella sua dimostrazione quanto brutale nelle sue conclusioni: l’offuscamento perfetto è un’impossibilità teorica. Non un’ipotesi, ma un risultato provato. La “scatola nera” ideale, quindi, non è un obiettivo da raggiungere, ma un confine da cui partire.

Questa tesi, quindi, non insegue un miraggio. Accetta questa impossibilità come punto di partenza per esplorare il dominio del possibile. Il suo scopo è dimostrare che è proprio qui, accettato il limite, che inizia la vera ingegneria della protezione: non nel costruire fortezze inviolabili, ma nel progettare difese così complesse e costose da scoraggiare l’attaccante.

1.1 Il Contesto: Protezione del Software e Reverse Engineering

Il software moderno è molto più di un semplice strumento: è un concentrato di **proprietà intellettuale (IP)**. Al suo interno sono custoditi algoritmi proprietari, logiche di business che definiscono il successo di un'azienda, e talvolta anche segreti critici come chiavi crittografiche o protocolli di comunicazione. Questo “capitale digitale” rappresenta un enorme vantaggio competitivo, ma la sua natura immateriale lo rende intrinsecamente vulnerabile.

La principale minaccia a questa proprietà intellettuale è il **reverse engineering**¹. [1]

Con questo termine si intende il processo di analisi di un programma compilato per dedurne il funzionamento interno, la struttura e il design. Sebbene possa avere usi legittimi, come garantire l'interoperabilità tra sistemi, nel contesto della sicurezza informatica è l'arma principale dell'avversario. Le motivazioni dietro un'analisi ostile sono molteplici e spaziano da:

- Pirateria e cracking, per eludere meccanismi di licenza o rimuovere limitazioni.
- Spionaggio industriale, per scoprire e replicare algoritmi e segreti commerciali.
- Ricerca di vulnerabilità, per sviluppare exploit e condurre attacchi informatici.

¹Reverse engineering (in italiano “ingegneria inversa”, “ingegnerizzazione inversa”) è un anglicismo che indica quell’insieme di analisi delle funzioni, degli impieghi, della collocazione, dell’aspetto progettuale, geometrico e materiale di un manufatto o di un oggetto che è stato rinvenuto (ad esempio un reperto, un dispositivo, componente elettrico, un meccanismo, software).

Il fine può essere quello di produrre un altro oggetto che abbia un funzionamento analogo o migliore, o più adatto al contesto in cui ci si trova (fitting); un altro fine può essere quello di tentare di realizzare un secondo oggetto in grado di interfacciarsi con l’originale.

1. INTRODUZIONE

- Manomissione (tampering), per alterare il comportamento del software, ad esempio per barare in un gioco online o per aggirare controlli di sicurezza.

Questa minaccia espone un dilemma fondamentale e inevitabile nella distribuzione del software. A differenza di un segreto custodito su un server remoto, un programma destinato all’utente finale deve essere eseguito su una macchina che non è sotto il controllo dello sviluppatore. Per poter funzionare, le sue istruzioni devono essere accessibili al processore e alla memoria del dispositivo, e di conseguenza, sono esposte anche agli strumenti di analisi di chi lo possiede. La necessità di distribuire ed eseguire il codice entra quindi in diretto conflitto con l’esigenza di proteggerne i segreti. È in questo spazio, tra funzionalità e segretezza, che si colloca la sfida della protezione del software e la necessità di meccanismi di difesa come l’offuscamento.

1.2 Panoramica sull’offuscamento del codice

Nello scenario precedentemente descritto l’offuscamento emerge come una delle principali strategie di difesa. Ma cos’è l’offuscamento? In programmazione, l’offuscamento del codice [2] è l’atto di creare deliberatamente codice sorgente difficile da comprendere per un lettore umano. L’offuscamento può avere diverse motivazioni:

- volontà di proteggere la proprietà intellettuale rendendo difficile il reverse engineering e il riutilizzo non autorizzato del proprio codice da parte di terzi;
- complicare la modifica malevola del codice, come l’inserimento di malware o la realizzazione di crack;
- rendere più difficili azioni di violazione della licenza d’uso, come la creazione di keygen;
- a scopo ricreativo, per creare rompicapo destinati alla comunità dei programmati.

1. INTRODUZIONE

L'offuscamento può essere realizzato direttamente dal programmatore o introdotto attraverso un **offuscatore**, ovvero un programma specificatamente progettato per modificare il codice sorgente introducendo variazioni che ne complicano la lettura.

```
(A)
function setText(data) {
    document.getElementById("myDiv").innerHTML = data;
}

(B)
function ghs3x(n) {
    h = "\x69\u006En\u0065r\x48T\u004DL";
    a="s c v o v d h e , n !";x=a.split(" ");b="gztxleWentBsyf";
    r=b.replace("z",x[7]).replace("x","E").replace("s","");
    replace("f","I");
    ["rep1" + "ace"]("m","m")+"d";
    c="my"+String.fromCharCode(68)+x[10]+"v";
    s=x[5]+x[3]+x[1]+"um"+x[7]+x[9]+"t";d=this[s][r](c);if(+![])
    | d[h]=n; | else | d[h]=c; | }
```

Figura 1.1: Esempio codice offuscato

1.3 Come funziona l'offuscamento del codice?

L'offuscamento del codice altera la struttura senza cambiare l'output previsto. Il processo di trasformazione in genere include:

- Rinominare variabili e funzioni con nomi senza senso.
- Rimozione o inserimento di codice ridondante che aumenta la complessità senza influire sull'esecuzione.
- Crittografia di stringhe e costanti per impedire la facile estrazione di dati sensibili.
- Appiattimento dei flussi di controllo per mascherare sequenze logiche e percorsi di esecuzione.

Con l'implementazione di tali tecniche, l'offuscamento rende più difficile e impegnativo, per gli aggressori, analizzare, decompilare o manomettere la logica di un'applicazione.

1.4 Diversi tipi di offuscamento del codice

Esistono diversi tipi di metodi di offuscamento del codice. Ognuno soddisfa requisiti di sicurezza e complessità diversi:

1. Offuscamento lessicale

L'offuscamento lessicale modifica i nomi di variabili, funzioni e classi trasformandoli in identificatori privi di significato, rendendo difficile comprendere lo scopo del codice.

2. Offuscamento del flusso di controllo

Questo metodo altera la sequenza di esecuzione, rendendo più difficile tracciare il flusso logico del programma.

3. Offuscamento dei dati

L'offuscamento dei dati crittografa o trasforma i valori letterali e memorizzati, impedendo la facile estrazione di informazioni critiche.

4. Offuscamento del debug

Questo metodo rimuove o indirizza in modo errato i simboli di debug e i messaggi di errore per impedire agli strumenti di analisi di acquisire informazioni sul comportamento dell'applicazione.

5. Offuscamento del modello di istruzione

La modifica di schemi di codice comuni impedisce ai decompilatori e agli analizzatori basati sulle firme di riconoscere routine note.

1.5 Come offuscare il codice: Best Practices

Ecco una breve checklist per implementare correttamente l'offuscamento del codice:

- Scegliere il giusto livello di offuscamento in base alle esigenze di sicurezza e all'impatto sulle prestazioni.

- Utilizzare strumenti di offuscamento automatizzati.
- Applicare più tecniche di offuscamento per aumentare la complessità.
- Crittografare i dati sensibili e le chiavi API per impedirne l'estrazione.
- Aggiornare regolarmente i metodi di offuscamento per rimanere al passo con le tecniche di decompilazione in evoluzione.
- Testare le prestazioni dell'applicazione per garantire che l'offuscamento non introduca inefficienze.
- Combinare l'offuscamento con altre misure di sicurezza, come la protezione runtime e la firma del codice.

1.6 Principali applicazioni degli Offuscatori

L'offuscamento del codice non è soltanto una tecnica di protezione, ma uno strumento che può essere applicato in diversi contesti, dal software commerciale alle primitive crittografiche. In questo paragrafo vengono riportate alcune delle principali applicazioni individuate in letteratura, si fa riferimento al paper di Barak et al, che ne mostrano sia la rilevanza pratica che il potenziale teorico. Gli offuscatori trovano applicazione soprattutto nella **protezione del software**. L'idea di base è che, rendendo il codice difficile da comprendere, si possa ostacolare il reverse engineering. Per esempio, se un ricercatore scoprissse un algoritmo molto efficiente per la fattorizzazione, potrebbe volerlo sfruttare in altri programmi (come nel caso della crittoanalisi di RSA) senza però rivelare direttamente l'algoritmo stesso. L'offuscamento permetterebbe di distribuire il software mantenendo nascosta la logica interna più preziosa.

Un'altra applicazione riguarda il **watermarking del software**. In questo caso, il venditore può inserire nel programma delle tracce che identificano univocamente l'acquirente. Offuscando il codice, tali "firme

1. INTRODUZIONE

digitali” diventano difficili da individuare e rimuovere, garantendo quindi una forma di protezione legale e commerciale.

Gli offuscatori sono anche collegati a problemi più teorici, come la **crittografia omomorfica**. In linea teorica, offuscando opportuni algoritmi è possibile trasformare un normale crittosistema a chiave pubblica in uno che consente di eseguire operazioni direttamente sui dati cifrati.

Un altro ambito è la **sostituzione degli oracoli casuali**. Nei modelli crittografici spesso si assume l'esistenza di una funzione casuale perfetta, ma nella pratica si usano funzioni hash. Gli offuscatori, applicati a famiglie di funzioni pseudocasuali, potrebbero aiutare a costruire alternative più realistiche a questo modello.

Infine, l'offuscamento permette di trasformare **schemi di crittografia simmetrica** in **schemi a chiave pubblica**: pubblicando un programma offuscato che esegue la cifratura con la chiave segreta, chiunque può cifrare messaggi, mentre solo chi possiede la chiave può decifrarli.

1.7 Il limite teorico: La ”Virtual Black Box” e la sua Impossibilità

Per anni, la ricerca ha inseguito un ideale quasi mitico: l'offuscatore perfetto, un compilatore in grado di generare una **”Virtual Black Box” (VBB)**. Un programma, cioè, che non rivela alcuna informazione in più rispetto a quella ottenibile semplicemente eseguendolo. Se esistesse, un tale offuscatore renderebbe il reverse engineering praticamente inutile.

Tuttavia, questo ideale si è scontrato con la dura realtà della teoria crittografica. Un lavoro fondamentale di Barak et al [3]. ha dimostrato in modo rigoroso che l'offuscamento inteso come “Virtual Black Box” è, in termini generali, impossibile da realizzare. I ricercatori hanno costruito famiglie di funzioni “intrinsecamente non-offuscabili”, per le quali è possibile estrarre informazioni dal codice offuscato che sarebbero inaccessibili tramite la sola esecuzione. Questa scoperta ha creato una profonda spaccatura nel campo: *se*

la protezione perfetta è un’utopia, come possiamo difendere il codice nel mondo reale?.

1.8 Offuscamento come esercizio creativo e culturale

L’offuscamento del codice non è soltanto uno strumento di protezione contro il reverse engineering, ma è anche riconosciuto come forma espressiva e intellettuale nel mondo della programmazione. Scrivere (e decifrare) codice intenzionalmente complesso può essere vissuto come un vero e proprio rompicapo per programmati esperti.

Esistono competizioni internazionali che premiano gli esempi più creativi e “illeggibili” di codice, come:

- International Obfuscated C Code Contest (IOCCC).
- Obfuscated Perl Contest.
- International Obfuscated Ruby Code Contest.

In tali eventi, i partecipanti si sfidano nell’esprimere creatività, umorismo e abilità tecnica, spesso con risultati sorprendenti. Alcuni sviluppatori inseriscono blocchi di codice offuscato come firma, rendendo i propri programmi unici.

Di seguito un esempio reale, questo programma C, scritto da Ian Phillips, ha vinto l’International Obfuscated C Code Contest nel 1988. Il programma (insospettabilmente) stampa a video i 12 versi della canzone **“Twelve Days of Christmas”**. [4]

1. INTRODUZIONE

```
1  /*
2   LEAST LIKELY TO COMPILE SUCCESSFULLY:
3   Ian Phillipps, Cambridge Consultants Ltd., Cambridge, England
4 */
5
6 #include <stdio.h>
7 main(t,_,a)
8 char
9 *
10 a;
11 {
12     return!
13
14 0<t?
15 t<3?
16
17 main(-79,-13,a+
18 main(-87,1-_,
19 main(-86, 0, a+1 )
20
21 +a));
22
23 1,
24 t<_?
25 main(t+1, _, a )
26 :3,
27
28 main ( -94, -27+t, a )
29 &&t == 2 ?_
30 <13 ?
31
32 main ( 2, _+1, "%s %d %d\n" )
33
34 :9:16:
35 t<0?
36 t<-72?
37 main( _, t,
38 "@n'#+#'*{}w+/w#cdnr/+,{}r/*de}+,*{+,/w{%,/w#q#+n,/#[1,+/n{n+,/#n+,/#;\ \
39 #q#+n+,/#k#;*+,/'r :'d*'3,{w+K w'K:'+'e#';dq#`1 q#'+d'K#!/#k#; \
40 q#`r)eKK#}w'r]eKK{n1}'/#;#q#n'){})#w')(){}n1]'/#n';d}rw' i;# ){}n1]!/n{n#'; \
41 r{#w'r nc{n1}'/#{1,+'K {rw' iK{;[{n1]'/w#q#\ \
42 \
43 n'wk nw' iwk{KK{n1}!/w{%'l##w#` i; :{n1}'/*{q#`ld;r'}{nlwb!/*de}`c ;; \
44 {n1}'-{rw}'/+,}##'*#nc,'#nw]'/#kd'+e}+; \
45 '#rdq#w! nr'/' ') }+}{rl#'{n' '}# }'+}##(!!/")
46 :
47 t<-50?
48 _==*a ?
49 putchar(31[a]);
50
51 main(-65,_,
52 a+1)
53 :
54 main((*a == '/') + t, _, a + 1 )
55 :
56 0<t?
57
58 main ( 2, 2 , "%s")
59 :*a=='/' ||
60
61 main(0,
62
63 main(-61,*a, "lek;dc i@bK'(q)-[w]*%n+r3#l,{}:\\nuwlloca-0;m .vpbks,fxntdCeghiry")
64
65 ,a+1);}
```

Figura 1.2: Programma scritto da Ian Phillips

1.9 Obiettivi e Struttura della tesi

Questa tesi si propone di esplorare questa affascinante dicotomia tra l'impossibilità teorica e la necessità pratica della protezione del software. L'obiettivo è dimostrare come, nonostante i limiti fondamentali, l'offuscamento rimanga uno strumento potente, a patto di ridefinirne gli scopi: non la ricerca di una sicurezza assoluta, ma la costruzione di una resilienza pratica. Con questo termine si intende la capacità di rendere un attacco di reverse engineering talmente costoso, in termini di tempo e risorse, da diventare economicamente svantaggioso.

Per raggiungere tale obiettivo, la tesi è così strutturata:

- **Capitolo 2** approfondirà i fondamenti teorici, analizzando in dettaglio il paper di Barak et al. e la prova di impossibilità che definisce i confini del campo.
- **Capitolo 3** esplorerà lo stato dell'arte delle tecniche pratiche, partendo dal lavoro di Collberg e Thomborson e analizzando esempi di framework moderni come Khaos e ERIC.
- **Capitolo 4** descriverà la progettazione e l'implementazione di una pipeline di offuscamento ibrida, il contributo sperimentale di questo lavoro.
- **Capitolo 5** presenterà l'analisi dei risultati ottenuti, valutando l'efficacia della pipeline.
- **Capitolo 6** trarrà le conclusioni finali, riassumendo come, nell'impossibilità di creare fortezze inespugnabili, l'offuscamento moderno si configuri come un'arte per costruire labirinti di codice efficaci, trasformando la protezione in una strategia di deterrenza economica.

CAPITOLO 2

LA PROVA FONDAMENTALE: TEORIA E LIMITI DELL’OFFUSCAMENTO SOFTWARE

Dopo aver introdotto nel capitolo precedente la sfida della protezione del software, questo capitolo si concentra sul pilastro teorico che ne definisce i confini. L’analisi che segue è dedicata interamente al lavoro di Barak et al., “**On the (Im)possibility of Obfuscating Programs**” , che ha rigorosamente dimostrato l’impossibilità di un offuscamento perfetto.

L’obiettivo qui non è solo riassumere questo risultato, ma decostruirne la logica per comprendere a fondo le ragioni della sua ineluttabilità. Inizieremo con la formalizzazione del concetto di “Virtual Black Box”, l’ideale irraggiungibile. Proseguiremo smontando, passo dopo passo, l’ingegnoso controesempio che ne prova l’impossibilità. Infine, discuteremo perché questa conclusione, apparentemente negativa, sia in realtà il punto di partenza essenziale per ogni approccio pragmatico alla sicurezza del software discusso

2. LA PROVA FONDAMENTALE: TEORIA E LIMITI DELL'OFFUSCAMENTO SOFTWARE

nel resto di questa tesi.

2.1 Il Principio Fondamentale: Il Programma come Messaggio

Il cuore della dimostrazione di impossibilità poggia su una distinzione cruciale, quasi filosofica, tra un programma e la sua funzione:

- Un **oracolo** è una “scatola nera” ideale. Possiamo dargli un input e osservare l’output, ma non possiamo ispezionarne i meccanismi interni. È un puro comportamento.
- Un **programma**, invece, è la descrizione di quei meccanismi. È un “libro di ricette”. Non solo possiamo usarlo per calcolare la funzione, ma possiamo anche leggere, analizzare e, soprattutto, trattare il suo codice sorgente o compilato come un semplice **dato**.

È proprio questa natura duale del programma — essere sia un esecutore di istruzioni sia un dato manipolabile — a creare una vulnerabilità che nessun offuscamento può eliminare del tutto. La prova sfrutta questa debolezza in modo magistrale.

2.2 La Definizione Formale: Cos’è un Offuscatore VBB?

Per poter dimostrare rigorosamente l’impossibilità di un concetto, è prima necessario definirlo in termini matematici precisi. Il lavoro di Barak et al. formalizza la nozione di offuscatore in questo modo:

Informalmente, un offuscatore O è un “compilatore” (efficiente e probabilistico) che riceve come input un programma (o circuito) P e produce un nuovo programma $O(P)$ che ha le stesse funzionalità di P ma è “non intelligibile” .

2. LA PROVA FONDAMENTALE: TEORIA E LIMITI DELL'OFFUSCAMENTO SOFTWARE

In particolare il lavoro stabilisce tre condizioni che un qualsiasi offuscatore O dovrebbe soddisfare.

1. **Funzionalità (Functionality):** Per qualsiasi circuito C , il programma offuscato $O(C)$ deve calcolare la stessa identica funzione di C .
2. **Rallentamento Polinomiale (Polynomial Slowdown):** La dimensione del programma offuscato $O(C)$ deve essere al massimo polinomialmente più grande della dimensione di C . Questo assicura che l'offuscamento sia una trasformazione efficiente e utilizzabile in pratica.
3. **Proprietà “Virtual Black Box” (VBB):** Questa è la condizione di sicurezza fondamentale. Richiede che per ogni avversario efficiente A (un PPT, ovvero una Macchina di Turing Probabilistica a tempo Polinomiale), esista un simulatore efficiente S (anch'esso un PPT) tale che, per ogni circuito C , le probabilità che A distingua l'output di $O(C)$ da quello prodotto da S con accesso oracolare a C differiscano al più di una quantità trascurabile $\alpha(|C|)$. Formalmente:

$$\Pr[A(O(C)) = 1] - \Pr[S^C(1^{|C|}) = 1] \leq \alpha(|C|),$$

dove $\alpha(|C|)$ è una funzione trascurabile, cioè decresce più velocemente dell'inverso di qualsiasi polinomio.

Questa definizione cattura l'intuizione che l'offuscato $O(C)$ non deve rivelare nulla di più rispetto a ciò che si potrebbe ottenere avendo solo accesso oracolare a C .

Nella formula appena definita ogni termine ha un significato preciso:

- $\Pr[\cdot]$: denota la probabilità, calcolata rispetto alla casualità interna degli algoritmi probabilistici (sia A , sia S , sia l'eventuale randomizzazione dell'offuscatore O).
- A : rappresenta l'avversario, cioè un algoritmo probabilistico in tempo polinomiale (PPT) che riceve in input il circuito offuscato $O(C)$ e tenta di estrarre informazioni aggiuntive.

2. LA PROVA FONDAMENTALE: TEORIA E LIMITI DELL'OFFUSCAMENTO SOFTWARE

- $O(C)$: è il programma (o circuito) offuscato prodotto dall'offuscatore O a partire dal circuito originale C .
- S : è il simulatore, anch'esso un PPT, che non vede direttamente $O(C)$ ma ha accesso oracolare al circuito originale C . Ciò significa che S può interrogare C sugli input e ricevere i corrispondenti output, ma non ha alcuna conoscenza interna del circuito.
- $S^C(1^{|C|})$: indica l'esecuzione del simulatore S con accesso oracolare a C e con parametro di sicurezza $1^{|C|}$. Quest'ultimo serve a fissare la complessità in funzione della dimensione $|C|$ del circuito.
- $\alpha(|C|)$: è una funzione trascurabile, ovvero una funzione che tende a zero più velocemente dell'inverso di qualsiasi polinomio in $|C|$. In altre parole, il vantaggio dell'avversario nel distinguere l'offuscato dal simulatore diventa insignificante man mano che cresce la dimensione del circuito.

In sintesi, la proprietà VBB stabilisce che tutto ciò che un avversario A può imparare osservando direttamente l'offuscato $O(C)$, potrebbe essere appreso (a parte un errore trascurabile) da un simulatore S che interagisce con il circuito solo come una “**scatola nera**”.

2.2.1 Costruzione del Controesempio

Per dimostrare l'impossibilità dell'offuscamento, Barak et al. introducono due programmi appositamente costruiti:

- $C_{\alpha,\beta}$: una macchina di Turing (o circuito) parametrizzata da due stringhe $\alpha, \beta \in \{0,1\}^k$. È definita come segue:

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{se } x = \alpha \\ 0^k & \text{altrimenti} \end{cases}$$

In altre parole, il programma restituisce il valore β soltanto quando riceve in input la chiave α ; per ogni altro input restituisce la stringa nulla.

2. LA PROVA FONDAMENTALE: TEORIA E LIMITI DELL'OFFUSCAMENTO SOFTWARE

- $D_{\alpha,\beta}$: un secondo programma che prende in input il codice di un programma P e verifica se P calcola la stessa funzione di $C_{\alpha,\beta}$. La sua logica è:

$$D_{\alpha,\beta}(P) = \begin{cases} 1 & \text{se } P(\alpha) = \beta \\ 0 & \text{altrimenti} \end{cases}$$

Questa costruzione mette in evidenza il punto cruciale dell'impossibilità: l'analisi di un programma offuscato permette comunque di distinguere la coppia $(C_{\alpha,\beta}, D_{\alpha,\beta})$ da programmi che non hanno quella relazione, mentre ciò non sarebbe possibile con il solo accesso *black-box*.

2.2.2 Come l'Avversario Svela l'Inganno

Immaginiamo ora un avversario il cui obiettivo è distinguere il programma speciale $C_{\alpha,\beta}$ da un programma banale che restituisce sempre 0^k . Se l'offuscamento rispettasse la definizione di *Virtual Black Box* (VBB), questo non dovrebbe essere possibile. Tuttavia, il paper dimostra come l'avversario riesce a vincere:

- L'avversario riceve in ingresso le versioni offuscate dei due programmi: $O(C_{\alpha,\beta})$ e $O(D_{\alpha,\beta})$. Sebbene il codice sia reso “inintelligibile”, la funzionalità deve essere preservata.
- L'avversario compie l'azione cruciale: esegue il programma $O(D_{\alpha,\beta})$ fornendogli come input il codice del programma $O(C_{\alpha,\beta})$.
- Poiché l'offuscamento mantiene la funzionalità, $O(D_{\alpha,\beta})$ si comporta esattamente come $D_{\alpha,\beta}$. Di conseguenza, verifica che $O(C_{\alpha,\beta})(\alpha) = \beta$ e restituisce 1.

In questo modo, l'avversario apprende una proprietà nascosta del programma, violando l'essenza stessa dell'offuscamento VBB.

2. LA PROVA FONDAMENTALE: TEORIA E LIMITI DELL'OFFUSCAMENTO SOFTWARE

2.2.3 La Prova Definitiva: Il Fallimento del Simulatore

La definizione di VBB crolla quando confrontiamo il successo dell'avversario con ciò che un “simulatore” con solo accesso oracolare può fare. La regola della VBB, infatti, impone che per ogni avversario **A** esista un simulatore **S** che, con solo accesso oracolare alla funzione, possa replicare i risultati di **A**.

- Il simulatore ha accesso a due “scatole nere”, una per C e una per D .
- Può dare input alla scatola C , ma riceverà sempre 0, poiché la probabilità di indovinare α è trascurabile.
- Può dare il codice di altri programmi alla scatola D .
- Tuttavia, il simulatore **non può dare la scatola C in pasto alla scatola D**. Non può usare un oracolo come dato per un altro oracolo.

Di conseguenza, il simulatore non può replicare l'esperimento dell'avversario e non può distinguere $C(\alpha, \beta)$ da un programma nullo con una probabilità maggiore del 50%. Poiché l'avversario (con il codice) può fare qualcosa che il simulatore (con l'oracolo) non può, l'offuscatore ha fallito nel suo compito. L'offuscamento VBB è impossibile.

2.3 Approfondimento: Le Implicazioni Reali della Prova

Al di là della sua eleganza tecnica, la dimostrazione di impossibilità ha implicazioni profonde che ridefiniscono il modo in cui dobbiamo pensare alla sicurezza del software.

2. LA PROVA FONDAMENTALE: TEORIA E LIMITI DELL'OFFUSCAMENTO SOFTWARE

- **La Frattura è Semantica, non Tecnica:** Il risultato di Barak et al. non dice semplicemente che le attuali tecniche di offuscamento non sono abbastanza buone. Dice qualcosa di molto più forte: che il problema è **semantico**, cioè legato al significato stesso di “avere un programma”. Avere il codice, per quanto confuso, conferisce un potere intrinsecamente superiore rispetto al semplice osservarne il comportamento.
- **Spostamento degli Obiettivi (*Shifting the Goalposts*):** La conseguenza più importante è stata costringere la comunità scientifica a spostare gli obiettivi. L’offuscamento non mira più all’ideale irraggiungibile della VBB. Gli obiettivi moderni sono più pragmatici: aumentare il costo per l’analista umano, sconfiggere tool di analisi specifici e proteggere contro classi di attacchi ben definite.
- **Giustificazione per la Difesa in Profondità (*Defense-in-Depth*):** L’impossibilità di un offuscamento perfetto fornisce la giustificazione teorica per un approccio di “**difesa in profondità**”. Poiché nessuna singola tecnica può garantire una protezione totale, è necessario combinare più livelli di sicurezza.

2.4 Conclusioni del Capitolo

L’analisi del lavoro seminale di Barak, Goldreich, Impagliazzo et al. conduce a una conclusione tanto netta quanto fondamentale: l’offuscamento, inteso come la creazione di una **Virtual Black Box** impenetrabile, è teoricamente impossibile.

Abbiamo dimostrato come questa impossibilità non derivi da un difetto tecnico, ma da una **profonda asimmetria** tra la natura di un programma come dato manipolabile e quella di un oracolo

2. LA PROVA FONDAMENTALE: TEORIA E LIMITI DELL'OFFUSCAMENTO SOFTWARE

come puro comportamento. La costruzione di funzioni intrinsecamente non-offuscabili ha svelato questa debolezza, dimostrando che esisterà sempre almeno una proprietà di un programma che può essere scoperta dal suo codice, per quanto offuscato, ma non dalla sua sola esecuzione.

Questo risultato non diminuisce l'importanza dell'offuscamento; al contrario, ne chiarisce il ruolo e i limiti. Abbandonato il sogno della sicurezza perfetta, la ricerca e la pratica si sono orientate verso obiettivi più realistici: non costruire fortezze inviolabili, ma **labirinti complessi** capaci di aumentare i costi e i tempi per l'avversario.

Questo capitolo ha quindi stabilito il “**perché**” teorico che motiva l’approccio pratico. Con la consapevolezza di questi limiti fondamentali, siamo ora pronti a esplorare, nel capitolo successivo, il “**come**”: le tecniche, le strategie e gli strumenti che costituiscono lo stato dell’arte dell’offuscamento nel mondo reale.

CAPITOLO 3

LO STATO DELL'ARTE DELL'OFFUSCAMENTO PRATICO

Dopo aver stabilito nel capitolo precedente i limiti teorici invalicabili dell'offuscamento, questo capitolo esplora la risposta pragmatica del mondo della ricerca e dell'industria. *Se la fortezza perfetta è un'illusione, come si costruiscono difese efficaci?*. La risposta risiede in un cambio di paradigma: dall'aspirazione alla sicurezza assoluta alla ricerca della **resilienza pratica**.

Questo capitolo analizza lo stato dell'arte delle tecniche di offuscamento che mirano non a rendere un attacco impossibile, ma a renderlo economicamente e tecnicamente insostenibile. Partendo dal lavoro seminale di Collberg e Thomborson, che ha gettato le basi per un approccio sistematico, esamineremo l'arsenale di trasformazioni a disposizione degli sviluppatori e analizzeremo alcuni dei più moderni

3. LO STATO DELL'ARTE DELL'OFFUSCAMENTO PRATICO

e ingegnosi framework di protezione, che esemplificano l’evoluzione di questo affascinante campo.

3.1 L’Approccio Pragmatico: Il Lavoro di Collberg e Thomborson

Il punto di partenza per comprendere l’offuscamento pratico è il paper [5] “**Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection**” di Christian Collberg e Clark Thomborson. Questo lavoro ha avuto il merito di sistematizzare il campo della protezione software, identificando tre aree fondamentali, ciascuna mirata a contrastare una specifica minaccia:

- **Offuscamento**, come difesa contro il reverse engineering.
- **Watermarking**, per contrastare la pirateria software.
- **Tamper-proofing**, per proteggere dalla manomissione.

Questa tesi, come specificato, si concentra esclusivamente sull’offuscamento. L’articolo di Collberg e Thomborson è cruciale perché sposta l’obiettivo dalla perfezione teorica a criteri pratici, sostenendo che l’offuscamento può soltanto alzare il costo e la complessità dell’attacco, non prevenirlo totalmente. Hanno inoltre introdotto una classificazione delle trasformazioni offuscanti (lessicali, di controllo, sui dati) che è diventata uno standard de-facto¹ e che useremo per strutturare la nostra analisi.

¹In italiano, “standard de-facto” si riferisce a uno standard che, pur non essendo ufficialmente riconosciuto o normato da un ente di standardizzazione, è ampiamente diffuso e accettato nella pratica, spesso a causa della sua adozione diffusa nel mercato. In sostanza, diventa uno standard “di fatto” per la sua popolarità e uso diffuso, non per un’approvazione formale.

3.2 Le Tecniche di Offuscamento: Un Arsenale per la Difesa

L’offuscamento moderno si basa su un’ampia gamma di trasformazioni, spesso applicate in modo combinato per creare una difesa stratificata. Di seguito analizziamo le principali categorie.

3.2.1 Offuscamento Lessicale

Questa è la forma più semplice di offuscamento. Agisce sui nomi di variabili, funzioni e classi, sostituendoli con etichette prive di significato (es. “contaUtenti” diventa “x4”). L’obiettivo primario è confondere un analista umano che legge il codice decompilato. Sebbene sia efficace per questo scopo, è una difesa debole contro l’analisi automatica, in quanto la logica del programma rimane invariata. Tool come ProGuard (per Java) o Obfuscator-LLVM implementano ampiamente queste trasformazioni.

3.2.2 Offuscamento del Flusso di Controllo

Queste tecniche mirano a rendere contorto e difficile da seguire il percorso di esecuzione del programma. Tra le più note vi sono:

- **Predicati Opachi (Opaque Predicates):** Si tratta dell’inserzione di condizioni booleane il cui esito è noto a priori al programmatore, ma che appaiono complesse e difficili da risolvere per un analista. Ad esempio, una condizione come “`if (x*x + 1 > 0)`” è sempre vera per numeri reali, ma costringe l’analista a fermarsi per comprenderla.
- **Appiattimento del Flusso di Controllo (Control Flow Flattening):** Questa tecnica distrugge la struttura gerarchica naturale del codice (if-then, loop) e la riorganizza all’interno di un grande costrutto “switch” o “select case”. Un dispatcher centrale

3. LO STATO DELL'ARTE DELL'OFFUSCAMENTO PRATICO

decide quale blocco di codice eseguire successivamente, rendendo il flusso logico un “piatto di spaghetti” quasi inestricabile.

3.2.3 Trasformazioni sui Dati

L’obiettivo di queste trasformazioni è nascondere il significato e l’uso dei dati all’interno del programma. Invece di memorizzare un valore nel suo formato naturale, questo viene codificato o suddiviso. Ad esempio, una variabile booleana “attivo” potrebbe essere sostituita da due interi “p” e “q” la cui relazione determina lo stato logico. Allo stesso modo, un numero come “10” potrebbe essere memorizzato come il risultato di un’operazione, ad esempio “3+7”.

3.2.4 Tecniche di Anti-Debugging

Queste tecniche hanno lo scopo specifico di ostacolare l’analisi dinamica, ovvero l’ispezione del programma mentre è in esecuzione. Il codice viene arricchito con controlli che rilevano la presenza di un debugger attivo. Se un debugger viene rilevato, il programma può terminare, fornire risultati errati o entrare in cicli infiniti per bloccare l’analisi. Altre strategie includono l’introduzione di pause temporali false o la generazione di errori finti per confondere l’analista.

3.2.5 Physical Unclonable Functions (PUF)

Le **Physical Unclonable Functions (PUF)** [6] sono meccanismi hardware che sfruttano le imperfezioni fisiche inevitabili durante il processo di fabbricazione dei circuiti integrati per generare identificativi unici e non replicabili.

A differenza delle chiavi crittografiche tradizionali, le PUF non sono memorizzate in memoria, ma generate al volo interrogando il

3. LO STATO DELL'ARTE DELL'OFFUSCAMENTO PRATICO

comportamento fisico del dispositivo. Questo rende estremamente difficile per un attaccante copiare o estrarre l'identificativo, anche con accesso diretto all'hardware.

Proprietà fondamentali di una PUF:

- **Unicità:** ogni dispositivo genera un output differente in risposta allo stesso stimolo.
- **Ripetibilità:** lo stesso dispositivo genera sempre la stessa risposta a uno stesso stimolo.
- **Non clonabilità:** è impossibile riprodurre la funzione esatta in un altro dispositivo, anche con conoscenza del progetto.
- **Difficoltà di simulazione:** una PUF ben progettata non può essere emulata software o replicata con precisione.

Le PUF vengono utilizzate in ambito di sicurezza e trovano applicazione in:

- Autenticazione dell'hardware (es. chip anticontraffazione).
- Generazione sicura di chiavi crittografiche.
- Protezione del firmware.
- Blocco geografico o vincolo all'ambiente fisico.

Nel contesto di questa tesi, realizzare una vera PUF non è disponibile. Tuttavia, è possibile simulare il comportamento di una PUF inserendo un controllo software su un identificativo fisso o calcolato (es. seriale del dispositivo, hash dell'ambiente, fingerprint univoco). Questo permette di replicare l'effetto di vincolare l'esecuzione del programma all'identità della macchina ospite, anche se in modo meno sicuro e reale.

Un'alternativa più solida, ampiamente utilizzata nei sistemi reali, è l'impiego del Trusted Platform Module (TPM). [7] [8] Il TPM è un chip di sicurezza standardizzato che contiene chiavi crittografiche

3. LO STATO DELL'ARTE DELL'OFFUSCAMENTO PRATICO

uniche generate e custodite a livello hardware, non esportabili. Attraverso un protocollo di attestazione basato su **challenge-response**, un'applicazione può inviare una sfida casuale al TPM e ricevere una risposta firmata che attesta sia l'identità del dispositivo sia lo stato della piattaforma (PCR, Platform Configuration Registers). Questo approccio, pur non essendo una PUF in senso stretto, fornisce proprietà analoghe di unicità, ripetibilità e non clonabilità, ed è già oggi largamente integrato in laptop e server moderni. Per ragioni di semplicità implementativa, nel lavoro presentato non è stato sviluppato lo step di attestazione TPM, ma se ne riconosce la validità come tecnica alternativa e più robusta per legare l'esecuzione del software all'identità hardware del dispositivo. In particolare, non è stato implementato neanche lo step di offuscamento basato su PUF, poiché la sola componente web non risulta sufficiente e l'integrazione con strumenti esterni in tempo reale rischierebbe di distogliere l'attenzione dagli obiettivi principali della piattaforma. È stato tuttavia previsto uno step alternativo legato alla cifratura che verrà ampiamente analizzato e discusso nel capitolo successivo.

3.3 Lo Stato dell'Arte: Esempi di Framework Moderni

Le tecniche descritte in precedenza vengono integrate in framework complessi per massimizzarne l'efficacia. Di seguito analizziamo alcuni esempi rilevanti menzionati nella letteratura recente.

3.3.1 Khaos: Offuscamento Inter-Procedurale

Khaos [9] è un framework che si concentra sull'offuscamento della struttura inter-procedurale di un binario. Invece di modificare solo l'interno di una funzione, Khaos utilizza due primitive potenti:

3. LO STATO DELL'ARTE DELL'OFFUSCAMENTO PRATICO

- **Fission:** Suddivisione di una singola funzione in più parti più piccole.
- **Fusion:** Fusione di più funzioni in una sola, più grande e complessa.

L'impatto di queste trasformazioni è notevole sui tool di *binary diffing* (che confrontano due versioni di un file binario), la cui precisione, secondo gli autori, crolla al di sotto del 19% con un overhead prestazionale inferiore al 7%. Khaos dimostra l'efficacia di applicare i concetti di offuscamento del flusso su una scala più ampia.

3.3.2 ERIC: Un Approccio Ibrido con Protezione Hardware

ERIC[10] “**An Efficient and Practical Software Obfuscation Framework**” rappresenta un’evoluzione che combina offuscamento software e protezione hardware. L’idea centrale è cifrare il binario del programma usando chiavi generate da una **Physical Unclonable Function (PUF)**. Il programma viene quindi decifrato ed eseguito solo su dispositivi dotati di quella specifica “firma hardware”, traducendo i concetti di *tamper-proofing* in una soluzione concreta.

3.3.3 White-Box Cryptography: Un Caso di Studio sui Limiti

Le implementazioni di crittografia White-Box, come quella per AES proposta da Chow et al.[11], rappresentano un tentativo di proteggere una chiave crittografica anche in un ambiente completamente **“trasparente”** (la *white-box*), dove l’attaccante ha pieno accesso al codice in esecuzione. L’idea è di **“fondere”** la chiave con la logica dell’algoritmo attraverso una serie di tabelle precalcolate. Tuttavia, questo approccio ha mostrato i suoi limiti nel tempo. Attacchi statistici

3. LO STATO DELL'ARTE DELL'OFFUSCAMENTO PRATICO

e differenziali si sono dimostrati efficaci nel recuperare la chiave da molte implementazioni di White-Box AES. Questo serve come un importante caso di studio: dimostra che anche le tecniche di offuscamento più sofisticate possono fallire e sottolinea la necessità di combinare più strategie di difesa.

3.3.4 Saxena e Wyseur: La Formalizzazione dei Limiti

Il lavoro di A. Saxena e B. Wyseur del 2008 [12], “**On White-box Cryptography and Obfuscation**”, fornisce un’importante prospettiva teorica che completa l’analisi dei casi pratici. Il loro contributo principale è la formalizzazione dei concetti di **White-Box Property (WBP)** e **Unbreakable White-Box Property (UWBP)**. Attraverso questa formalizzazione, gli autori dimostrano l’impossibilità di ottenere un offuscamento perfetto in generale, rafforzando da un punto di vista matematico i limiti che erano già stati anticipati da Collberg. Questo lavoro è significativo perché giustifica teoricamente la necessità di adottare soluzioni pratiche basate su compromessi: se la protezione perfetta è irraggiungibile, allora l’obiettivo deve necessariamente spostarsi verso l’innalzamento del costo dell’attacco.

3.4 Conclusioni del Capitolo

Questo capitolo ha tracciato il percorso dell’offuscamento dall’astrazione teorica alla sua applicazione pratica. Partendo dal lavoro pionieristico di Collberg e Thomborson, abbiamo visto come l’obiettivo del campo si sia spostato dalla ricerca di una sicurezza perfetta e irraggiungibile verso un approccio pragmatico e stratificato. L’arsenale a disposizione degli sviluppatori è vasto e comprende trasformazioni lessicali, di controllo, sui dati e di anti-debugging.

3. LO STATO DELL'ARTE DELL'OFFUSCAMENTO PRATICO

Framework moderni come Khaos ed ERIC dimostrano che la ricerca è attiva e si muove verso soluzioni sempre più complesse, che combinano diverse tecniche (Khaos) o integrano anche la dimensione hardware (ERIC). Allo stesso tempo, i fallimenti di approcci come la White-Box Cryptography ci ricordano che nessuna tecnica è infallibile.

La conclusione è che l'offuscamento efficace oggi è un'attività di ingegneria della complessità. La strategia vincente non risiede in una singola trasformazione, ma nella loro combinazione intelligente per costruire un programma che, pur non essendo teoricamente inviolabile, sia praticamente troppo costoso e difficile da analizzare. Questa consapevolezza è il trampolino di lancio per il capitolo successivo, in cui progetteremo e implementeremo una nostra pipeline di offuscamento ibrida.

Tabella 3.1: Tabella riassuntiva: Confronto tra tecniche di offuscamento

Tecnica	Resilienza	Furtività	Overhead	Tool Disponibile
Khaos	Alta	Media	Basso	LLVM pass
ERIC + PUF	Molto alta	Alta	Medio	GitHub repo
White-box AES (2004)	Bassa	Alta	Alto	No
Saxena C Wyseur (2008)	Teoricamente alta	Alta	N/D	No

CAPITOLO 4

PROGETTAZIONE E Sperimentazione di una PIPELINE IBRIDA

4.1 Obiettivi della fase sperimentale

L’obiettivo principale della fase sperimentale è dimostrare, in maniera pratica, l’efficacia di diverse tecniche di offuscamento del codice sorgente per andare ad aumentare la complessità cognitiva e la resistenza a tecniche di reverse engineering. A tal fine, è stata progettata e sviluppata una piattaforma interattiva, **ObfuSee**, che consente di applicare in modo incrementale sei diversi step di offuscamento su codice scritto nei linguaggi C, Python, Java, JavaScript e Rust.

4. PROGETTAZIONE E SPERIMENTAZIONE DI UNA PIPELINE IBRIDA

4.2 Descrizione della piattaforma ObfuSee

ObfuSee si propone di diventare una piattaforma interattiva che consente di applicare in maniera incrementale sei diversi tipi di offuscamento su codici scritti in differenti linguaggi (C, Python, Java, JavaScript e Rust). Il nome ObfuSee rappresenta l'unione dei due aspetti fondamentali della piattaforma: da un lato l'offuscamento del codice (Obfu), e dall'altro la possibilità di vedere ed esplorare in modo interattivo i cambiamenti applicati (See). L'obiettivo è duplice: proteggere e confondere – ma anche educare e comprendere. La piattaforma mostra passo dopo passo cosa avviene, permettendo agli utenti di imparare non solo come offuscare il codice, ma anche come analizzarlo e decostruirlo. Si tratta di una web application sviluppata con Python(Flask) per il backend e HTML/CSS/Javascript per il frontend. L'interfaccia utente si presenta in maniera molto semplice ed intuitiva, suddivisa in tre blocchi consente di:

- Inserire codice sorgente originale attraverso un box presente sulla sinistra.
- Selezionare il linguaggio di programmazione attraverso un menù a tendina presente in alto a sinistra.
- Applicare uno dei sei step di offuscamento incrementale attraverso un menù a tendina posizionato accanto alla scelta del linguaggio.
- Visualizzare in tempo reale il codice offuscato il quale verrà mostrato all'interno del box centrale. In particolare tutte le modifiche vengono evidenziate con colore giallo così da permettere una visione immediata dei cambiamenti.
- Consultare una cronologia dettagliata delle modifiche apportate al codice originale , questo è visibile nel box a destra.
- Applicare tutti gli step di offuscamento contemporaneamente e ottenere il codice offuscato completo, cliccando sul bottone “Applica”

4. PROGETTAZIONE E Sperimentazione di una Pipeline ibrida

tutti gli step”. In particolare al click su “Applica tutti gli step” la piattaforma richiede di scegliere una sola modalità di protezione delle stringhe (Base64 oppure AEAD), poiché l’uso simultaneo è ridondante; la motivazione è approfondita nel paragrafo 4.3.6.4.

In questo modo al termine della fase di offuscamento è possibile confrontare visivamente codice originale e codice offuscato, mettendo in evidenza le principali modifiche e osservando quanto è difficile comprendere il codice offuscato rispetto a quello originale. Di seguito verranno mostrati degli screenshot della piattaforma per chiarire quanto appena detto.

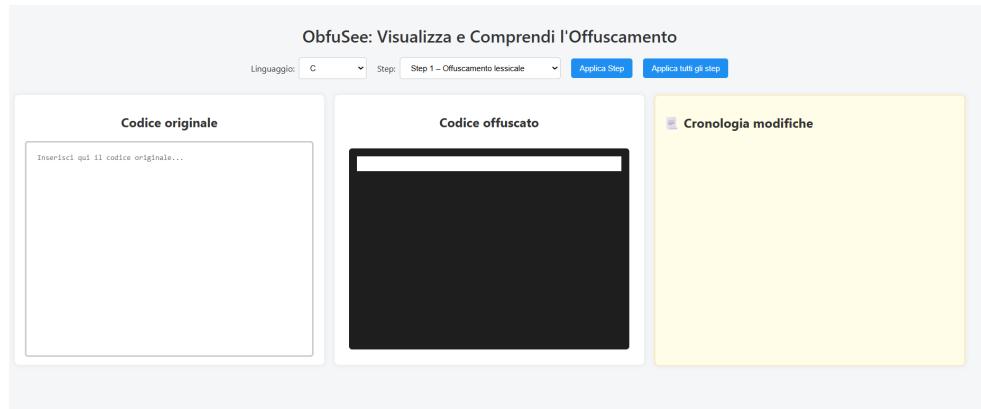


Figura 4.1: Pagina iniziale

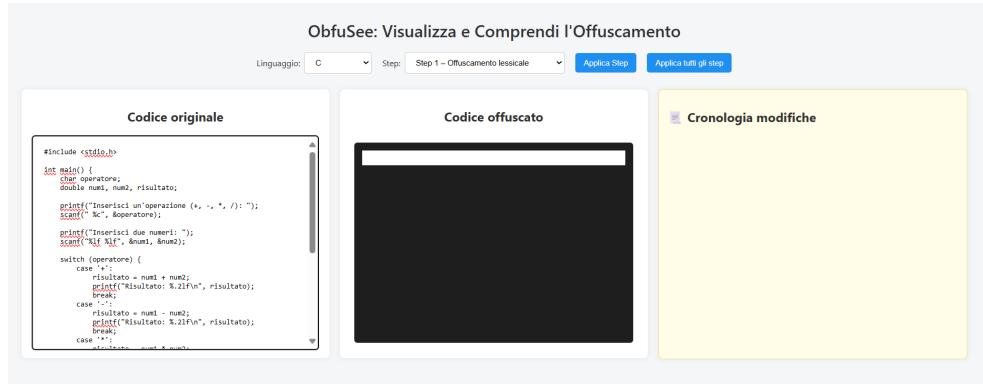


Figura 4.2: Inserimento codice

4. PROGETTAZIONE E Sperimentazione di una Pipeline Ibrida

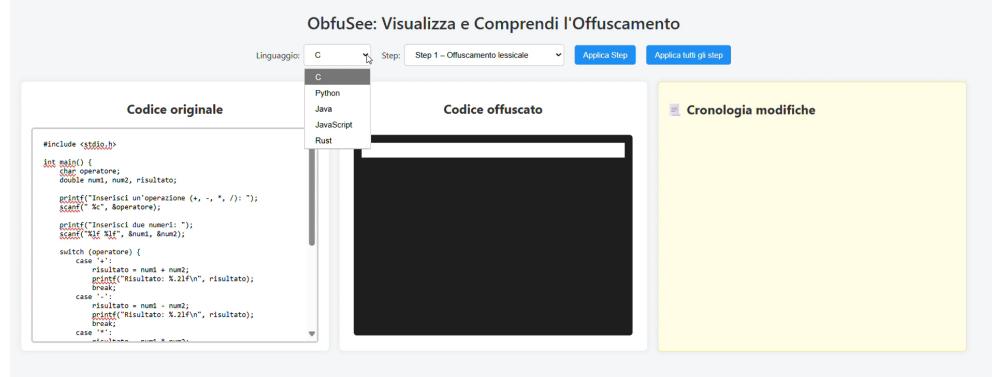


Figura 4.3: Scelta linguaggio

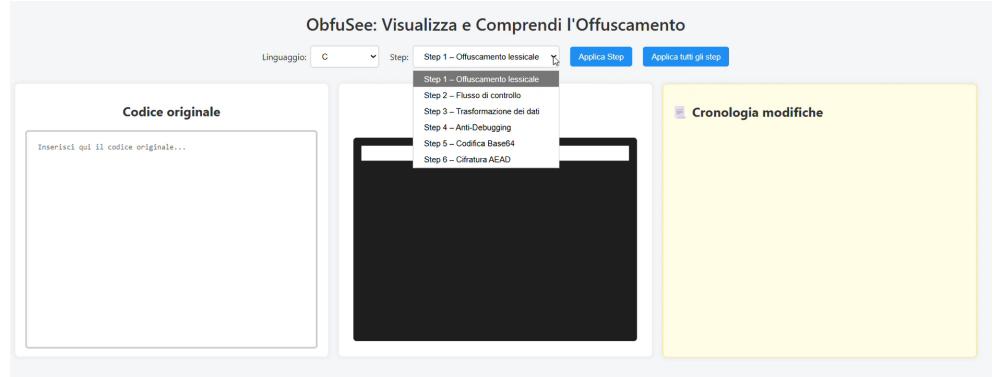


Figura 4.4: Scelta step



Figura 4.5: Applicazione Step 1

4. PROGETTAZIONE E Sperimentazione di una Pipeline ibrida

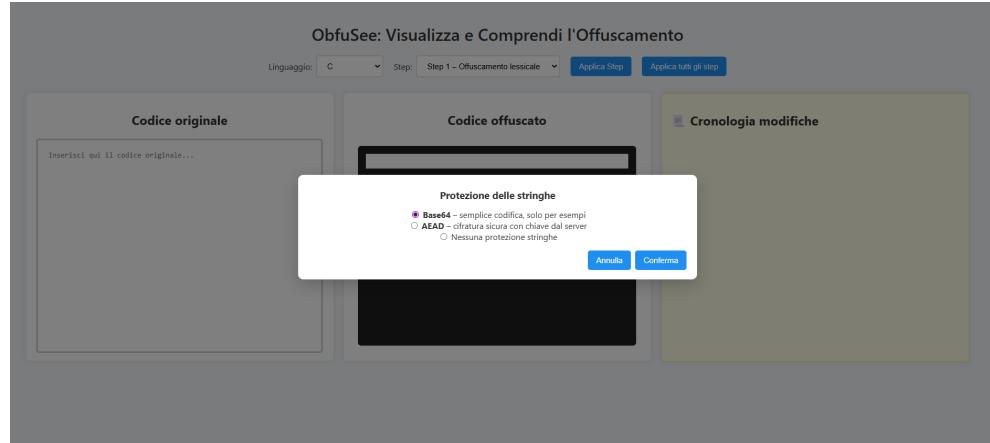


Figura 4.6: Scelta finale al click su ”Applica tutti gli step”

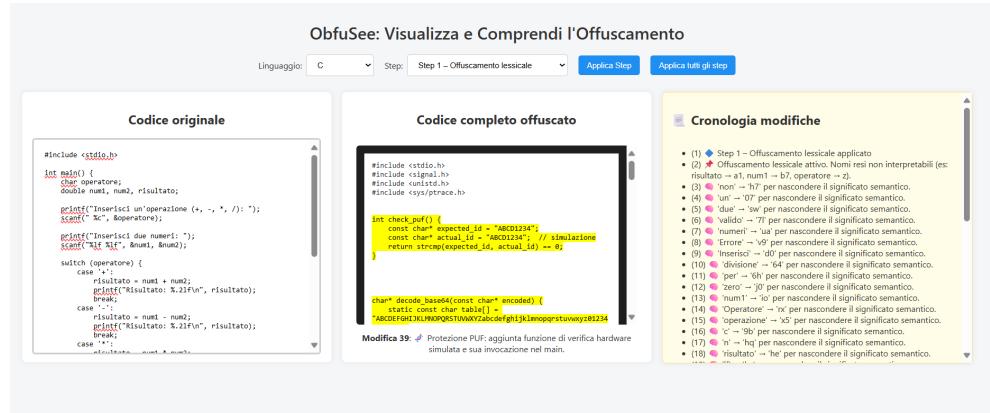


Figura 4.7: Codice completo offuscato

4.3 Gli step di offuscamento

In questo paragrafo verranno analizzati gli step di offuscamento proposti nella piattaforma (già descritti anche nei capitoli precedenti) e per ognuno di essi verrà mostrato un esempio concreto.

4.3.0.1 Step 1 - Offuscamento lessicale

In questo step vengono rinominati identificatori (variabili, funzioni) con nomi arbitrari, spesso generati casualmente. L'obiettivo è confondere il lettore e rendere il codice meno leggibile, senza alterarne il comportamento.

4. PROGETTAZIONE E Sperimentazione di una pipeline ibrida

Esempio : Questo codice in linguaggio C

```
int somma(int a, int b) {
    return a + b;
}

int main() {
    int x = 5;
    int y = 7;
    int risultato = somma(x, y);
    printf("Il risultato è: %d\n", risultato);
    return 0;
}
```

Figura 4.8: Codice C originale

Diventa

```
int ZqX_93dK(int kL0_a, int v9M_b) {
    return kL0_a + v9M_b;
}

int main() {
    int hT1u_ = 5;
    int GgL9q = 7;
    int _aZpQ3x = ZqX_93dK(hT1u_, GgL9q);
    printf("Il risultato è: %d\n", _aZpQ3x);
    return 0;
}
```

Figura 4.9: Codice C offuscato mediante offuscamento lessicale

Gli identificatori sono diventati stringhe pseudo-casuali, che rendono difficile intuire il loro scopo. Il codice funziona identicamente, ma chi lo legge deve perdere molto più tempo per capirne il significato.

4.3.1 Step 2 – Offuscamento del flusso di controllo

L’offuscamento del flusso di controllo avviene mediante l’introduzione di **opaque predicates**, ovvero condizioni logicamente inutili ma sempre vere, che mascherano il vero flusso di esecuzione.

Esempio: Inserimento di una condizione logica inutile in C

4. PROGETTAZIONE E Sperimentazione di una Pipeline ibrida

```
if (b7 * b7 + 1 >= 1) {  
    // codice utile  
}
```

Figura 4.10: Condizione logica

4.3.2 Step 3 – Trasformazione dei dati

L’offuscamento dei dati consiste nel trasformare i valori utilizzati dal programma tramite operazioni matematiche o logiche, in modo da renderli meno riconoscibili e più difficili da analizzare. I dati vengono memorizzati o manipolati in forma mascherata e solo all’occorrenza vengono decodificati, senza alterare il risultato finale del programma.

Esempio: Offuscamento tramite operazione XOR in C

```
// Codice originale  
double risultato = a + b;  
  
// Dopo trasformazione  
int x = 1234;  
double tmpA = (int)a ^ x;  
double tmpB = (int)b ^ x;  
double risultato = (tmpA ^ x) + (tmpB ^ x);
```

Figura 4.11: Offuscamento XOR

I valori a e b sono mascherati con uno XOR prima dell’uso. La leggibilità è compromessa, e strumenti di analisi statica dovranno “decifrare” il significato.

4.3.3 Step 4 – Tecniche Anti-Debugging

Per ostacolare le tecniche di analisi dinamica, viene introdotta una funzione dedicata (`anti_debug_check()`), la quale effettua un controllo sull’eventuale presenza di un debugger attivo. Se viene rilevato un

4. PROGETTAZIONE E SPERIMENTAZIONE DI UNA PIPELINE IBRIDA

processo di debug (ad esempio con strumenti come gdb), il programma reagisce terminando immediatamente la propria esecuzione.

Questa strategia ha due effetti principali:

- **Compromette l'attività dell'analista**, che non può eseguire passo passo il programma né monitorarne lo stato interno.
- **Aumenta la resilienza del software contro il reverse engineering**, rendendo più difficile comprendere il flusso logico e il comportamento del codice durante il runtime.

Esempio: Anti-Debugging in C

```
// Funzione anti-debugging (inserita dopo gli #include)
void anti_debug_check() {
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == -1) {
        raise(SIGKILL);
    }
}

// All'interno del main()
int main() {
    anti_debug_check(); // <-- chiamata evidenziata
    // resto del codice...
}
```

Figura 4.12: Anti-Debugging in C

In questo modo, se il programma viene eseguito in un ambiente monitorato, il controllo fallisce e il processo viene forzatamente interrotto. Tale meccanismo costituisce una barriera efficace contro il debugging interattivo, impedendo a un attaccante di osservare variabili, punti di interruzione o flussi di esecuzione in tempo reale. Di conseguenza, l'analisi dinamica diventa più complessa e richiede strumenti avanzati o tecniche di bypass, aumentando significativamente il costo dell'attacco.

4.3.4 Step 5 - Codifica Base64

In questo step le stringhe letterali vengono codificate in Base64 e poi decodificate a runtime tramite la funzione `decode_base64()` inserita

4. PROGETTAZIONE E Sperimentazione di una pipeline ibrida

subito dopo gli import. Base64 è una semplice trasformazione di rappresentazione (binario → testo ASCII) e non fornisce segretezza crittografica¹. L'effetto desiderato è rendere meno immediata l'estrazione di stringhe tramite strumenti di analisi statica elementare (ad es. strings), poiché nel binario non compaiono in chiaro ma nella loro forma codificata. A runtime, l'interprete o compilatore esegue normalmente la chiamata a `decode_base64()`, ricostruendo la stringa originale e preservando la funzionalità del programma.

4.3.4.1 Come funziona `decode_base64()`?

La funzione applica l'algoritmo standard di decodifica: mappa ciascun simbolo Base64 in un valore a 6 bit, accumula i bit in un registro, ed emette un byte ogni 8 bit disponibili fino a ricostruire la sequenza originaria. Nell'implementazione mostrata:

- scorre i caratteri della stringa codificata e, per quelli validi, aggiorna un accumulatore a 6 bit;
- quando l'accumulatore raggiunge almeno 8 bit, estrae un byte e lo scrive nel buffer di output;
- termina aggiungendo il carattere di fine stringa \0 e restituisce un puntatore alla memoria allocata.

4.3.4.2 Perché complica l'analisi statica?

Le stringhe non sono più in chiaro nel binario: strumenti come strings non le riconoscono direttamente.

Tuttavia, Base64 è reversibile senza chiave: un'analisi statica più attenta può identificare pattern Base64 e decodificarli; strumenti dinamici o

¹Base64 non è una cifratura: non impiega alcuna chiave segreta e non offre confidenzialità. È una codifica di rappresentazione che rende i dati compatibili con canali testuali ed è banalmente reversibile.

4. PROGETTAZIONE E SPERIMENTAZIONE DI UNA PIPELINE IBRIDA

di instrumentazione vedranno comunque il valore in chiaro a runtime. In altre parole, si tratta di offuscamento leggero utile contro ispezioni superficiali, non di protezione crittografica.

```
// Funzione per decodificare una stringa Base64 (semplificata)
char* decode_base64(const char* input) {
    const char table[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
    int len = strlen(input), i, j;
    char *decoded = malloc(len * 3 / 4 + 1);
    if (!decoded) return NULL;

    int val = 0, valb = -8;
    for (i = 0, j = 0; i < len; i++) {
        char* p = strchr(table, input[i]);
        if (!p) continue;
        val = (val << 6) + (p - table);
        valb += 6;
        if (valb >= 0) {
            decoded[j++] = (val >> valb) & 0xFF;
            valb -= 8;
        }
    }
    decoded[j] = '\0';
    return decoded;
}
```

Figura 4.13: Funzione `decode_base64()`

Nell'esempio sottostante, la stringa “risultato” è stata codificata in Base64 per nascondere il messaggio a strumenti di analisi statica come strings. Solo a runtime viene eseguita la funzione (`decode_base64()`) che recupera la stringa originale e la stampa. Questo tipo di offuscamento rende più difficile individuare messaggi sensibili nel binario compilato, aumentando la sicurezza contro il reverse engineering.

```
int main() {
    double risultato = 10.0;

    // Messaggio codificato in Base64: "Risultato: %.2lf\n"
    const char* base64_msg = "UmlzdWx0YXRvOiAlljJmXG4=";

    // Decodifica al momento dell'esecuzione
    char* messaggio = decode_base64(base64_msg);
    if (messaggio) {
        printf(messaggio, risultato);
        free(messaggio);
    } else {
        printf("Errore nella decodifica del messaggio.\n");
    }

    return 0;
}
```

Figura 4.14: Main con codifica base64

4. PROGETTAZIONE E SPERIMENTAZIONE DI UNA PIPELINE IBRIDA

4.3.5 Step 6 - Cifratura AEAD

L'idea di base di questo step è di sostituire le stringhe in chiaro presenti nel sorgente in **blob cifrati**² e inserire nel codice delle chiamate del tipo *dec_aead(blob, id)*. A runtime, il programma chiede al server la chiave associata a **id**, decifra il blob in memoria e usa il testo. La chiave non è nel binario: è custodita sul server. In assenza di chiave (o se il blob è alterato) la decifratura fallisce.

4.3.6 Perchè AEAD ?

Usiamo una tecnica di cifratura **AEAD** (Authenticated Encryption with Associated Data), ovvero una modalità che garantisce **confidenzialità** (il testo non è leggibile senza chiave) e **integrità-autenticità** (manomissioni del ciphertext³ vengono rilevate). AEAD definisce un'interfaccia e più algoritmi concreti; fra i più diffusi ci sono **AES-GCM** e **ChaCha20-Poly1305**. [13] [14] [15]

4.3.6.1 Come funziona?

- **Fase di generazione del codice (build):** per ogni stringa si genera un nonce⁴ e si cifra con un algoritmo AEAD (es. AES-GCM). Si ottiene (*nonce || ciphertext || tag*), poi codificato in Base64 per l'inserimento nel sorgente. La chiave simmetrica usata viene registrata sul server e identificata da un ID. Nel codice, la stringa è rimpiazzata da *dec_aead("blob_b64", "id")*.

²Un blob cifrato (encrypted blob) è un blocco di dati (Binary Large Object, da cui “blob”) che è stato protetto con un algoritmo di cifratura.

³Ciphertext significa testo cifrato: è il risultato dell'applicazione di un algoritmo di cifratura (encryption) su un testo in chiaro (plaintext).

⁴Nonce = Number used once. È un numero casuale o un valore univoco che viene generato una sola volta e serve per garantire sicurezza nelle operazioni crittografiche

4. PROGETTAZIONE E SPERIMENTAZIONE DI UNA PIPELINE IBRIDA

- **Fase di esecuzione (runtime):** *dec_aead* ricava la chiave contattando l'endpoint del server (es. `/k?id=<id>`), decifra in RAM e restituisce il testo in chiaro alla funzione chiamante. Se il ciphertext è stato modificato, AEAD rigetta la decifrazione (proprietà di autenticazione). [16]

4.3.6.2 Motivazioni di sicurezza

- **Niente chiave nel binario:** anche analizzando l'eseguibile, le stringhe restano cifrate; la chiave sta fuori e può essere protetta, ruotata o revocata sul server.
- **Integrità garantita:** AEAD include un tag di autenticazione; se qualcuno altera i dati, la verifica fallisce.
- **Gestione operativa migliore:** il server può implementare policy (autorizzazione, logging, rate-limit) e pratiche note come envelope encryption o KMS: la chiave di dati è protetta da un'ulteriore chiave gestita dal servizio. [17] [18]

4.3.6.3 Cosa vede l'utente nella UI ?

- Nel codice trasformato compaiono chiamate come *dec_aead("...","ID")*.
- La differenza con lo Step 5 (Base64) è chiarita nella modale di scelta mostrata all'utente quando clicca su “Applica tutti gli step”: Base64 è solo codifica (facilmente reversibile), AEAD è cifratura con chiave esterna (non presente nel file). In particolare, quando l'utente cliccherà sul bottone “Applica tutti gli step” gli verrà chiesto quale delle due modalità andare ad utilizzare.

4. PROGETTAZIONE E SPERIMENTAZIONE DI UNA PIPELINE IBRIDA

4.3.6.4 Scelta della protezione delle stringhe quando si applicano tutti gli step

Quando l'utente clicca “**Applica tutti gli step**”, la piattaforma apre una finestra di scelta che chiede quale protezione delle stringhe usare tra:

- **Base64** (codifica, non cifratura), oppure
- **AEAD** (cifratura con chiave lato server).

L'uso simultaneo delle due modalità non è previsto perché ridondante e inutile:

- **Base64 non aggiunge sicurezza:** è solo una codifica di rappresentazione; se la applichi oltre ad AEAD non ottieni più confidenzialità, aumenti solo complessità e overhead.
- **Ordine Base64 → AEAD:** dopo la codifica Base64 i literal diventano chiamate “opache” (segnate internamente), quindi AEAD non vede più le stringhe da cifrare e non può agire.
- **Ordine AEAD → Base64:** i blob AEAD sono già convertiti in Base64 per essere inseriti nel sorgente; una seconda passata Base64 sarebbe puramente ridondante.
- **Chiarezza e manutenzione:** scegliere una sola modalità evita doppie trasformazioni, diff più confusi e possibili effetti collaterali nella pipeline.

In sintesi, la UI forza una scelta esclusiva (Base64 oppure AEAD) perché solo così si ottiene un risultato coerente: Base64 per un offuscamento didattico e leggero; AEAD per una protezione effettiva con chiave esterna.

4.4 Esempio pratico: Offuscamento di una calcolatrice

4.4.1 Introduzione all'esempio

Per dimostrare l'efficacia della pipeline di offuscamento sviluppata, è stato scelto come caso di studio un semplice programma in linguaggio C che implementa una calcolatrice da terminale, in grado di eseguire le quattro operazioni aritmetiche di base su due numeri in virgola mobile.

Questo esempio è stato scelto per la sua semplicità e per la chiarezza del flusso di esecuzione, che rende evidenti gli effetti di ciascuno step di offuscamento.

4.4.2 Codice Originale

Nei prossimi paragrafi andremo a mostrare come il nostro codice di esempio di una calcolatrice viene offuscato, seguendo gli step definiti in precedenza, mediante la piattaforma ObfuSee. Il codice originale preso in considerazione è riportato di seguito:

4. PROGETTAZIONE E SPERIMENTAZIONE DI UNA PIPELINE IBRIDA

```
1 #include <stdio.h>
2
3 int main() {
4     char operatore;
5     double num1, num2, risultato;
6
7     printf("Inserisci un'operazione (+, -, *, /): ");
8     scanf(" %c", &operatore);
9
10    printf("Inserisci due numeri: ");
11    scanf("%lf %lf", &num1, &num2);
12
13    switch (operatore) {
14        case '+':
15            risultato = num1 + num2;
16            printf("Risultato: %.2lf\n", risultato);
17            break;
18        case '-':
19            risultato = num1 - num2;
20            printf("Risultato: %.2lf\n", risultato);
21            break;
22        case '*':
23            risultato = num1 * num2;
24            printf("Risultato: %.2lf\n", risultato);
25            break;
26        case '/':
27            if (num2 != 0) {
28                risultato = num1 / num2;
29                printf("Risultato: %.2lf\n", risultato);
30            } else {
31                printf("Errore: divisione per zero\n");
32            }
33            break;
34        default:
35            printf("Operatore non valido\n");
36
37    }
38
39    return 0;
40 }
```

Figura 4.15: Calcolatrice in C

4.4.3 Processo di offuscamento passo per passo

La piattaforma ObfuSee, realizzata per questo progetto, consente di applicare ciascuno dei sei step di offuscamento separatamente. Di seguito viene illustrato come il codice viene modificato progressivamente ad ogni fase.

4.4.3.1 Step 1 - Offuscamento lessicale

In questo step sono stati modificati i nomi delle variabili e delle funzioni con identificatori casuali o criptici.

4. PROGETTAZIONE E Sperimentazione di una Pipeline ibrida

The screenshot shows the ObfuSee interface with the following details:

- Linguaggio:** C
- Step:** Step 1 – Offuscamento lessicale
- Cronologia modifiche:**
 - (1) Offuscamento lessicale attivo. Nomi resi non interpretabili (es: risultato → a1 num1 → b7 operatore → z).
 - (2) 'non' → '91' per nascondere il significato semantico.
 - (3) 'un' → 'm7' per nascondere il significato semantico.
 - (4) 'due' → 'xg' per nascondere il significato semantico.
 - (5) 'valore' → '99' per nascondere il significato semantico.
 - (6) 'summa' → '9c' per nascondere il significato semantico.
 - (7) 'tutte' → 'cv' per nascondere il significato semantico.
 - (8) 'Inserisci' → 'qv' per nascondere il significato semantico.
 - (9) 'divisione' → 'ou' per nascondere il significato semantico.
 - (10) 'per' → 'ej' per nascondere il significato semantico.
 - (11) 'ogni' → '9d' per nascondere il significato semantico.
 - (12) 'num' → 'zj' per nascondere il significato semantico.
 - (13) 'Operatore' → 'Zx' per nascondere il significato semantico.
 - (14) 'operazione' → 'kr' per nascondere il significato semantico.
 - (15) 'C' → '37' per nascondere il significato semantico.
 - (16) 's' → '9f' per nascondere il significato semantico.
 - (17) 'risultato' → '3m' per nascondere il significato semantico.
 - (18) 'If' → '9c' per nascondere il significato semantico.

Figura 4.16: Calcolatrice: Offuscamento lessicale

4.4.3.2 Step 2 – Offuscamento del flusso di controllo

Sono state introdotte diverse opaque predicate, condizioni sempre vera ma scritte in modo non ovvio, per complicare l'analisi statica e confondere il reverse engineer.

The screenshot shows the ObfuSee interface with the following details:

- Linguaggio:** C
- Step:** Step 2 – Flusso di controllo
- Cronologia modifiche:**
 - (1) Offuscamento del flusso: inseriti più opaque predicates per confondere il flusso logico.

Figura 4.17: Calcolatrice: Offuscamento del flusso di controllo

4.4.3.3 Step 3 - Trasformazione dei dati

Le costanti e i messaggi printf sono stati sostituiti con variabili o trasformazioni più complesse per rendere il codice meno leggibile.

4. PROGETTAZIONE E Sperimentazione di una Pipeline IBRIDA

The screenshot shows the ObfuSee interface with the following sections:

- Linguaggio:** C
- Step:** Step 3 – Trasformazione dei dati
- Codice originale:**

```
#include <stdio.h>
int main() {
    char operator;
    double num1, num2, risultato;
    printf("Inserisci un'operazione (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Inserisci due numeri: ");
    scanf("%lf %lf", &num1, &num2);

    switch(operator) {
        case '+':
            risultato = num1 + num2;
            printf("Risultato: %.2lf\n", risultato);
            break;
        case '-':
            risultato = num1 - num2;
            printf("Risultato: %.2lf\n", risultato);
            break;
        case '*':
            risultato = num1 * num2;
            printf("Risultato: %.2lf\n", risultato);
            break;
        case '/':
            risultato = num1 / num2;
            printf("Risultato: %.2lf\n", risultato);
            break;
    }
}
```
- Codice offuscato:**

```
case '+':
    risultato = ((num1 == num1) ? num1 : num0) +
    ((num2 == num2) ? num2 : num0);
    printf("Risultato: %.2lf\n", risultato);
    break;
case '-':
    risultato = ((num1 == num1) ? num1 : num0) -
    ((num2 == num2) ? num2 : num0);
    printf("Risultato: %.2lf\n", risultato);
    break;
case '*':
    risultato = num1 * num2;
    printf("Risultato: %.2lf\n", risultato);
    break;
case '/':
    if ((num1 == num1) && (num2 != "0")) {
        risultato = num1 / num2;
        printf("Risultato: %.2lf\n", risultato);
    } else {
        printf("Errore: divisione per zero\n");
    }
}
```
- Cronologia modifiche:**
 - (1) Offuscamento dei dati: valori letterali trasformati in condizioni logiche.
 - (2) $x < 0 \rightarrow (x == x \&& x != \text{'0'})$
 - (3) $a * b \rightarrow (a < b) || b < a$
 - (4) true $\rightarrow (\text{A} == \text{A})$
 - (5) false $\rightarrow (\text{A} != \text{A})$
 - (6) \leftarrow Sottrazione offuscata con inversione
 - (7) \leftarrow Sottrazione offuscata con negazione condizionata

Figura 4.18: Calcolatrice: Trasformazione dei dati

4.4.3.4 Step 4 - Tecniche Anti-Debugging

È stata inserita una funzione che termina il programma se viene rilevato l'uso di un debugger.

The screenshot shows the ObfuSee interface with the following sections:

- Linguaggio:** C
- Step:** Step 4 – Anti-Debugging
- Codice originale:**

```
#include <stdio.h>
int main() {
    char operator;
    double num1, num2, risultato;
    printf("Inserisci un'operazione (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Inserisci due numeri: ");
    scanf("%lf %lf", &num1, &num2);

    switch(operator) {
        case '+':
            risultato = num1 + num2;
            printf("Risultato: %.2lf\n", risultato);
            break;
        case '-':
            risultato = num1 - num2;
            printf("Risultato: %.2lf\n", risultato);
            break;
        case '*':
            risultato = num1 * num2;
            printf("Risultato: %.2lf\n", risultato);
            break;
        case '/':
            risultato = num1 / num2;
            printf("Risultato: %.2lf\n", risultato);
            break;
    }
}
```
- Codice offuscato:**

```
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>

void anti_debug_check() {
    if (ptrace(PTRACE_ATTACH, 0, 0, 0) == -1)
        raise(SIGTRAP);
}

int main() {
    anti_debug_check();
    char operator;
    double num1, num, risultato;
}
```
- Cronologia modifiche:**
 - (1) Anti-Debugging: inserita funzione per rilevare esecuzione sotto debugger, evidenziata nel main.

Figura 4.19: Calcolatrice: Tecniche Anti-Debugging

4.4.3.5 Step 5 - Codifica Base64

Le stringhe visualizzate dall'utente vengono codificate in base64 e decodificate dinamicamente in fase di esecuzione.

4. PROGETTAZIONE E Sperimentazione di una Pipeline ibrida

The screenshot shows the ObfuSee interface with three main sections: 'Codice originale' (Original Code), 'Codice offuscato' (Obfuscated Code), and 'Cronologia modifiche' (Change History).

Codice originale:

```
#include <stdio.h>
int main() {
    char operatore;
    double num1, num2, risultato;
    printf("Inserisci un'operazione (+, -, *, /): ");
    scanf("%c", &operatore);
    printf("Inserisci due numeri: ");
    scanf("%lf", &num1, &num2);
    switch (operatore) {
        case '+':
            risultato = num1 + num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '-':
            risultato = num1 - num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '*':
            risultato = num1 * num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '/':
            risultato = num1 / num2;
            printf("Risultato: %.2f\n", risultato);
            break;
    }
}
```

Codice offuscato:

```
#include <stdio.h>
char *decode_base64(const char *encoded) {
    static const char table[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
    int len = strlen(encoded);
    char *decoded = malloc(len);
    int i, j, val;
    for (i = 0; i < len; i++) {
        char *p = strchr(table, encoded[i]);
        if (p != NULL) {
            val = p - table;
            if (val < 64) {
                decoded[i] = val;
            } else continue;
            val += 64;
            if (val > 90) {
                decoded[i] = (char)((val >> 6) & 0x0f);
                val -= 90;
            }
            decoded[i] |= (char)(val & 0x0f);
        }
    }
    Modifica 1: Offuscamento crittografico: stringhe codificate in Base64 e decodificate a runtime.
}
```

Cronologia modifiche:

- (1) Offuscamento crittografico: stringhe codificate in Base64 e decodificate a runtime.

Figura 4.20: Calcolatrice: Codifica Base64 funzione `decode_base64`

The screenshot shows the ObfuSee interface with three main sections: 'Codice originale' (Original Code), 'Codice offuscato' (Obfuscated Code), and 'Cronologia modifiche' (Change History).

Codice originale:

```
#include <stdio.h>
int main() {
    char operatore;
    double num1, num2, risultato;
    printf("Inserisci un'operazione (+, -, *, /): ");
    scanf("%c", &operatore);
    printf("Inserisci due numeri: ");
    scanf("%lf", &num1, &num2);
    switch (operatore) {
        case '+':
            risultato = num1 + num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '-':
            risultato = num1 - num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '*':
            risultato = num1 * num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '/':
            risultato = num1 / num2;
            printf("Risultato: %.2f\n", risultato);
            break;
    }
}
```

Codice offuscato:

```
int main() {
    char operatore;
    double num1, num2, risultato;

    printf(decode_base64("So9j2zXjpc2hjHvJ2wX3hni1vbgCcs
    [REDACTED]p0tP1u"));
    scanf(decode_base64("ICV1"), &operatore);

    printf(decode_base64("Su9j2zXjpc2hjHvJ2wX3hni1vbgCcs
    [REDACTED]p0tP1u"));
    scanf(decode_base64("3mew1v5zq--3"), &num1, &num2);

    switch (operatore) {
        case '+':
            risultato = num1 + num2;
            printf(decode_base64("Um1z1d0yXh0iA1i1j7z1lo"),
            Modifica 1: Offuscamento crittografico: stringhe codificate in Base64 e decodificate a runtime.
}
```

Cronologia modifiche:

- (1) Offuscamento crittografico: stringhe codificate in Base64 e decodificate a runtime.

Figura 4.21: Calcolatrice: Codifica Base64

4.4.3.6 Step 6 - Cifratura AEAD

In questo step le stringhe mostrate all’utente non sono più presenti in chiaro nel codice: vengono sostituite da blob cifrati e recuperate solo a runtime tramite una funzione di decifrazione. Rispetto allo step 5 (Base64), qui si usa cifratura autentificata (AEAD), quindi senza la chiave corretta il testo non è leggibile e ogni manomissione viene rilevata. Nel dettaglio , nel main tutte le printf e scanf con stringhe letterali sono rimpiazzate da chiamate del tipo:

```
printf(dec_aead("BLOB_BASE64", "46e754ba095591d7"));
scanf(dec_aead("BLOB_BASE64", "cd6f7bd27254a7e6"), &operatore);
```

Figura 4.22: Calcolatrice: esempio di printf e scanf cifrate

4. PROGETTAZIONE E SPERIMENTAZIONE DI UNA PIPELINE IBRIDA

Ciascun BLOB_BASE64 contiene (nonce || ciphertext || tag) della stringa originale. Lo stesso schema si ripete per i prompt dei due operandi e per i messaggi dei risultati/casi di errore.

The screenshot shows the ObfuSee interface with three main sections: 'Codice originale' (Original Code), 'Codice offuscato' (Obfuscated Code), and 'Cronologia modifiche' (Modification History). The 'Codice originale' section contains the original C code for a calculator. The 'Codice offuscato' section shows the same code after being processed by the obfuscation steps, appearing significantly more complex and unreadable. The 'Cronologia modifiche' section at the bottom indicates that step 6 (AEAD encryption) has been applied.

```
#include <stdio.h>
int main()
{
    char operatore;
    double num1, num2, risultato;
    printf("Inserisci un'operazione (+, -, *, /): ");
    scanf(" %c", &operatore);
    printf("Inserisci due numeri: ");
    scanf("%lf %lf", &num1, &num2);
    switch (operatore) {
        case '+':
            risultato = num1 + num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '-':
            risultato = num1 - num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '*':
            risultato = num1 * num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        case '/':
            risultato = num1 / num2;
            printf("Risultato: %.2f\n", risultato);
            break;
        default:
            printf("Operatore non valido.\n");
    }
    return 0;
}
```

```
#include <openssl/evp.h>
#include <openssl/aes.h>
#include <string.h>
#include <string.h>
#include <stdio.h>

/* OBFUSEE_FH */
static unsigned char* _obf_p64dec(const char* b64, int* outlen)
{
    BIO *b = BIO_new_mem_buf(b64, -1);
    BIO_set_flags(b, BIO_FLAGS_BIO_NOCLOSE);
    b = BIO_push(b, B);
    B->set_flags(B, BIO_FLAGS_BASE64_NO_MJ);
    const unsigned char* buf = malloc(strlen(b64));
    *outlen = B->read(b, buf, strlen(b64));
    B->free_all();
    return buf;
}

static int get_running_key(const char* id, unsigned char**
```

Figura 4.23: Calcolatrice: codice offuscato mediante cifratura aead

4.5 Codice originale VS Codice Offuscato

Al termine dei singoli step di offuscamento è possibile combinare tutti gli step ed avere il codice completo offuscato. Per fare questo basterà cliccare sul pulsante “Applica tutti gli step”. Non è necessario eseguire in precedenza i singoli step, si può anche avere la combinazione completa anche sin da subito, ma è opportuno eseguire un percorso incrementale per capire bene come queste singole tecniche funzionano. Al clic su “Applica tutti gli step” la piattaforma richiede di scegliere una sola modalità di protezione delle stringhe (Base64 oppure AEAD), poiché l’uso simultaneo è ridondante; la motivazione è stata già discussa nei paragrafi precedenti. Il codice risultante presenta una complessità strutturale e visiva significativamente aumentata rispetto all’originale.

4.6 Considerazioni finali

L’esempio della calcolatrice evidenzia chiaramente come ciascuna tecnica contribuisca ad aumentare la difficoltà di analisi e comprensione del

4. PROGETTAZIONE E Sperimentazione di una pipeline ibrida

codice:

- Lo step 1 rende meno intuitivi i nomi (variabili/funzioni), ostacolando la lettura semantica.
- Lo step 2 introduce rami logici fuorvianti (opaque predicates), complicando il tracciamento del flusso.
- Lo step 3 riduce le evidenze statiche dei dati (costanti e messaggi), rendendo più onerosa l’inferenza.
- Lo step 4 ostacola il debugging (terminazione/contromisure se si rileva il debugger).
- Lo step 5 nasconde i dati con codifica (Base64), utile contro strumenti banali (es. strings), ma reversibile e dunque non assimilabile a cifratura.
- Lo step 6 cifra le stringhe con AEAD (es. AES-GCM), garantendo confidenzialità e integrità; la chiave non è nel binario ma viene fornita a runtime (variabile d’ambiente o server). Senza chiave i testi restano illeggibili, e manomissioni del blob vengono rilevate.

Nel complesso, la pipeline ibrida incrementa la resistenza sia contro analisi statiche (nomi offuscati, dati trasformati/cifrati) sia contro analisi dinamiche (anti-debug). Lo Step 6 introduce un salto qualitativo: sposta l’anello critico dalla distribuzione del binario alla gestione esterna della chiave, che può essere protetta, ruotata e revocata. L’onere è la disponibilità del canale (chiave e, se previsto, server), e la consapevolezza che a esecuzione avviata testo e chiave esistono in RAM per un tempo limitato.

Nel capitolo successivo verranno analizzati i risultati sperimentali ottenuti applicando strumenti di analisi (ad es. strings, gdb, tecniche di reverse engineering), con confronto dei binari generati e valutazioni qualitative sulla resilienza del codice—incluse le differenze tra Base64 e AEAD nella protezione delle stringhe.

CAPITOLO 5

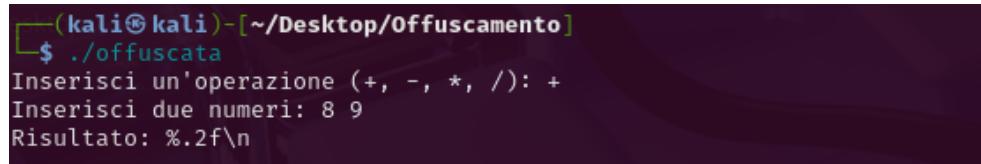
ANALISI DEI RISULTATI OTTENUTI

In questo capitolo vengono illustrati i risultati ottenuti applicando le tecniche di offuscamento descritte nel capitolo precedente a un semplice programma scritto in linguaggio C: una calcolatrice aritmetica da terminale. L’obiettivo è mostrare come ciascuno dei sei step di offuscamento abbia contribuito a rendere il codice più difficile da comprendere, analizzare e manipolare tramite strumenti di debugging e reverse engineering.

5.1 Comportamento in esecuzione — variante Base64 (Step 5)

Il programma offuscato è stato eseguito normalmente da terminale, confermando il corretto funzionamento. La figura seguente mostra un esempio di esecuzione su Linux.

5. ANALISI DEI RISULTATI OTTENUTI



```
(kali㉿kali)-[~/Desktop/Offuscamento]
$ ./offuscata
Inserisci un'operazione (+, -, *, /): +
Inserisci due numeri: 8 9
Risultato: %.2f\n
```

Figura 5.1: Esecuzione normale del programma offuscato

Come si può notare, nonostante il codice sia stato trasformato, l’output visivo è coerente con quello della versione originale. Tuttavia, la stringa `%.2f` non è più esplicitamente visibile nel codice, in quanto è stata codificata in Base64 e decifrata solo a runtime. Si tratta quindi di un offuscamento leggero (codifica, non cifratura): utile contro strumenti molto semplici (es. strings), ma reversibile.

5.2 Comportamento in esecuzione — variante AEAD (Step 6)

Con la variante AEAD, le stringhe non sono più presenti in chiaro nel binario: sono cifrate e ricostruite solo a runtime. L’esecuzione risulta quindi corretta solo se il programma può accedere alla chiave (fornita dall’ambiente o da server, a seconda della configurazione). All’atto pratico, l’interazione a terminale è identica alla versione originale (es. inserimento operatore, due operandi, stampa del risultato). La differenza è interna: ad ogni stampa/lettura, il codice invoca `dec_aead(blob, id)`, richiede la chiave e decifra in memoria la stringa necessaria. In assenza di chiave o con chiave errata, la verifica di integrità (tag AEAD) fallisce e il testo non viene ricostruito (stampe mancanti o terminazione, in base alla gestione degli errori). Questo è il comportamento previsto: senza chiave, niente testo. La chiave non è mai contenuta nel binario.

5. ANALISI DEI RISULTATI OTTENUTI

5.2.0.1 Come si esegue ?

Tramite terminale linux generiamo la nostra chiave attraverso il seguente comando (la chiave generata non viene mostrata in questo documento): `openssl rand -base64 32`. In questo modo lo Step 6 userà proprio quella chiave per cifrare i blob. Successivamente avviamo la nostra web app proprio con la chiave appena generata, in questo modo:

```
export AEAD_KEY_B64="INCOLLA QUI LA TUA BASE64"  
python3 app.py.
```

Avviata la nostra web app, copiamo e incolliamo il nostro codice, applichiamo tutti gli step (modalità AEAD) o soltanto lo step 6, e salviamo il codice offuscato ottenuto come *calcolatriceOffuscata.c*. Compiliamo collegando OpenSSL in questo modo

```
gcc calcolatrice.c -o calcolatrice -lssl -lcrypto
```

ed eseguiamo con la chiave precedentemente creata

```
export AEAD_KEY_B64="INCOLLA QUI LA TUA BASE64"  
../calcolatrice.c.
```

Appena il programma parte se tutto è andato a buon fine riusciamo ad effettuare le nostre operazioni altrimenti se la chiave non è impostata o è diversa, i messaggi non compariranno correttamente.

5.3 Analisi con Debugger

L'applicazione è stata testata anche tramite gdb (GNU Debugger)¹, con il seguente comando:

¹Il GNU Debugger (GDB) è un debugger portatile che funziona su molti sistemi Unix-like e funziona per molti linguaggi di programmazione, tra cui Ada, Assembly, C, C++, D, Fortran, Haskell, Go, Objective-C, OpenCL C, Modula-2, Pascal, Rust etc... Rileva i problemi in un programma lasciandolo in esecuzione e consente agli utenti di esaminare diversi registri.

5. ANALISI DEI RISULTATI OTTENUTI

```
(kali㉿kali)-[~/Desktop]
$ gdb offuscata
GNU gdb (Debian 16.3-1) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word" ...
Reading symbols from offuscata ...
(No debugging symbols found in offuscata)
(gdb) run
Starting program: /home/kali/Desktop/offuscata
[Thread debugging using libthread_db enabled] for zero(n);
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program terminated with signal SIGKILL, Killed.
The program no longer exists.
(gdb) █
```

Figura 5.2: Esecuzione con gdb

Come visibile, il programma viene immediatamente terminato con un segnale SIGKILL. Ciò è dovuto all'inserimento, tramite lo Step 4, di un controllo anti-debug basato su ptrace(), che rileva l'attacco e chiude forzatamente il processo. Questo dimostra l'efficacia della protezione contro il debugging statico e dinamico.

5.4 Reverse Engineering con strings - Codifica Base64 (Step 5)

Per valutare l'efficacia delle tecniche di offuscamento applicate, è stata eseguita un'analisi statica dei binari generati (originale e offuscato) utilizzando il comando **strings**, che consente di estrarre tutte le stringhe leggibili da un file binario. Questo tipo di analisi è comunemente impiegata durante operazioni di reverse engineering per ottenere informazioni sensibili o comprendere il comportamento di un programma. Il file `originale.txt` è stato ottenuto eseguendo il comando

5. ANALISI DEI RISULTATI OTTENUTI

```
(kali㉿kali)-[~/Desktop/Offuscamento]
$ strings originale > originale.txt
```

Figura 5.3: Analisi comparativa con `strings`, file originale

Contiene tutte le stringhe in chiaro presenti nel programma originale.

Tra queste, si notano:

- I messaggi utente del programma.
- I nomi di funzioni standard della libreria C(`printf`,`scanf`,`puts`).
- I nomi del file sorgente.
- Diversi simboli, segmenti e riferimenti usati durante la compilazione.

Queste informazioni rendono immediatamente chiaro il funzionamento del programma e la sua logica interna, facilitando eventuali operazioni di reverse engineering o modifica. Di seguito un estratto del file `originale.txt`.

5. ANALISI DEI RISULTATI OTTENUTI

```
|x/lib64/ld-linux-x86-64.so.2
|puts
|__libc_start_main
|__cxa_finalize
|printf
|__isoc99_scanf
libc.so.6
GLIBC_2.7
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
Inserisci un'operazione (+, -, *, /):
Inserisci due numeri:
%lf %lf
Risultato: %.2lf
Errore: divisione per zero
Operatore non valido
;*3$"
GCC: (Debian 14.2.0-19) 14.2.0
Scrt1.o
__abi_tag
crtstuff.c
```

Figura 5.4: Estratto del file originale.txt

Invece il file offuscata.txt ottenuto mediante il seguente comando:

```
[kali㉿kali)-[~/Desktop/Offuscamento]
$ strings offuscata > offuscata.txt
```

Figura 5.5: Analisi comparativa con **strings**, file offuscato

presente caratteristiche differenti:

- Alcuni messaggi utente sono ancora presenti in chiaro, ma i messaggi chiave sono stati codificati in Base64.
- È presente la tabella dei caratteri Base64.
- Compiono le funzioni aggiuntive di protezione e offuscamento.

5. ANALISI DEI RISULTATI OTTENUTI

Di seguito un estratto del file offuscata.txt

```
operator e non vedi es
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
;*3$"
GCC: (Debian 14.2.0-19) 14.2.0
Scrt1.o
__abi_tag
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.0
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
offuscata.c
table.0
__FRAME_END__
__DYNAMIC
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
raise@GLIBC_2.2.5
__libc_start_main@GLIBC_2.34
_ITM_deregisterTMCloneTable
puts@GLIBC_2.2.5
__edata
__fini
strlen@GLIBC_2.2.5
strchr@GLIBC_2.2.5
printf@GLIBC_2.2.5
__data_start
__gmon_start__
__dso_handle
__IO_stdin_used
anti_debug_check
malloc@GLIBC_2.2.5
__end
ptrace@GLIBC_2.2.5
__bss_start
main
decode_base64
    isoc99_scanf@GLIBC_2.7
```

Figura 5.6: Estratto del file offuscata.txt

5.5 Reverse Engineering con strings - Variante AEAD (Step 6).

Nel binario offuscato con AEAD le stringhe non compaiono più in chiaro.

L'output di **strings** evidenzia:

- blob in Base64 passati a `dec_aead(..., "id")`, non decodificabili senza chiave;
- riferimenti a librerie/funzioni crittografiche (es. `EVP_DecryptInit_ex, EVP_aes_256_gcm, EVP_DecryptFinal_ex`);

5. ANALISI DEI RISULTATI OTTENUTI

- le funzioni di supporto iniettate (`dec_aead`, `get_runtime_key`, `_obf_b64dec`);
- nella variante “chiave da ambiente” compaiono stringhe come `AEAD_KEY_B64` (in alternativa, per la variante server: `AEAD_KEY_SERVER`, ecc.).

Diversamente dalla sola Base64 (reversibile), `strings` non consente di recuperare i messaggi: i blob sono *cifrati* e l'integrità è tutelata dal tag AEAD (alterazioni → fallimento della decifrazione a runtime).

5.6 Valutazione dell'Efficacia

L'efficacia dell'offuscamento può essere sintetizzata come segue:

Tecnica	Risultato
Offuscamento lessicale	Rende meno leggibile il codice
Offuscamento del controllo	Inserisce branch inutili ma ingannevoli
Trasformazione dati	Codifica le costanti e le stringhe
Anti-debug	Termina il processo sotto debugger
Codifica Base64	Occulta i messaggi a tool semplici
Cifratura AEAD	Stringhe cifrate: senza chiave illeggibili;

Tabella 5.1: Risultati delle tecniche di offuscamento

CAPITOLO 6

CONCLUSIONI

Nel corso di questo lavoro di tesi è stato affrontato in modo approfondito il tema dell’offuscamento del codice, partendo dalle fondamenta teoriche che ne dimostrano l’impossibilità formale, fino ad arrivare alla progettazione e realizzazione di una piattaforma interattiva chiamata ObfuSee. Attraverso l’analisi delle principali tecniche di offuscamento, definite e approfondite nei capitoli precedenti, è stato possibile costruire una pipeline multi-step in grado di trasformare codice sorgente leggibile in una versione logicamente equivalente ma notevolmente più difficile da analizzare per un attaccante o un reverse engineer.

La piattaforma sviluppata consente di visualizzare in modo incrementale ogni trasformazione applicata, offrendo sia una modalità didattica sia uno strumento di analisi della resilienza delle tecniche implementate. I risultati ottenuti, illustrati tramite confronti tra codice originale e offuscato, tool di reverse engineering (es. strings, gdb...) e metriche di complessità, dimostrano un aumento significativo dell’opacità del codice generato e della sua resistenza all’analisi statica.

6. CONCLUSIONI

La tesi ha così raggiunto un duplice obiettivo: da un lato, confermare l'importanza dell'approccio ibrido e incrementale nella costruzione di strategie di offuscamento efficaci; dall'altro, fornire un contributo pratico e replicabile sotto forma di piattaforma interattiva, accessibile e modulare, pensata per supportare studenti, sviluppatori e ricercatori nel comprendere e sperimentare le tecniche di protezione del software.

Questo lavoro di tesi è stato realizzato grazie ad un intenso studio della letteratura e analisi di lavori già esistenti. In particolare per comprendere ed approfondire alcuni aspetti crittografici avanzati sono state utilizzate fonti presenti in rete (come si evince dall'inserimento delle citazioni) e anche supporti di rielaborazione forniti dall'intelligenza artificiale. Il materiale è comunque stato interamente prodotto e revisionato dalla candidata.

6.1 Lavori futuri

Il lavoro svolto offre molteplici spunti per futuri sviluppi, sia dal punto di vista teorico che implementativo:

- **Estensione della piattaforma a nuovi linguaggi:** al momento ObfuSee supporta C, Python, Java, JavaScript e Rust. Un'evoluzione naturale consiste nell'includere altri linguaggi critici per il software embedded o mobile (es. Kotlin, Swift, Assembly).
- **Integrazione con tecniche di offuscamento dinamico:** attualmente la pipeline si basa su trasformazioni statiche. Si potrebbe esplorare l'integrazione di tecniche runtime, come code mutation e self-modifying code, per aumentare ulteriormente la resilienza.
- **Metriche automatiche di valutazione:** aggiungere un modulo che calcoli automaticamente le metriche di complessità, entropia,

6. CONCLUSIONI

ridondanza e confusione, aiuterebbe a misurare oggettivamente il grado di offuscamento ottenuto.

- **Simulazione di attacchi automatici:** un modulo che simula tentativi di de-offuscamento automatico (es. symbolic execution, pattern matching) potrebbe rendere il sistema anche uno strumento di benchmarking della robustezza.
- **Modalità di formazione e gamification:** si potrebbe trasformare ObfuSee in una piattaforma educativa con sfide interattive, valutazione di codice obfuscato e ranking tra utenti, utile per corsi universitari o CTF (Capture The Flag).
- **Rilascio come progetto open source:** rendere la piattaforma pubblicamente accessibile su GitHub, accompagnata da una documentazione dettagliata, favorirebbe l'adozione e la collaborazione nella comunità scientifica e professionale.

RINGRAZIAMENTI

*«Avrai, avrai, avrai
il tuo tempo per andar lontano,
camminerai dimenticando,
ti fermerai sognando...»
- Claudio Baglioni.*

Con queste parole desidero aprire i miei ringraziamenti: perché questo traguardo rappresenta proprio un tempo nuovo, un tempo che mi porterà lontano, fatto di sogni che diventano realtà e di ricordi che restano impressi. Questa tesi non è soltanto la conclusione di un percorso di studi, ma il frutto dell'amore, della fiducia e del sostegno che tante persone mi hanno donato lungo la strada.

Ringrazio con tutto il cuore la mia famiglia, in particolare mamma, papà e Carmen. Siete la mia forza e il mio porto sicuro, la certezza che mi ha permesso di non smettere mai di credere in me stessa. Ogni vostro sacrificio, ogni incoraggiamento e ogni sorriso mi hanno guidata fino a questo giorno. Senza di voi questo traguardo non avrebbe avuto lo stesso valore: siete la mia radice e, al tempo stesso, la mia spinta verso il futuro.

Un grazie speciale al mio relatore, per la sua disponibilità, la pazienza e la preziosa guida scientifica che ha reso possibile la realizzazione di

RINGRAZIAMENTI

questo lavoro. Non è stato soltanto un supporto accademico, ma anche umano, insegnandomi il valore della precisione e della dedizione.

Grazie ai miei amici, che hanno trasformato lo studio in condivisione, la fatica in leggerezza e i momenti di sconforto in nuove risate. Con voi ho imparato che l'amicizia è una delle forze più grandi: siete stati compagni di viaggio insostituibili, capaci di rendere più lieve ogni salita e di raddoppiare la gioia di ogni traguardo. Questo risultato è anche vostro, perché in ogni pagina c'è un po' del vostro sostegno, della vostra presenza e del vostro affetto.

È doveroso ora fermarmi un attimo e provare a trovare le parole giuste per ringraziare ognuno di voi. Voglio partire dalle prime persone che ho conosciuto proprio tra questi banchi: Katia, Rita, Gabriele e Andrea. La loro costante presenza ha reso più leggeri e spensierati momenti di tensione, dimostrandomi che l'università non è solo un percorso di studio, ma anche un intreccio di vite che si arricchiscono a vicenda.

Un ringraziamento dal profondo del cuore va a Katia. Sin dal primo giorno siamo entrate in sintonia come se ci conoscessimo da sempre, e da allora non mi ha mai fatto mancare la sua vicinanza. Insieme abbiamo condiviso sorrisi, fatiche e momenti che porterò sempre con me: con lei ho imparato che l'amicizia autentica è quella che ti sostiene senza bisogno di parole, che ti accompagna silenziosa ma costante lungo il cammino.

Un grazie speciale anche a Rita, sempre pronta a portare serenità quando la tensione prendeva il sopravvento. La sua capacità di trasformare i silenzi pesanti in leggerezza, di ricordarci il valore della calma e della fiducia, ha reso ogni momento più armonioso e ci ha insegnato che la vera forza sta anche nella dolcezza.

Anche i nuovi arrivati in questo percorso, Umberto, Giorgio e Marco, sono stati importanti: la loro presenza ha reso l'ambiente più vivo. Insieme abbiamo vissuto non solo esperienze universitarie, ma anche

RINGRAZIAMENTI

momenti di quotidianità, serate insieme, risate e condivisioni che hanno reso più leggere giornate che sembravano infinite.

Un pensiero speciale va a Costantina, che con la sua solarità contagiosa e la sua spontaneità sincera ha reso più luminoso ogni giorno. La sua presenza costante, il suo modo di esserci con leggerezza e calore, ha illuminato anche le giornate più “ventose” di Fisciano. Con lei basta trascorrere appena cinque minuti perché i brutti pensieri si dissolvano come per magia... e chiunque la conosca sa che non esagero: provare per credere.

A voi, mie coinquiline, che siete state la mia famiglia lontano da casa, voglio dedicare un pensiero dal cuore.

Alle nuove arrivate, il mio augurio è quello di vivere ogni attimo che l'università vi regalerà: assaporate i giorni luminosi e custodite anche quelli più difficili, perché saranno proprio loro a farvi crescere, a rendervi più forti e consapevoli.

Sara, tu mi hai insegnato a non arrendermi mai e a inseguire con coraggio i miei sogni. I tuoi consigli sono stati preziosi, una fonte continua di crescita e ispirazione che mi ha spinto ad andare avanti anche nei momenti più incerti. La tua forza e la tua determinazione hanno acceso in me la voglia di credere che tutto sia possibile.

Fede, anima dolce e silenziosa, ma colma d'amore: sei stata la mia certezza, soprattutto nell'ultimo anno, quando tutto sembrava scivolarmi via. Le tue parole gentili mi hanno sempre rassicurata e i tuoi incoraggiamenti mi hanno spinta ad andare oltre, anche quando non ci credevo nemmeno io.

Hai avuto la capacità di starmi accanto senza mai farmi sentire un peso, con quella tua presenza discreta ma sempre sicura. Nei momenti più fragili sei stata il mio rifugio, nei giorni più sereni la mia compagna di sorrisi. Con te ho imparato il valore della delicatezza, di quei gesti silenziosi che valgono più di mille parole. Ti ringrazio per avermi fatto

RINGRAZIAMENTI

sentire al sicuro, per avermi mostrato che la forza può essere custodita anche nella dolcezza, e ti auguro di continuare a brillare così come sei, con quella luce silenziosa che scalda il cuore di chi ti è accanto.

Pia, la mia lagonegrese di fiducia, hai portato in casa una ventata di allegria e spensieratezza. Sempre pronta ad ascoltare e a trovare una soluzione a ogni problema, sei stata quella luce capace di rendere più semplice anche ciò che sembrava complicato. Con te ho imparato che a volte basta un sorriso sincero per alleggerire anche la giornata più pesante. Ti ringrazio per aver saputo portare leggerezza dove serviva, e forza nei momenti più complessi.

Maria, quante cose abbiamo condiviso negli ultimi due anni. I momenti difficili ci hanno fatto crescere, quelli belli ci hanno arricchite e unite ancora di più. Ricorderò sempre le giornate infinite a chiacchierare, a raccontarci, a sostenerci a vicenda quando sembrava non esserci via d'uscita.

A te, che la vita ha tolto tanto ma ha donato una forza immensa, di quelle che lasciano senza parole. Mi hai insegnato a guardare il mondo con occhi diversi, a trovare la luce anche nei giorni più bui, a non smettere mai di credere che, nonostante tutto, il bello della vita torni sempre a farsi vedere.

Grazie per la tua immensa disponibilità, per la tua gentilezza e per la tua presenza costante, anche quando sarebbe toccato a me starti accanto. Sei stata un esempio di resilienza, di amore e di speranza, e per questo ti porterò sempre nel cuore.

Ti auguro sempre il meglio: di continuare a splendere, di inseguire i tuoi sogni senza mai smettere di crederci e di non dimenticare mai quanto vali. E ricordati che io ci sarò sempre, da qualche parte, pronta ad esserci ogni volta che avrai bisogno. Perché certe persone diventano casa, e tu per me lo sei stata.

RINGRAZIAMENTI

E non posso dimenticare le mie coinquilini acquisite, Martina e Mariarosaria, che hanno rappresentato un tassello importante di questo percorso. Anche con voi ho condiviso momenti che hanno reso questa esperienza più ricca, più completa, più vera. Ognuna di voi, con la sua presenza, ha lasciato un segno speciale nella mia vita.

E a te, Nives, la mia amica di sempre. La mia spalla destra, sempre pronta ad ascoltarmi senza mai giudicare, mia confidente e fonte di ispirazione più profonda. Grazie per esserci sempre stata, per aver vissuto con me intensamente questi anni, consigliandomi e incoraggiandomi a non mollare mai.

Sei la persona che mi conosce meglio di chiunque altro e, nonostante i miei mille difetti, non hai mai smesso di restarmi accanto. Con te ho imparato che l'amicizia vera non ha bisogno di grandi gesti per esistere: basta uno sguardo, una parola, un silenzio condiviso. Sei stata la mia forza nei momenti in cui non ne avevo, il mio rifugio sicuro quando tutto intorno sembrava instabile.

Ti auguro di continuare a splendere, di realizzare i tuoi sogni uno ad uno, e di non dimenticare mai quanto vali. E ricorda, io ci sarò sempre, in ogni traguardo e in ogni caduta, pronta a gioire con te e a rialzarti quando servirà. La nostra amicizia è il filo invisibile che lega le nostre vite, e so che non si spezzerà mai.

A tutti voi, non solo a chi è qui presente ma anche a chi, in questi anni, mi è stato accanto, va il mio pensiero più sincero. Ognuno ha rappresentato qualcosa di unico ed è stato parte fondamentale di questo percorso: avete lasciato in me tracce, insegnamenti e ricordi che porterò sempre nel cuore. E spero, con la stessa intensità, di aver saputo lasciare anch'io qualcosa in voi.

E infine, grazie a me stessa: perché ho imparato che i sogni, se coltivati con costanza e coraggio, possono davvero diventare realtà. Questa tesi è il mio piccolo “tempo per andar lontano”, la prova che ogni passo,

RINGRAZIAMENTI

ogni caduta e ogni rinascita hanno avuto un senso. È il segno della mia determinazione quando tutto sembrava pesante, della mia resilienza quando avrei potuto fermarmi, della mia capacità di credere in me anche nei giorni più bui.

Rappresenta la mia voce interiore che mi ha ricordato di non mollare, la mia forza silenziosa che mi ha portato fin qui. È un traguardo, ma anche un inizio: un promemoria che dentro di me esiste la forza di andare sempre oltre.

REFERENCES

- [1] Sito ufficiale Wikipedia. *Reverse engineering*. n.d. URL: https://it.wikipedia.org/wiki/Reverse_engineering#%E2%80%A2:_text=Nelle%20varie%20discipline-,Architettura,mondo%20informatico%20della%20Motion%20Capture..
- [2] Sito ufficiale Wikipedia. *Offuscamento del codice*. n.d. URL: https://it.wikipedia.org/wiki/Offuscamento_del_codice#%E2%80%A2:_text=In%20programmazione%2C%20l'offuscamento%20del,comprendere%20per%20un%20lettore%20umano..
- [3] Sito ufficiale Google Scholar. *On the (im)possibility of obfuscating programs*. 2012. URL: <https://dl.acm.org/doi/pdf/10.1145/2160158.2160159>.
- [4] Sito ufficiale Wikipedia. *Offuscamento a scopo ricreativo*. URL: https://it.wikipedia.org/wiki/Offuscamento_del_codice.
- [5] Sito ufficiale Google Scholar. *Watermarking, tamper-proofing, and obfuscation - tools for software protection*. 2002. URL: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1027797>.
- [6] Sito ufficiale Wikipedia. *Physical unclonable function*. URL: https://en.wikipedia.org/wiki/Physical_unclonable_function.

REFERENCES

- [7] Trusted Computing Group. *TPM 2.0: A Brief Overview*. Accessed: 2025-09-04. 2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/2019_TCG TPM2_BriefOverview_DR02web.pdf.
- [8] Trusted Computing Group. *TPM 2.0 Library Specification*. Accessed: 2025-09-04. 2019. URL: <https://trustedcomputinggroup.org/resource/tpm-library-specification/>.
- [9] Sito ufficiale Google Scholar. *The Impact of Inter-Procedural Code Obfuscation on Binary Differencing Techniques*. 2023. URL: <https://dl.acm.org/doi/10.1145/3579990.3580007>.
- [10] Sito ufficiale Google Scholar. *An Efficient and Practical Software Obfuscation Framework*. 2022. URL: <https://ieeexplore.ieee.org/document/9834700>.
- [11] Sito ufficiale Google Scholar. *White-Box Cryptography and an AES Implementation*. 2002. URL: <https://www.iacr.org/archive/sac2002/25010417/25010417.pdf>.
- [12] Sito ufficiale Google Scholar. *On White-box Cryptography and Obfuscation*. 2008. URL: https://link.springer.com/chapter/10.1007/978-3-540-71039-4_14.
- [13] Network Working Group. *An Interface and Algorithms for Authenticated Encryption*. 2008. URL: <https://datatracker.ietf.org/doc/rfc5116/>.
- [14] Y. Nir. *ChaCha20 and Poly1305 for IETF Protocols*. 2015. URL: <https://datatracker.ietf.org/doc/rfc7539/>.
- [15] Morris Dworkin (NIST). *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. 2007. URL: <https://csrc.nist.gov/pubs/sp800/38/d/final>.

REFERENCES

- [16] Tink Cryptographic Library. *Crittografia autenticata con dati associati (AEAD)*. URL: <https://developers.google.com/tink/aead?hl=it>.
- [17] AWS Key Management Service. *AWS KMS cryptography essentials*. URL: <https://docs.aws.amazon.com/kms/latest/developerguide/kms-cryptography.html>.
- [18] AWS Key Management Service. *Concepts in the AWS Encryption SDK*. URL: <https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/concepts.html>.

ELENCO DELLE FIGURE

1.1	Esempio codice offuscato	9
1.2	Programma scritto da Ian Phillips	14
4.1	Pagina iniziale	35
4.2	Inserimento codice	35
4.3	Scelta linguaggio	36
4.4	Scelta step	36
4.5	Applicazione Step 1	36
4.6	Scelta finale al click su "Applica tutti gli step"	37
4.7	Codice completo offuscato	37
4.8	Codice C originale	38
4.9	Codice C offuscato mediante offuscamento lessicale	38
4.10	Condizione logica	39
4.11	Offuscamento XOR	39
4.12	Anti-Debugging in C	40
4.13	Funzione <code>decode_base64()</code>	42

ELENCO DELLE FIGURE

4.14 Main con codifica base64	42
4.15 Calcolatrice in C	47
4.16 Calcolatrice: Offuscamento lessicale	48
4.17 Calcolatrice: Offuscamento del flusso di controllo	48
4.18 Calcolatrice: Trasformazione dei dati	49
4.19 Calcolatrice: Tecniche Anti-Debugging	49
4.20 Calcolatrice: Codifica Base64 funzione <code>decode_base64</code> . .	50
4.21 Calcolatrice: Codifica Base64	50
4.22 Calcolatrice: esempio di printf e scanf cifrate	50
4.23 Calcolatrice: codice offuscato mediante cifratura aead . .	51
 5.1 Esecuzione normale del programma offuscato	54
5.2 Esecuzione con gdb	56
5.3 Analisi comparativa con <code>strings</code> , file originale	57
5.4 Estratto del file <code>originale.txt</code>	58
5.5 Analisi comparativa con <code>strings</code> , file offuscato	58
5.6 Estratto del file <code>offuscata.txt</code>	59

ELENCO DELLE TABELLE

3.1 Tabella riassuntiva: Confronto tra tecniche di offuscamento	32
5.1 Risultati delle tecniche di offuscamento	60