On the impossibility of code obfuscation

L’impossibilità dell’offuscamento del codice è un risultato ottenuto nell’ambito della crittografia ed in particolare si distingue in:

- **Black-box security** → vuole garantire la sicurezza di un sistema di cifratura, assumendo che l’attaccante non veda niente del sistema di cifratura, ad eccezione dell’input e dell’output. Il sistema, cioè, viene visto dall’attacante come una scatola nera ed esso (ovvero l’attaccante) può osservare l’input e gli output prodotti e da questi può cercare di fare il reverse engineering → in questo modello, quindi, l’attaccante ha accesso alla minor quantità di dati del sistema, dato che l’attante è in grado solamente di osservarne il comportamento;
- **Grey-box security** → non permette all’attaccante di osservare il funzionamento interno del sistema, ma si ipotizza che l'attaccante abbia un accesso fisico parziale alla chiave crittografica come risultato della perdita di informazioni, quali ad esempio: il consumo di energia o il tempo richiesto per svolgere un’operazione. Tali informazioni, possono essere utilizzate dall’attaccante per capire il funzionamento del sistema;
- **White-box security** → è lo scenario in cui ci immaginiamo di essere, quando parliamo di offuscamento e protezione del codice, in quanto nello scenario White-box l’attaccante ha accesso a tutte le informazioni. Quindi, il processo è in esecuzione su una macchina pienamente controllata dall’attaccante.

> 💡 Esse, quindi, si differenziano in base al tipo di informazione che l’attaccante ha accesso.

La White-box security, quindi, si chiede se è possibile ottenere una sicurezza come quella Black-box, avendo però che l’attaccante ha un accesso White-box al codice. Ovvero, la White-box security si chiede se è possibile confondere talmente tanto le cose, per cui l’attaccante che ha accesso White-box al codice, è come se fosse accesso Black-box al codice e quindi l’accesso al codice non è per nulla informativo per l’attaccante? L’obiettivo, quindi, della White-box security è di rendere una White-box una Black-box. Quindi, lo scenario di attacco in cui assumiamo di trovarci quando parliamo di offuscamento, è uno scenario associato alla White-box crypto,

ovverosia è lo scenario Man-at-the-end, in cui l'attacco può essere interpretato da un utente non fidato, il quale ha pieno pieno controllo del codice e del sistema e il nostro obiettivo è quello di rendere difficilmente leggibile e comprensibile il codice ad un utente, che può osservare il codice in ogni suo dettaglio.

Ci chiediamo a questo punto: "**Che cos'è l'offuscamento?**" Un offuscatore è un algoritmo O, tale che per un qualsiasi programma P, O(P) è un programma tale che:

- O(P) ha la stessa funzionalità di P;
- O(P) sia impossibile da analizzare, ovvero su di esso non sia possibile fare il reverse engineering. Questo fatto di essere impossibile da analizzare, può essere formalizzato dal concetto della **virtual black-box property**, ovvero: una trasformazione è un offuscamento, se l'attaccante che ha accesso al programma offuscato (ovverosia O(P)) può imparare dall'analisi di O(P) tutto e solo quello che si può imparare avendo accesso black-box a P, ovvero avendo accesso all'input e all'output di P → quindi, avere accesso a O(P) equivale ad avere accesso black-box a P, ovvero il programma è stato talmente tanto confuso, che avere accesso al codice è come avere accesso black-box a quest'ultimo.

Nel 1997 viene dimostrato che è possibile offuscare un particolare tipo di funzione, ovverosia le **Point Functions** → esse sono delle funzioni particolari, in quanto vanno ad 1 solamente in un punto. Quindi, le Point Functions valgono sempre zero ed in punto particolare valgono 1. Ad esempio, la funzione "Ix(w)" è una funzione, che ritorna 1 solamente quando gli viene passato in input "x", mentre in tutti gli altri casi ritorna zero.

$$I_x(w) = \{1 \text{ if } w = x, 0 \text{ otherwise}\}$$

É stato dimostrato, che è possibile offuscare le Point Functions, se si utilizza una **One-way function** → le funzioni One-way sono delle funzioni utilizzate in crittografia, che hanno una caratteristica particolare, ovverosia che sono funzioni:

- **facili da calcolare**;
- **impossibili (o comunque molto difficili) da invertire**.

e quindi se ho a disposizione l'output della funzione One-way, è impossibile (o comunque molto complicato) sapere quale input devo utilizzare, per eseguire la funzione ed ottenere nuovamente l'output che ho a disposizione.

Sorge allora spontanea la domanda: "**Come posso utilizzare le One-way function per offuscare le Point function?**"Chiaramente se vedo il codice della Point

Function capisco immediatamente il suo funzionamento, dal momento che vedo se “w = x” la funzione ritorna 1, e allora vediamo il suo offuscamento tramite una funzione One-way:

$$\mathrm{ObfI}_x(w) = \{\text{if } y = f(w) \text{ return 1 else return 0}\}$$

ovvero prende in input “w” e ritorna 1 in alcuni casi e in tutti gli altri casi zero. In particolare, ritorna 1 se “y = f(w)”, dove “y = f(x)”. Ricapitolando, abbiamo:

- f → è una One-way function, ovvero una funzione facile da calcolare e difficile da invertire;
- x → è il valore “particolare” per cui la nostra Point function va a 1;
- y → è l’output della One-way function (ovverosia “f”) calcolato su “x”. Quindi, “y” è l’output di f(x). Da notare il fatto, che da “y” non siamo in grado di ricavare “x”, dal momento che “f” è una One-way function.

Dal momento che “f” è una One-way function, se l’attaccante conosce “y” non ha alcuna informazione su “x” ed è proprio qui che l’idea dell’offuscamento, ovvero: non utilizzo più la condizione “ se w = x”, bensì utilizzo la condizione “se y = f(x) = f(w) ritorna 1, altrimenti va a zero” e noi sappiamo, che f(w) = y quando w = x → l’attaccante, quindi può solamente imparare, che l’output di f(x) = y, ma non riesce a scoprire il segreto “x”, che manda la Point function a 1 → **questo comporta, che l’attaccante ha un accesso black box alla Point function, ovvero fa sembrare all’attaccante che la Point function si comporti una costante settata a zero, in quanto l’attaccante deve provare tutti i possibili input per capire il comportamento della Point function, la quale va a 1 solamente in un punto e in tutti gli altri va a zero**.

Verrebbe da chiedersi, se è possibili sfruttare questo meccanismo di Virtual black-box anche per le altre tipologie di funzioni e la risposta a questa è NO, ovvero questo meccanismo di virtual black-box funziona solamente per le Point Function.

Diamo ora una definizione di offuscamento: l’offuscamento è una trasformazioni di programmi, che deve godere di tre proprietà:

1. **Preservare la funzionalità** → il programma originale e il programma offuscato devono avere lo stesso input e lo stesso output;

2. **Si ammette uno slow-down** (ovverosia una perdita di performance), dal programma originale al programma offuscato, **al più polinomiale** sia in tempo sia in spazio;
3. **Virtual black-box property** → ovverosia che ogni cosa, che l'attaccante può imparare dal programma offuscato, può essere dedotta avendo accesso black-box al programma originale. Quindi, tutto ciò che si può imparare dal programma offuscato, lo si può imparare avendo accesso black-box al programma originale → di fatto, stiamo dicendo che l'offuscamento trasforma il programma in una black-box. Questo lo si può tradurre, in modo più formale, nel seguente modo:

$$\left| Pr[A(O(M)) = 1] - Pr[S^{M}(1^{|M|}) = 1] \right| \leq \alpha(|M|)$$

Oracle access

dove:

- A → attaccante;
- M → Touring Machine (dato che tutte le dimostrazioni sono state fatte su Macchine di Touring e non su un programma).

e vuol dire che: La probabilità, che l'osservazione da parte dell'attaccante della macchina di Touring offuscata, sia 1 (ovverosia, la probabilità che l'attaccante impari qualcosa dal programma offuscati) deve essere più o meno uguale alla probabilità, che una macchina che ha accesso oracolo (ovverosia che ha accesso solamente input/output) al programma originale, sia 1 → un altro modo per dire ciò è: Quello che l'attaccante può imparare dal programma offuscato deve essere più o meno uguale a quello, che una macchina può imparare avendo accesso input/output (ovverosia accesso oracolo) al programma originale.

> 💡 Chiaramente è impossibile avere un offuscamento perfetto, ovverosia è impossibile nascondere ogni proprietà di ogni programma, bisogna capire cosa è possibile nascondere per ogni programma.

Abbiamo detto, che nell'offuscamento rappresentiamo i programmi come Macchine di Turing e in particolare **possiamo rappresentare i programmi come Macchine di Turing polinomiali e probabilistiche**, ovvero: Ad ogni passo, la macchina può prendere diverse decisioni (=diversi cammini) e ad

ognuna di queste decisioni viene assegnata una probabilità. Quindi, dal momento in cui siamo in grado di associare una probabilità ad ogni computazione e visto che ogni computazione porta ad uno stato finale accettante oppure ad uno stato finale di rifiuto, siamo conseguentemente in grado di determinare la probabilità di accettazione o di rifiuto. Adesso che sappiamo quali macchine di Turing utilizzare, possiamo dare una definizione di offuscamento per le macchine di Turing: L'offuscamento è una funzione, che trasforma un programma **M** in un programma **O(M)**, i quali hanno:

- la **stessa funzionalità**, ovvero la macchina M e la macchina O(M) devono calcolare la stessa identica cosa;
- lo **slowdown in tempo e in spazio deve essere al più polinomiale**;
- deve valere la **Virtual black-box property**, ovvero quello che posso imparare dalla macchina offuscata equivale a ciò che posso imparare avendo accesso oracolo alla macchina originale.

→ **quindi, una macchina O(M) offusca una macchina M, quando valgono queste tre proprietà appena sopra citate**.

La definizione di offuscamento, poi, è stata estesa a due macchine di Turing. In particolare, la definizione rimane invariata, ovverosia:

- deve essere mantenuta la funzionalità di ciascuna macchina;
- lo slowdown in tempo e in spazio deve essere al più polinomiale;
- la Virtual black-box property deve essere definita avendo accesso alle due macchine, ovvero: l'attaccante che ha accesso alle due macchine offuscate, può imparare tutto quello che può imparare avendo accesso oracolo alle due macchine originali.

L'essenza di questa prova è che esiste una differenza fondamentale tra ottenere l'accesso black-box a una funzione rispetto che ottenere l'accesso al programma che la calcola, non importa quanto sia offuscato. Ovvero, avere accesso al codice per quanto esso sia offuscato, è sempre più informativo rispetto che non averne accesso. Naturalmente, se la funzione è esattamente apprendibile tramite interrogazioni all'oracolo, questa differenza scompare. Vediamo meglio questo concetto attraverso un esempio:

$$\alpha, \beta \in \{0,1\}^k \quad \text{Secret}$$

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$$

$$D_{\alpha,\beta}(C) = \begin{cases} 1 & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

Abbiamo la macchina C è una macchina che assomiglia ad una Point Function, in quanto ritorna beta solamente quando “x = alpha” e in tutti gli altri casi va a zero e quindi è una macchina che ritorna un valore diverso da zero solamente in un punto. La macchina D, invece, prende in ingresso un’altra macchina (nel nostro caso C) e ritorna 1 se la macchina presa in input (ovvero C) ritorna come valore beta e in tutti gli altri casi ritorna zero.

Consideriamo, inoltre, un attaccante che date due macchine di Turing ne esegue una sull’altra, ovvero esegue D su C. Chiaramente, se si esegue D su C si otterrà come risultato sempre 1 e ci chiediamo: “**Cosa succede se l’attaccante esegue l’offuscamento della macchina D sull’offuscamento della macchina C?**” Per definizione l’offuscamento preserva la funzionalità e quindi se prendo una macchina offuscata D e la eseguo su una macchina offuscata C, il risultato dovrà essere sempre 1 (come nel caso delle macchine non offuscate).

Adesso consideriamo anche una macchina di Turing Z, che fornisce sempre come risultato zero e ho un attaccante che ha accesso oracolo a C e a D, ovvero deve capirne il loro funzionamento attraverso l’input e l’output e quindi deve capirne il funzionamento interrogandole. Quello che succede, è che l’attaccante, che ha accesso oracolo, fa fatica a distinguere la macchina C dalla macchina Z, o meglio dire che l’attaccante ha una bassa probabilità di distinguere la macchina C dalla macchina Z, dato che Z restituisce sempre zero, mentre C restituisce sempre zero tranne che in un punto → questo ci fa capire, che avere accesso oracolo è diverso rispetto ad avere accesso al codice, in quanto:

- se l’attaccante ha accesso al codice, se esso esegue D su C dirà che il risultato è 1;
- se, invece, l’attaccante ha accesso oracolo, allora se esegue D su C dirà che il risultato è zero, in quanto la macchina C è come se fosse la macchina Z per l’attaccante, dato che se l’attaccante:
    - interroga Z, quest’ultima gli fornisce sempre come risultato zero;

    - interroga C, quest'ultima gli fornisce sempre zero ad eccezione di un punto.

> 💡 **La Virtual black-box property non viene quindi rispettata**.

Questo dimostra, quindi, che è impossibile avere un offuscamento perfetto e l'implicazione di questa impossibilità è:

- non può esistere un **offuscatore ideale**, ovvero è impossibile rendere completamente intellegibile il codice, se non per il suo input e output, con uno slowdown al più polinomiale. É importante capire, che questo non implica il fatto che non si possa offuscare il codice. Infatti nella realtà quello che accade, è che si accettano offuscamenti più che polinomiali e che interessa nascondere determinate porzioni di codice a determinate tipologie di attaccanti.

Concludiamo, dicendo che nel 2001 è stata introdotta una quarta proprietà dell'offuscamento, ovvero l'**indistinguishability obfuscator** → essa stabilisce che: per ogni coppia di programmi che calcolano la stessa funzione e una volta che li offusco, la funzione offuscata non mi permette di capire qual era il programma originale da cui siamo partiti, ovvero: se abbiamo i due programmi C1 e C2, i quali calcolano esattamente la stessa funzione, l'offuscamento di C1 e l'offuscamento di C2 appaiono equivalenti all'attaccante, ovverosia l'attaccante non è in grado di capire se il codice che sta analizzando proviene da C1 oppure da C2.

Semantics-based Obfuscation

Quando si parla di offuscamento dobbiamo sempre capire:

- che cosa viene nascosto;
- rispetto a quale attaccante.

In particolare, l'attaccante lo modelliamo come un'analisi, la quale a sua volta può essere:

- analisi **statica** → essa prende il codice e lo va a decorare con delle proprietà invarianti, che va a riconoscere sempre come vere in determinati punti del codice → l'analisi statica, quindi, mi permette di capire alcune informazioni del codice, che sono importanti per:

- la comprensione del codice stesso;
- il debugging;
- la codifica.

Quindi, se ci immaginiamo l'attaccante come un analizzatore statico, l'offuscamento riesce a confondere quest'ultimo, nel momento in cui va a degradare la comprensione che l'analizzatore ha del programma, andando per esempio ad inserire dei predicati opachi e di conseguenza inserendo delle finte dipendenze tra le variabili. Vediamo un esempio:

```
Original()
    int c, nl = 0, nw = 0, nc = 0, in;
    in = false;

    while (( c = getchar()) != EOF){
        nc++;

        if(c == ' ' || c = '\n' || c = '\t') in = false;

        elseif(in  == false) {in = true; nw++}

        if(c == '\n') nl++;

        }

    out(nl,nw,nc);}
```

```
Obfuscated()
    int c, nl = 0, nw = 0, nc = 0, in;
    in = false;

    while (( c = getchar()) != EOF){
        nc++;

        if(c == ' ' || c = '\n' || c = '\t') in = false;

        elseif(in  == false) {in = true; nw++}

        if(c == '\n') {if(nw <= nc) nl++};

        if(nl > nc) nw = nc + nl;

        elseif(nw > nc) nc = nw – nl;

        }
    out(nl,nw,nc);}
```

	c	nc	nw	nl
c				
nc	X	X		
nw	X		X	
nl	X			X

	c	nc	nw	nl
c				
nc	X	X	X	X
nw	X	X	X	X
nl	X	X	X	X

Vediamo, che nel programma originale abbiamo che:

- la variabile "nc" (= new character) dipende dalla variabile "c" (= carattere corrente) e dalla variabile "nc" stessa:
- la variabile "nw" (= new word) dipende sempre dalla variabile "c" e da se stessa;
- infine la variabile "nl" (= new line) dipende sempre dalla variabile "c" e da se stessa.

Il programma offuscato, invece, va ad inserire tre predicati opachi:

- if(nw <= nc) → il quale è sempre vero;
- if(nl > nc) →> il quale è sempre falso;
- elseif(nw > nc) → il quale è sempre falso.

Lo scopo di questi predicati opachi è di inserire delle false dipendenze, in modo tale da complicare l'analisi all'analizzatore statico. Come si può vedere dalla tabella, quindi, grazie a questi tre predicati opachi per l'analizzatore statico il numero di dipendenze è aumentato, ovvero per l'analizzatore ogni variabile dipende da tutte le altre variabili del programma → in questo modo viene confuso l'analizzatore statico, dato che alle dipendenze reali viene aggiunto del rumore, ovverosia vengono aggiunte delle false dipendenze e ciò porta ad una minore comprensione del codice.

L'analisi statica, quindi, corrisponde ad un'esecuzione astratta del programma, dove il dominio astratto è quello che rappresenta la proprietà di interesse per l'attaccante, dove per proprietà intendiamo, ad esempio:

- le dipendenze;
- input/output;
- il valore massimo che una variabile può assumere durante l'esecuzione.

Tali proprietà possono essere ordinate (in modo parziale), ottenendo quindi un reticolo, dove possiamo ottenere:

- proprietà tra loro non confrontabili;
- alcune proprietà più forti di altre.

Quindi, avremo che la proprietà che osserva l'attaccante è tanto più forte quanto più è precisa, ovvero: l'attaccante che ha accesso a tutto (ovvero che può vedere il valore in memoria di tutte le variabili) è più forte rispetto ad un attaccante che ha accesso parziale (ovvero che può vedere solamente l'input e l'output).

- analisi **dinamica** → corrisponde ad un'osservazione astratta di un'esecuzione concreta, ovvero considera solamente gli aspetti certi di un'esecuzione certa.

Il nostro obiettivo è di interpretare formalmente il mondo dell'offuscamento andando a studiare gli effetti dell'offuscamento sulla **semantica** del programma. Quindi abbiamo:

- la semantica → è l'insieme delle proprietà delle varie tracce di esecuzione del programma;
- un attaccante → il quale è interessante a scoprire alcune cose delle tracce di esecuzione del programma, in base ai suoi obiettivi;

- l’offuscamento → il quale va ad aggiungere rumore all’interno delle tracce di esecuzione, rispettando però delle invariati, come ad esempio preservare l’input e l’output. In particolar modo, ciò che mi dice il comportamento offuscante di una trasformazione di programmi, è ciò che quella trasformazione preserva o non preserva della semantica del programma. Per comprendere meglio questo ultimo concetto, immaginiamo di trasformare un programma in un altro programma che fa la stessa cosa, tutto quello che preservo del programma originale è quello che l’attaccante può ancora conoscere. Quindi, se riusciamo a caratterizzare qual è la più concreta proprietà che viene preservata dall’offuscamento, stiamo di fatto andando a porre il limite di quello che posso ancora conoscere dopo l’offuscamento e quello che non posso più conoscere → tutto ciò che preservo lo conosco anche dopo l’offuscamento, mentre tutto quello che non preservo non lo conosco più dopo l’offuscamento.

> 💡 Quindi, tutte le proprietà semantiche preservate dall’offuscamento possono ancora essere conosciute, mentre quelle che non vengono preservate non possono essere conosciute.

Adesso che abbiamo capito i concetti sopra, passiamo a comprendere l’idea alla base dell’offuscamento della semantica: prima abbiamo detto, che l’offuscamento deve preservare l’input e l’output, ma l’input/output è una particolare proprietà della semantica e noi vorremmo preservare anche di meno dell’input/output, ovvero vorremmo preservare solamente alcuni aspetti dell’input/output e non tutti i dettagli. Quindi, l’offuscamento preserva qualcosa ed in particolare, caratterizziamo l’offuscamento in base alla più concreta proprietà che preserva → **la più concreta proprietà semantica preservata ci dà il limite di quello che è e quello che non è preservato** (in quanto una proprietà meno precisa di quella limite è preservata). **L’offuscamento, quindi, è una trasformazione sintattica di programmi e protegge tutto quello che non viene preservato della semantica** ed essendo proprietà semantiche, quest’ultime le posso ordinare nel reticolo e a questo punto posso dimostrare formalmente, se un offuscamento sconfigge oppure no un attaccante.

In questa visione, è interessante il fatto che ogni trasformazione di codice può essere vista come un offuscamento, in quanto ogni volta che si trasforma viene perso qualcosa (a livello di semantica) e viene preservato qualcosa (a livello di semantica) e naturalmente tutto quello che si perde è quello che si

protegge, dato che quello che si perse è quello che non si può più conoscere del programma originale rispetto al programma offuscato.