UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Tesi di Laurea Triennale in Ingegneria Informatica

Analisi su vulnerabilità del generatore di numeri pseudo-casuali su curve ellittiche Dual EC-DRBG

Relatori:

Prof. Giuseppe Lettieri

Dott. Gaspare Ferraro

Candidato:

Alex Parri

ANNO ACCADEMICO 2022/2023

Indice

Capitolo 1

Motivazione

La crittografia è una disciplina assai antica e studiata approfonditamente da numerosi anni e ultimamente sta ricoprendo un ruolo sempre più d'importanza non solo per scopi militari, motivazione per cui è stata creata originalmente, ma sta anche agendo nella vita di ogni normale cittadino, seppur in maniera completamente trasparente al fruitore: senza di essa, ad oggi, nulla sarebbe più segreto.

In un mondo ideale la crittografia risulterebbe completamente inutile, un mondo in cui nessuno prova a carpire informazioni sul prossimo, ma si può ben intuire che questa di cui si sta parlando è un utopia: vi sarà sempre un soggetto malintenzionato che proverà ad usare tutti gli strumenti che ha a disposizione per recare danno al prossimo, o comunquesia per trarre vantaggio a sé stessi.

È uno strumento molto potente, ma delicato: come ogni strumento, va saputo utilizzare, altrimenti si potrebbe recare danno a sé stessi o anche al prossimo, è quindi di vitale importanza essere consapevoli di che cosa si stia maneggiando e come usarlo, prima di poter prendere qualunque decisione, è quindi un'arma a doppio taglio, come si suol dire.

Basti pensare che esistono cifrari *dichiarati sicuri*, cioè tali per cui non sappiamo dell'esistenza di algoritmi che siano in grado di attaccarli in tempo ragionevole, che però vengono *forzati* per altri mezzi, ad esempio applicando attacchi sulla piattaforma in cui sono implementati anziché passare per il cifrario stesso.

L'obiettivo di questa tesi è di discutere del generatore di numeri pseudo-casuali Dual EC-DRBG, generatore *crittograficamente sicuro*, che è divenuto ufficialmente standard tramite una pubblicazione del NIST[1] nel 2006 nonostante numerosi articoli anche da parte di esperti del settore abbiano messo alla luce dubbi sulla sua legittimità.

Innanzitutto faremo una introduzione sintetica alla crittografia, per poi passare alle curve ellittiche e ai generatori di numeri pseudo-casuali in generale, alla famiglia dei crittograficamente sicuri, per poi arrivare al generatore in questione, verrà infine dimostrato l'attacco *Shumow-Ferguson* attraverso il quale sarà possibile, supposta l'esistenza di una relazione tra due particolari parametri impiegati nel generatore,

[1]National Institute of Standards and Technology (https://www.nist.gov/)

risalire al suo stato interno, con annessa Proof of Concept (PoC) scritta in linguaggio Python.

Per concludere verranno effettuati dei test di benchmark della PoC con curve ellittiche P-192, P-224, P-256, P-384 e P-521 per dare una osservazione critica sul pur aumentando la dimensione dei dati, anche una macchina più o meno potente disponibile ad un cittadino qualunque sia in grado di effettuare correttamente l'attacco in brevissimo tempo.

Capitolo 2

Crittografia

In termini precisi, la crittografia è quella tecnica di rappresentazione di un messaggio in una forma tale che l'informazione in esso contenuta possa essere recepita solo da destinatari autorizzati: il suo scopo, quindi, è quello di garantire *confidenzialità* [18].

Quest'ultima si può ottenere tramite *algoritmi crittografici* che constano tutti di due fasi principali:

- *cifratura*: viene applicata una funzione matematica C che altera una stringa di dati in input m, messaggio in chiaro, in una stringa di dati di output c, crittogramma
$$C(m) = c \tag{2.1}$$
in questo caso si dice di aver eseguito la cifratura del messaggio in chiaro m.

- *decifratura*: viene applicata una funzione matematica D che altera il crittogramma c in maniera tale da riottenere perfettamente il messaggio in chiaro m.
$$D(c) = m \tag{2.2}$$
in questo caso si dice di aver eseguito la decifratura del crittogramma c.

Affinché la fase di decifratura vada sempre a buon fine, si deve avere che

$$D(C(m)) = m \tag{2.3}$$

ovvero che la funzione C sia invertibile e sia invertita dalla funzione D, quindi all'effettivo deve essere tale per cui

$$D = C^{-1} \longrightarrow C^{-1}(C(m)) = m \tag{2.4}$$

se questa condizione non è rispettata, la cifratura non è iniettiva e può quindi accadere in alcuni casi di non riuscire a ritornare al messaggio originale m una volta cifrato nel crittogramma c.

In questa maniera chiunque potrebbe cifrare e decifrare qualunque messaggio: affinché la crittografia possa garantire confidenzialità, e quindi riuscire a nascondere

un messaggio a chiunque non si voglia che ne venga a conoscenza, si deve introdurre un elemento segreto, la chiave k, che può avere molteplici forme: un singolo carattere, una stringa, una pagina di un libro, o altro, l'importante è che sia mantenuta segreta solo dalle parti che devono riuscire a risalire al messaggio in chiaro m.

In base alla metodologia in cui la chiave k è combinata con le funzioni C e D, si può suddividere i cifrari in due famiglie principali:

- cifrari a chiave simmetrica, detti anche *a chiave privata*
- cifrari a chiave asimmetrica, detti anche *a chiave pubblica*

In entrambi i casi, il modo in cui sono composte le funzioni in questione può essere molto intricato come banale, e col passare degli anni si sono sempre più complicate, questo dovuto dal fatto che la crittografia ha un ruolo sempre più rilevante nella vita di tutti i giorni, soprattutto dopo l'avvento di internet.

Considerata questa importanza, si ha la necessità di realizzare cifrari sicuri: affinché un cifrario si possa ritenere tale, esso deve riuscir a nascondere l'informazione cifrata per un periodo di tempo considerato *molto lungo*, ovvero per un tempo sufficiente a rendere l'informazione letta da una sua eventuale forzatura obsoleta.

In particolare, è importante che le fasi di cifratura e decifratura C, D siano:

- *computazionalmente facili* per chi possiede la chiave k, ovvero si possano calcolare in tempo *ragionevole*, alla peggio in alcuni minuti
- *computazionalmente difficile* risalire al messaggio in chiaro m senza conoscere k, seppur conoscendo le funzioni C, D, ovvero che richieda una quantità di tempo almeno nell'ordine di un paio di anni

per rispettare queste condizioni, si fanno uso di funzioni $f(x)$ definite come segue

Definizione 1 (Funzione one-way trap-door)**.** *Una funzione $f(x)$ one-way trap-door è una funzione computazionalmente facile da calcolare dato un input x*

$$y = f(x) \tag{2.5}$$

ma computazionalmente difficile da invertire conoscendo solo l'immagine y

$$x = f^{-1}(y) \tag{2.6}$$

a meno che non si conosca una informazione speciale, chiamata trap-door k

$$x = g(y, k) \tag{2.7}$$

Per scopi crittografici moderni, le funzioni one-way trap-door impiegate sono accuratamente scelte in modo tale che una loro eventuale forzatura sia possibile solo andando ad operare su un insieme di campioni molto elevato e andando ad interagire con numeri di centinaia o anche migliaia di cifre binarie: se implementati correttamente, l'inversione risulta praticamente impossibile anche per le macchine più potenti al mondo.

2.1 Cifrari a chiave simmetrica

In questa famiglia di cifrari, la chiave k, che prenderà il nome di chiave *simmetrica*, deve essere condivisa dalle due parti, ovvero che sia la stessa usata sia per cifrare che per decifrare

$$c = C(m, k) \quad m = D(c, k) \tag{2.8}$$

In altre parole, le funzioni C, D e la chiave k devono essere tali che

$$m = D(C(m, k), k) \tag{2.9}$$

Il problema in questo approccio è evidente: se i due utenti non possono incontrarsi, è necessario che la chiave venga scambiata nel canale insicuro, cioè prima che l'algoritmo crittografico sia entrato in azione, questo problema si può risolvere in più modi:

1. implementando un cifrario a chiave asimmetrica per cifrare la chiave k
2. utilizzando un algoritmo di costruzione della chiave, il più comune è l'algoritmo *Diffie-Hellmann*
3. scambiandosi la chiave nella vita reale, che per applicazioni di massa risulta impraticabile.

I cifrari a chiave simmetrica svolgono operazioni semplici sui dati in input che ricevono, come permutazioni, shift, somme bit a bit modulo 2 (xor) con aggiunta di funzioni non lineari per aumentarne la sicurezza.

2.2 Cifrari a chiave asimmetrica

In questo caso ogni utente i possiede una coppia di chiavi:

$$i \longrightarrow (k^i_{pub}, k^i_{priv})$$

- una chiave pubblica k^i_{pub} che può essere usata da un qualsiasi mittente per cifrare messaggi diretti al destinatario i

  $$c = C(m, k^i_{pub}) \tag{2.10}$$

  per questo motivo, questa famiglia di cifrari sono anche detti *a chiave pubblica*

- una chiave privata k^i_{priv} che verrà usata dal destinatario i per decifrare tutti i messaggi costruiti usando la rispettiva chiave pubblica

  $$m = D(c, k^i_{priv}) \tag{2.11}$$

Siccome la chiave di cifratura k^i_{pub} è pubblica per definizione, chiunque può inviare messaggi cifrati al destinatario i: la sicurezza sta però nel fatto che solo i può decifrare i messaggi rivolti verso lui, poichè $C, D, k^i_{priv}, k^i_{pub}$ sono tali per cui

$$m = D(C(m, k^i_{pub}), k^i_{priv}) \tag{2.12}$$

ovvero che k^i_{priv} è l'unico elemento che attraverso la funzione D riesce ad invertire la trasformazione applicata con C tramite k^i_{pub} ad m.

La coppia (k^i_{pub}, k^i_{priv}) dovrà essere generata in maniera appropriata per ogni destinatario i, in particolare:

- devono essere computazionalmente facili da generare
- devono essere generate nel modo più imprevedibile e casuale possibile
- deve risultare praticamente impossibile per due utenti scegliere la stessa coppia di chiavi
- dovranno essere tali per cui vale 2.12

Questi tipi di cifrari risolvono il problema dello scambio semplicemente non richiedendo lo scambio, in quanto la chiave privata generata non è mai scambiata o ceduta, ma è posseduta *in primis* da ogni destinatario.

Un altro grande vantaggio che offrono è il fatto che, in un ipotetico sistema con N utenti ove ogni utente vuole comunicare con ogni altro, implementando un cifrario

- a chiave simmetrica: si necessiterebbe di una chiave simmetrica per ogni comunicazione, quindi di un numero di chiavi pari a

  $$\frac{N(N-1)}{2} = \frac{1}{2}\left(N^2 - N\right) \longrightarrow O(N^2)$$

  che quindi cresce quadraticamente nel numero degli utenti, senza contare il problema dello scambio per ogni coppia di utenti

- a chiave pubblica: è sufficiente che ogni utente generi la propria coppia di chiavi, quindi si necessitano di un numero di chiavi pari a

  $$2N \longrightarrow O(N)$$

  che cresce linearmente nel numero degli utenti, ove le rispettive chiavi private non devono essere scambiate, è quindi una soluzione che scala in maniera molto più efficiente.

Nonostante questi vantaggi, la crittografia a chiave pubblica non ha rimpiazzato quella a chiave privata ma convive insieme ad essa, questo poiché quella asimmetrica:

- è computazionalmente più pesante di quella a chiave privata

- necessita di chiavi più molto lunghe a parità di sicurezza
- è soggetta (per sua natura) ad attacchi di tipo *chosen plain-text*, dato dal fatto che chiunque può inviare messaggi cifrati al destinatario
- è vulnerabile alla computazione quantistica
- è vulnerabile ad attacchi di tipo *man-in-the-middle*

Negli approcci più moderni si preferisce implementare un sistema a *cifrario ibrido*, ove si impiega:

- un cifrario a chiave pubblica per lo scambio di una chiave simmetrica k_S
- un cifrario a chiave privata per il resto della comunicazione, che utilizza la chiave simmetrica scambiata k_S, ora disponibile a mittente e destinatario

2.2.1 L'algoritmo RSA

L'algoritmo crittografico RSA (Rivest-Shamir-Adleman, 1977) è lo standard per quanto riguarda i cifrari a chiave asimmetrica [8].

Ogni utente che lo implementa effettuerà le seguenti operazioni preliminari per la generazione della propria coppia di chiavi pubblica e privata:

- genera due interi primi casuali p, q molto elevati e sufficientemente diversi tra loro di almeno 1024 cifre binarie l'uno [3]
- calcola il numero $n = p \cdot q$
- calcola il *toziente*[1] del numero n, ovvero il valore $\phi(n) = (p-1)(q-1)$
- genera un numero intero primo casuale e in modo tale che sia minore di $\phi(n)$ e coprimo ad esso, cioè con cui non condivida fattori a comune
- calcola il numero $d = e^{-1} \bmod \phi(n)$, ove e^{-1} è l'inverso moltiplicativo di e modulo $\phi(n)$, cioè è tale che:

$$e \cdot e^{-1} \equiv 1 \bmod \phi(n) \tag{2.13}$$

  che è possibile calcolare in tempo polinomiale tramite l'*algoritmo di Euclide Esteso*.

Le chiavi pubbliche e private sono:

$$\begin{cases} k^i_{pub} = (n, e) \\ k^i_{priv} = d \end{cases}$$

[1] Il *toziente* di un numero intero n detto anche *funzione di Eulero* indica il numero di elementi interi minori di n e coprimi con n

mentre le operazioni di cifratura e decifratura sono:

$$\text{cifratura: } c = m^e \bmod n$$
$$\text{decifratura: } m = c^d \bmod n$$

ove m e c sono rappresentazioni binarie del messaggio in chiaro e del crittogramma rispettivamente, inoltre si richiede che m sia tale per cui:

$$m^e < n \tag{2.14}$$

altrimenti interverrebbe la riduzione in modulo e la cifratura non sarebbe più iniettiva.

Risulta particolarmente importante che anche i parametri $p, q, \phi(n)$ siano mantenuti segreti, in quanto è sufficiente la conoscenza di solo uno di essi per poter risalire a d.

Infatti, conoscendo p oppure q, siccome si ha che $n = p \cdot q$ allora si può ottenere banalmente l'altro

$$p = \frac{n}{q} \text{ oppure } q = \frac{n}{p} \tag{2.15}$$

a questo punto possiamo risalire a $\varphi(n)$ con una semplice moltiplicazione

$$\phi(n) = (p-1)(q-1) \tag{2.16}$$

e infine ottenere d tramite l'algoritmo di Euclide Esteso

$$d = e^{-1} \bmod \phi(n) \tag{2.17}$$

Andando però a considerare $p, q, \phi(n)$ non noti e p, q scelti opportunamente, siccome e è un elemento pubblico, l'unico modo per risalire alla chiave privata d è tramite $\phi(n)$

$$\phi(n) = f(p, q)$$

essendo funzione di p, q, l'unica scelta possibile è di andare deve ricavare questi ultimi fattorizzando n direttamente, il quale, se p, q sono scelti opportunamente, è una operazione ad oggi computazionalmente difficile.

La sicurezza dell'algoritmo, con parametri scelti opportunamente, è garantita quindi dalla difficoltà computazionale della fattorizzazione di numeri semiprimi molto elevati; tuttavia, è una difficoltà *presunta*: è vero che ad oggi è difficile, ma lo è perchè non sappiamo se esistano algoritmi efficienti in grado di risolverla, in quanto sia un problema della classe NP e non NP-Complete.

A peggiorare la situazione, in problema in questione non è più difficile quanto una volta, infatti per fattorizzare un numero semiprimo di n bit:

- tramite attacco forza bruta, ovvero il più basilare possibile, la complessità è pari a $O(2^n)$, quindi esponenziale

- tramite attacco GNFS (General Number Field Sieve), l'attacco più efficiente ad oggi, è pari a $O(2^{\sqrt{n \log n}})$, quindi subesponenziale

Non è affatto un risultato di poco conto: infatti comporta che a parità di sicurezza serviranno chiavi più lunghe, le operazioni di cifratura e decifratura saranno più computazionalmente pesanti, in quanto andranno ad operare su numeri di grandezze superiori.

Capitolo 3

Crittografia su curve ellittiche

La crittografia su curve ellittiche è un sistema alternativo alla crittografia a chiave pubblica, ideata per far fronte al problema del carico computazionale conseguente all'uso di un cifrario a chiave pubblica "tradizionale".

Per poterla trattare, è necessario dover fornire un paio di definizioni preliminari essenziali.

Definizione 2 (Campo)**.** *Un campo $\mathbb{K}$ è una struttura algebrica composta da un insieme di elementi su cui sono definite operazioni di somma, sottrazione, moltiplicazione, divisione e l'esistenza dell'elemento neutro additivo $(-a)$ e dell'elemento neutro moltiplicativo (a^{-1}) per ogni membro a che vi appartiene.*

Definizione 3 (Caratteristica di un campo)**.** *La caratteristica $char(\mathbb{K})$ di un campo $\mathbb{K}$ indica il minimo numero di volte p che si deve sommare l'elemento neutro moltiplicativo con sé stesso per ottenere l'elemento neutro additivo.*

$$\underbrace{1+1+1+...+1}_{p\ volte} = 0 \tag{3.1}$$

Se tale numero non esiste, è posto per definizione $p = 0$.

Definizione 4 (Curva ellittica)**.** *Una curva ellittica E su un campo $\mathbb{K}$ è l'insieme di punti $(x, y) \in \mathbb{K}^2 = \mathbb{K} \times \mathbb{K}$ che rispettano la seguente equazione*

$$y^2 + axy + by = x^3 + cx^2 + dx + e \qquad a, b, c, d, e \in \mathbb{K} \tag{3.2}$$

a cui tipicamente si va ad includere un punto O, detto punto all'infinito, che prenderà il ruolo di elemento neutro additivo.

Definizione 5 (Forma Normale di Weierstrass)**.** *L'equazione verificata dall'insieme di punti $(x, y) \in \mathbb{K}$ di una curva ellittica E si può semplificare nella seguente forma, detta Forma Normale di Weierstrass*

$$y^2 = x^3 + ax + b \qquad a, b \in \mathbb{K} \tag{3.3}$$

se e solo se la caratteristica del campo $\mathbb{K}$ su cui è definita la curva E è tale che

$$char(\mathbb{K}) \neq \{2, 3\}$$

Da adesso in poi andremo a considerare quest'ultima forma, in quanto più semplice per la nostra trattazione.

3.1 Curve ellittiche sui reali

In questo tipo di curve, si suppone che il campo $\mathbb{K}$ su cui sono definite le curve ellittiche sia l'insieme dei numeri reali $\mathbb{R}$

$$\mathbb{K} = \mathbb{R} \longrightarrow \begin{cases} a, b \in \mathbb{R} \\ (x, y) \in \mathbb{R}^2 \end{cases}$$

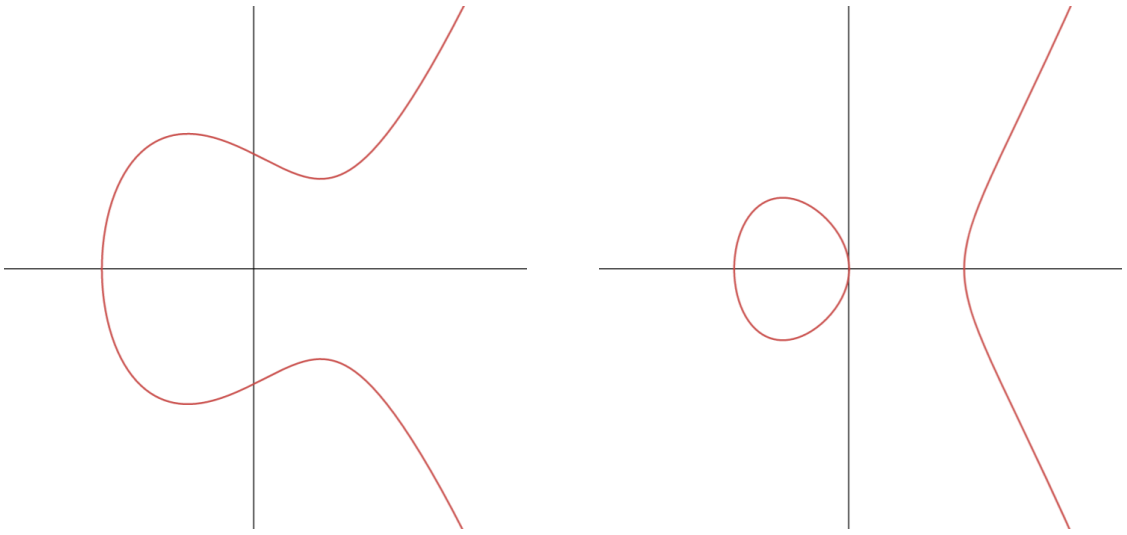
siccome quest'ultimo è un campo con caratteristica nulla

$$char(\mathbb{R}) = 0$$

in quanto non esiste un minimo numero di volte tale per cui si può sommare l'unità per ottenere lo zero, allora si ricade nella casistica per cui si può fare semplificare l'equazione dei punti nella Forma Normale di Weierstrass:

$$y^2 = x^3 + ax + b \qquad a, b \in \mathbb{R} \tag{3.4}$$

l'insieme di punti (x, y) che rispettano l'equazione 3.4 saranno facilmente rappresentabili nel piano cartesiano, ad esempio come in figura 3.1.



(a) $y^2 = x^3 - x + 1$ (b) $y^2 = x^3 - x$

Figura 3.1: Due esempi di curve ellittiche sui reali

Le curve ellittiche sui reali presentano tutte simmetria orizzontale, questo accade perché l'equazione che le caratterizza, come si può notare in 3.4, presenta un y^2 e quindi è risolta contemporaneamente sia da y che da $-y$, infatti andando a sostituire $y = \pm y$, si riottiene la Forma Normale di Weierstrass:

$$(\pm y)^2 = y^2 = x^3 + ax + b \tag{3.5}$$

per ogni punto P della curva sono definiti due valori dell'ascissa, uno opposto all'altro, a questo punto sarebbe più che corretto dare la definizione di punto opposto

Definizione 6 (Punto opposto)**.** *Dato un qualsiasi punto P appartenente alla curva ellittica sui reali E il punto opposto $(-P)$ è definito come come lo stesso punto con ordinata cambiata di segno*

$$P = (x, y) \longrightarrow -P = (x, -y) \tag{3.6}$$

che appartiene alla stessa curva E del punto P, in quanto ne verifica l'equazione, mentre per quanto riguarda il punto all'infinito O, il suo opposto è pari a sé stesso

$$-O = O \tag{3.7}$$

Da un punto di vista grafico, in una curva ellittica sui reali si può ottenere facilmente l'opposto $-P$ di un punto P andando a ribaltare il punto stesso rispetto all'asse x, come si può notare in figura 3.2.

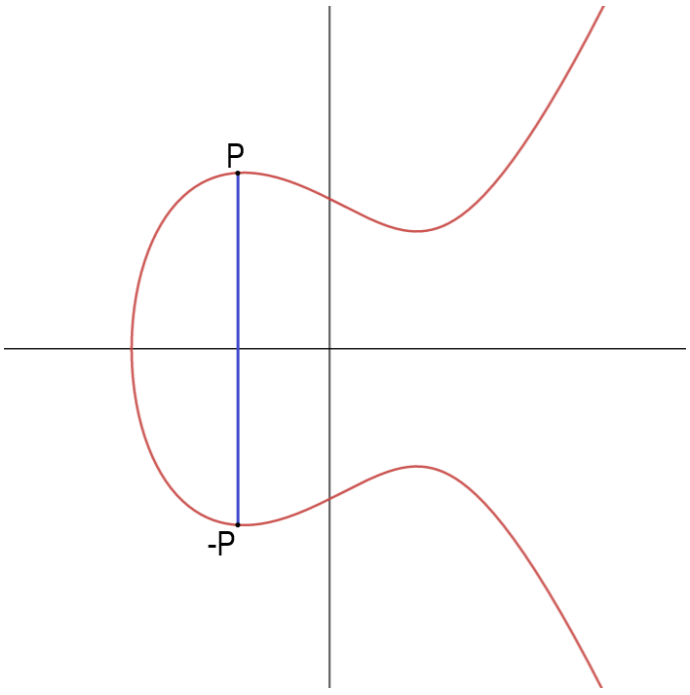


Figura 3.2: Punto opposto

A questo punto possiamo definire l'operazione di somma tra punti della curva, che risulterà molto utile ed efficace per fini crittografici: per poter ottenere ciò, è necessario che quest'ultima sia una *operazione interna*

Definizione 7 (Operazione interna)**.** *Una operazione interna di un insieme non vuoto X ad n argomenti è una funzione $*$ che associa ad una n-upla di valori di X un valore sempre appartenente ad X*

$$* : \underbrace{X \times X \times ... \times X}_{n \text{ volte}} = X^n \to X \tag{3.8}$$

in altre parole, la funzione $*$ *è tale che*

$$x_1, x_2, ..., x_n \in X \longrightarrow (x_1 * x_2 * ... * x_n) \in X \tag{3.9}$$

Vediamo come si può ottenere questo risultato, partendo da un punto di vista analitico: andando a risolvere l'intersezione tra una retta qualsiasi e l'equazione della curva in Forma Normale di Weierstrass

$$\begin{cases} y = mx + q \\ y^2 = x^3 + ax + b \end{cases}$$

si ottiene la seguente equazione:

$$x^3 + (-m^2)x^2 + (a - 2mq)x + (b - q^2) = 0 \tag{3.10}$$

le cui radici saranno i punti di intersezione: siccome è di terzo grado, consterà al più di 3 soluzioni: siccome le soluzioni complesse coniugate si presentano sempre a coppie di due, ci saranno due opzioni possibili per l'equazione 3.10

- 3 punti reali di intersezione
- 1 punto reale di intersezione e 2 complessi coniugati

andando a fare l'ipotesi che la curva sia *non singolare*, cioè che non ammetta nodi o cuspidi, ovvero che il suo discriminante Δ sia tale che

$$\Delta = 4a^3 + 27b^2 \neq 0 \tag{3.11}$$

allora è garantita l'esistenza e l'univocità dei punti di intersezione, ovvero delle radici di 3.10, e questo risultato si può notare anche graficamente:

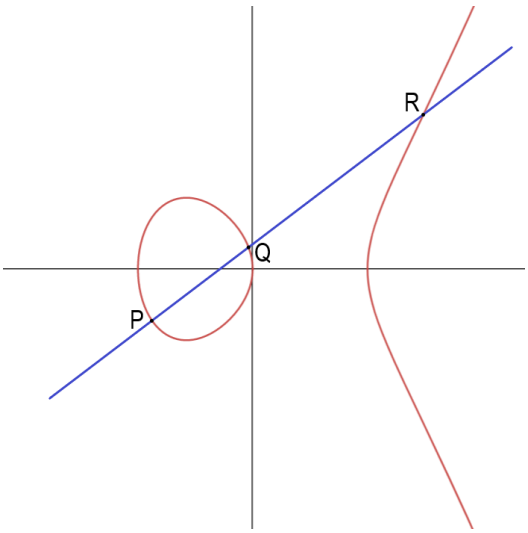


Figura 3.3: Intersezione tra curva ellittica e una retta

Come si può notare in figura 3.3, se la retta interseca due punti P, Q appartenenti alla curva, allora è garantita l'esistenza di un terzo punto di intersezione R, che appartiene sempre alla stessa curva: a questo punto, possiamo dare la definizione formale di somma tra punti della curva

Definizione 8 (Somma tra punti)**.** *Dati tre punti P, Q, R appartenenti ad una curva ellittica E definita su un campo $\mathbb{K}$, se i punti sono anche di intersezione e il discriminante della curva è diverso da zero, si definisce la seguente relazione*

$$P + Q + R = O \tag{3.12}$$

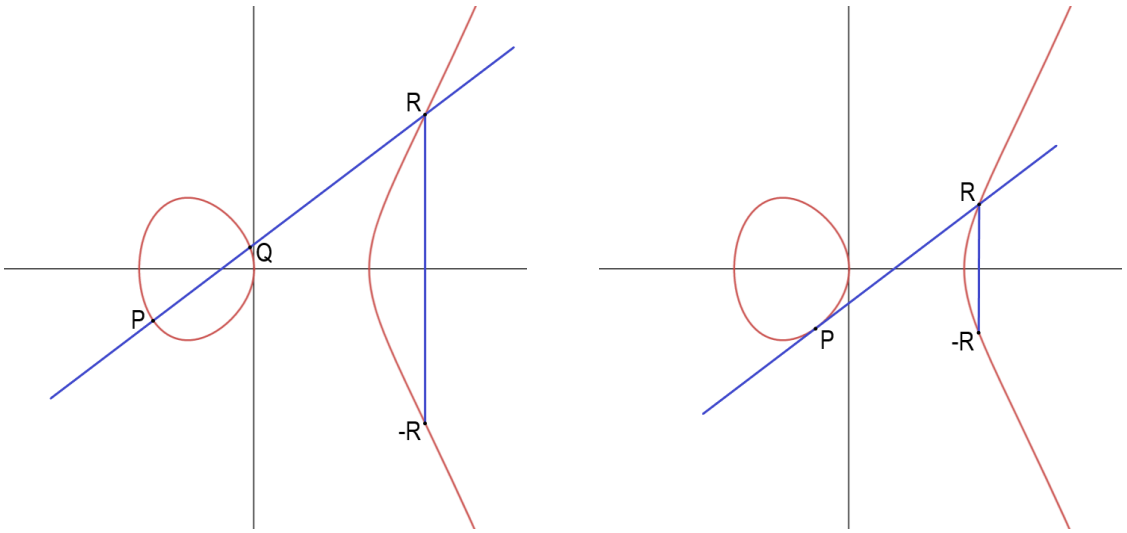
la somma tra punti sarà definita quindi come

$$P + Q = -R \tag{3.13}$$

ove O corrisponde al punto all'infinito.

Graficamente, il punto risultato della somma tra due punti P, Q nelle curve ellittiche sui reali si può ottenere nel seguente modo:

- se $P \neq Q$: si traccia la retta che interseca i due punti, l'esistenza di un terzo punto di intersezione alla retta è garantita dalla condizione 3.11, il quale verrà ribaltato rispetto all'asse x
- se $P = Q$: si traccia la tangente al punto P, siccome i due punti sono uguali si contano due volte, l'esistenza del terzo punto di intersezione è anche in questo caso garantita dalla condizione 3.11, il quale verrà ribaltato rispetto all'asse x



(a) $P \neq Q$ (b) $P = Q$

Figura 3.4: Somma tra punti

Analiticamente, la somma di punti è svolta tramite semplici operazioni algebriche tra le coordinate dei punti.

In conclusione, la somma tra punti appena definita è tale che

- se la retta interseca la curva in 2 punti, esisterà sicuramente un terzo punto di intersezione sempre appartenente alla curva, il cui opposto sarà il punto somma;
- se la retta interseca la curva in un solo punto, la retta sarà verticale e quindi il punto somma sarà il punto all'infinito O, che appartiene alla curva per definizione.

abbiamo appena dimostrato che la somma tra punti $(+)$ è quindi una operazione chiusa nelle curve ellittiche sui reali, cioè è tale che

$$+ : \mathbb{R}^2 \to \mathbb{R}^2 \tag{3.14}$$

poiché in ogni possibile scenario, con due punti P, Q della curva $E(a, b)$ in input si ottiene consistentemente un altro punto della curva $-R$ in output

$$P, Q \in E(a, b) \longrightarrow P + Q = -R \in E(a, b) \tag{3.15}$$

è dimostrabile che la proprietà 3.14 si può estendere qualsiasi campo $\mathbb{K}$, tuttavia sulle curve ellittiche sui reali si può dimostrare in maniera efficace.

3.2 Curve ellittiche su campi finiti

Nella crittografia moderna, per via della vasta mole di dati che deve essere trattata durante lo svolgimento di un algoritmo, l'elaborazione è delegata ai calcolatori, che però non sono molto abili a gestire numeri reali, soprattutto quando si devono trattare numeri trascendenti o irrazionali: più precisamente, non ci si può permettere durante l'applicazione di un algoritmo crittografico di mal interpretare un numero per via di errori di troncamento o arrotondamento.

Per questo motivo, la crittografia su curve ellittiche fa uso di curve definite su campi finiti, non sull'insieme dei numeri reali: questo tipo di curve sono meno intuitive ma più efficaci.

Per poterle trattare, prima di tutto dobbiamo dare la definizione di *campo finito*

Definizione 9 (Campo finito)**.** *Un campo finito $\mathbb{F}_{p^n}$ è un campo contenente un numero finito di elementi pari a p^n con p primo e n naturale, e caratteristica pari a p.*

Una curva ellittica su un campo finito sarà quindi sempre una curva ellittica, ma supposto che il campo $\mathbb{K}$ su cui è definita sia un campo finito $\mathbb{F}_{p^n}$

$$\mathbb{K} = \mathbb{F}_{p^n} \longrightarrow \begin{cases} a, b \in \mathbb{F}_{p^n} \\ (x, y) \in \mathbb{F}_{p^n}^2 \end{cases} \tag{3.16}$$

ove da ora in poi andremo a supporre $p > 3$ per poter semplificare l'equazione della curva nella Forma Normale di Weierstrass, come segue:

$$y^2 = x^3 + ax + b \qquad a, b \in \mathbb{F}_{p^n} \tag{3.17}$$

Successivamente, è importante specificare che tutti i campi finiti $\mathbb{F}_{p^n}$ sono *isomorfi* tra loro:

Definizione 10 (Isomorfismo)**.** *Due strutture algebriche sono isomorfi, ovvero esiste un isomorfismo tra le due strutture, se esiste una corrispondenza uno-ad-uno tra gli elementi delle due che conserva la struttura presente tra elementi stessi.*

All'effettivo si possono quindi si possono trattare in maniera praticamente equivalente: in particolare, se supponiamo di avere

$$n = 1 \tag{3.18}$$

e sfruttando il fatto che esiste un isomorfismo con l'insieme dei numeri interi modulo p primo, ovvero $\mathbb{Z}_p$

$$\mathbb{F}_p \cong \mathbb{Z}_p \tag{3.19}$$

allora si possono definire le curve effettivamente utilizzare in crittografia, le curve ellittiche prime

3.2.1 Curve ellittiche prime

Una curva ellittica prima E_p è una curva ellittica definita sul campo $\mathbb{Z}_p$, cioè sull'insieme dei numeri interi modulo p primo, tramite l'isomorfismo $\mathbb{F}_p \cong \mathbb{Z}_p$

$$\mathbb{K} = \mathbb{F}_p \cong \mathbb{Z}_p \longrightarrow \begin{cases} a, b \in \mathbb{Z}_p \\ (x, y) \in \mathbb{Z}_p^2 \end{cases} \tag{3.20}$$

Tutte le operazioni che sono applicate alle curve ellittiche definite su un campo $\mathbb{K}$ valgono anche per le curve ellittiche prime, con l'accortezza che sono tutte effettuate in modulo p, sia l'equazione che verificano i punti della curva, supposto $p > 3$:

$$y^2 \equiv x^3 + ax + b \pmod{p} \qquad a, b \in \mathbb{Z}_p \tag{3.21}$$

sia la condizione per cui si può definire la somma di punti, cioè 3.11:

$$\Delta = 4a^3 + 27b^2 \pmod{p} \neq 0 \tag{3.22}$$

sia le formule per la somma di punti, e così via.

La caratteristica particolare delle curve ellittiche prime, come quelle su campi finiti, è che sono composti da un numero finito di elementi, come si può vedere in figura 3.5.

Nonostante ciò, la curva ellittica mantiene la sua proprietà di simmetria orizzontale, che non sarà più rispetto all'asse x ma rispetto alla retta $y = \frac{p}{2}$, dovuto dal fatto che trattando numeri interi in modulo p si rinuncia ai numeri negativi: il dominio di tutti i valori, anziché essere definito da $-\frac{p}{2}$ a $\frac{p}{2}$, sarà da 0 a $p - 1$

$$(x, y) \in [0, p-1] \times [0, p-1] \tag{3.23}$$

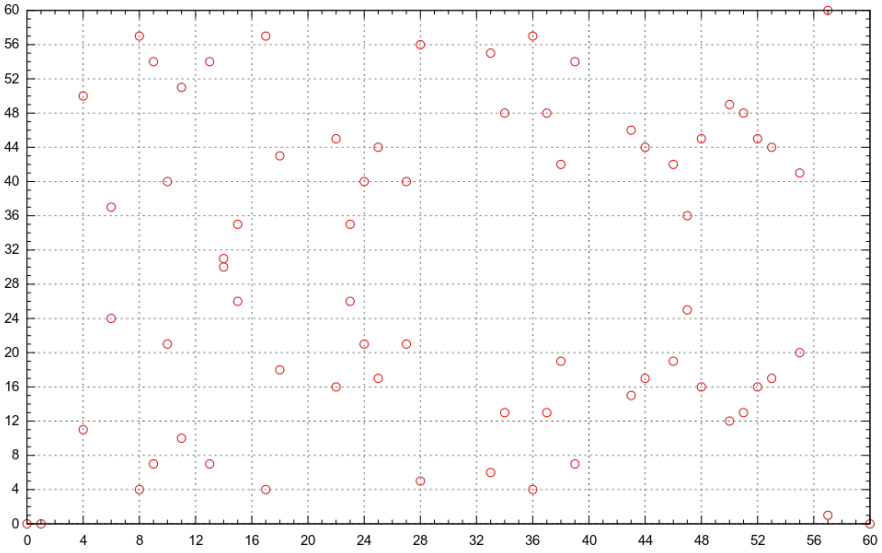


Figura 3.5: Curva ellittica prima definita in $\mathbb{Z}_{p=61}$

Una caratteristica importante da notare per le curve ellittiche prime è il fatto che, dato un qualsiasi valore $\overline{x} \in [0, p-1]$, e sostituito alla equazione della curva

$$\overline{y}^2 \equiv \overline{x}^3 + a\overline{x} + b \pmod{p} \tag{3.24}$$

non è garantita l'esistenza del valore $\overline{y}$, in quanto si deve ottenere come radice quadrata in modulo p dell'espressione della cubica, ovvero

$$\overline{y} = \sqrt{\overline{y}^2} \pmod{p} \tag{3.25}$$

questo è dato dal fatto che non tutti gli elementi x appartenenti al campo $\mathbb{Z}_p$ sono *residui quadratici*, ma all'incirca la metà, cioè $\frac{p-1}{2}$, lo saranno:

Definizione 11 (Residuo quadratico)**.** *Un numero intero x è detto residuo quadratico modulo p se esiste un intero y tale che*

$$y^2 \equiv x \ (mod\ p) \tag{3.26}$$

Segue che il numero di punti di una curva ellittica prima, detto anche *ordine*, non è noto a priori: la teoria[1] ci dimostra però che è un valore circa pari a p.

3.3 Moltiplicazione scalare di punti su curve ellittiche

Abbiamo già definito la somma di punti su curve ellittiche: è di nostro particolare interesse analizzare il caso in cui $P = Q$, detto anche *raddoppio*:

$$P + P = -R = 2P \tag{3.27}$$

[1] In particolare il teorema di Hasse

Il cui risultato, $2P$, sarà un punto appartenente alla curva.

In generale, si può sommare un punto con sé stesso più volte, ad esempio k volte:

$$\underbrace{P + P + ... + P}_{k \text{ volte}} = kP \tag{3.28}$$

Quest'ultimo risultato non corrisponde a moltiplicare le coordinate di P per lo scalare k, ma ad eseguire il raddoppio del punto P con sé stesso k volte: ad ogni raddoppio si ottiene un punto appartenente alla curva, per induzione quindi è semplice arrivare alla conclusione che anche kP sarà un punto appartenente alla curva.

Le operazioni svolte durante un raddoppio sono moltiplicazioni e somme, che constano di complessità al più quadratica

$$O(n^2)$$

con n pari al numero delle cifre dei numeri coinvolti.

Quindi andare a sommare un punto con sé stesso k volte sembrerebbe una operazione polinomiale a prima vista: in realtà, sapendo che k è composto da n cifre binarie, allora si ha che

$$n = \Theta(\log k) \longrightarrow k = 2^n \tag{3.29}$$

si ha che l'operazione risulta puramente esponenziale nel numero di cifre n

$$O(k) = O(2^n) \tag{3.30}$$

in pratica è inutilizzabile in crittografia, poiché necessario andare a lavorare con numeri di grandi dimensioni: esiste però un algoritmo efficiente per la moltiplicazione scalare di punti, detto algoritmo dei *raddoppi ripetuti*.

3.4 Il problema del logaritmo discreto

Se si vuole andare ad invertire l'operazione di moltiplicazione scalare di punti, si ottiene un problema molto difficile: il problema del logaritmo discreto su curve ellittiche

Definizione 12 (Logaritmo discreto su curve ellittiche - ECDLP)**.** *Noti due punti P, Q appartenenti ad una curva ellittica E, si richiede di trovare lo scalare k tale per cui vale la seguente relazione*

$$Q = kP \tag{3.31}$$

Il problema è tipicamente riassunto nella seguente equazione:

$$k = \log_P Q \tag{3.32}$$

Non si è al corrente dell'esistenza di algoritmi efficienti in grado di risolvere il problema ECDLP, ad oggi esso rimane puramente esponenziale nel numero di cifre di k, cioè n

$$O(2^{\frac{n}{2}})$$

Riassumendo, abbiamo una operazione definita su curve ellittiche, cioè la moltiplicazione scalare di punti, che risulta

- computazionalmente facile da calcolare tramite l'algoritmo dei raddoppi ripetuti
- computazionalmente difficile da invertire, poiché sarebbe necessario risolvere il problema ECDLP

ricopre quindi perfettamente il ruolo di funzione one-way trap-door da utilizzare con curve ellittiche.

Nell'algoritmo RSA la funzione one-way è la fattorizzazione di numeri semiprimi, qui invece sarà il problema ECDLP, il che è una ottima notizia in quanto sia un problema molto più difficile della fattorizzazione, quest'ultima difatti è ormai subesponenziale, mentre ECDLP è ancora un problema di difficoltà puramente esponenziale. [12]

Di conseguenza, la crittografia su curve ellittiche rispetto alla crittografia a chiave pubblica tradizionale:

- a parità di sicurezza, può lavorare su dati di dimensioni ridotte
- a parità di lunghezza delle chiavi, è molto più sicura, in quanto basata su un problema più difficile

Capitolo 4

Generatori di numeri casuali

La generazione di numeri casuali (RNG) è un processo software o hardware attraverso il quale si ottengono risultati impredicibili.

La casualità è impiegata in varie applicazioni al giorno d'oggi: nel gioco d'azzardo, nei giochi a ruolo, nei videogiochi, per simulazioni e anche in crittografia, anche se in quest'ultima con particolari accortezze per garantirne una maggior sicurezza d'utilizzo.

Il concetto e la generazione di casualità hanno origini antichissime: tra alcuni esempi più classici si possono includere il lancio di una moneta, di un dato ad n facce, mescolare delle carte da gioco, e moltissimi altri.

Le fonti più efficaci di casualità sono fenomeni presenti in natura, per esempio il segnale di rumore di sottofondo, di rumore termico, o altri segnali dovuti a fenomeni di natura elettromagnetica: in generale, maggiore è l'*entropia* di una fonte, più efficace sarà a generare casualità:

Definizione 13 (Entropia)**.** *L'entropia $H(X)$ di una variabile aleatoria X che assume valori nell'alfabeto $\mathcal{X}$, cioè $p : \mathcal{X} \to [0, 1]$, è la quantità media di informazione incerta contenuta nei suoi possibili output*

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_b p(x) \tag{4.1}$$

ove $p(x) \triangleq P(X = x)$ e la base del logaritmo b dipende dall'unità che si vuole usare per l'entropia: base 2 per bits, base e per nats, base 10 per dits.

In particolare, si preferiscono fonti con una grande quantità di entropia, cioè fonti il cui output è caratterizzato da una forte impredicibilità.

La forte richiesta di numeri casuali ha portato alla realizzazione di generatori di numeri casuali durante gli anni, sia tramite software che hardware: in base alla metodologia impiegata per la generazione, e conseguentemente alla "qualità" dei numeri che forniscono in uscita, si contraddistinguono i generatori in diverse famiglie, cioè PRNG, TRNG, CSPRNG, ove una è preferita rispetto all'altra dipendentemente dal ruolo che dovrà ricoprire il dato casuale che generano.

4.1 Pseudo-Random Number Generators

Per realizzare dei generatori di numeri casuali, l'approccio più semplice è di fare ricorso alla matematica attraverso *processi iterativi*, ovvero algoritmi che prendono in input una stringa breve, denominato seme (*seed*) x_0, dei parametri costanti, e stampano in output una sequenza di valori $x_1, x_2, ..., x_n$ *apparentemente casuali*, ottenuti da x_0 in maniera ricorsiva e con un certo *periodo* n, superato il quale la sequenza si ripeterà

$$x_1, x_2, ..., x_n \quad x_1, x_2, ..., x_n \quad ... \tag{4.2}$$

ove ogni elemento x_i dipende dal precedente x_{i-1}

$$x_i = f(x_{i-1}) \tag{4.3}$$

In qualsiasi momento, qualora si preferisca la generazione di bit anziché di numeri, si può stabilire di ottenere il bit pseudocasuale b_i attraverso un criterio specifico, ad esempio calcolando la parità del numero pseudocasuale x_i

$$b_i = 1 \longleftrightarrow x_i \bmod 2 = 1 \tag{4.4}$$

In questo caso, il numero x_i prenderà il nome di *stato interno* del generatore: questa è l'idea di fondo con cui si realizza un Pseudo-Random Number Generator (PRNG), conosciuti anche come *Deterministic Random Bit Generators* (DRBG) per via della deterministicità del valore x_i.

Sempre per questo motivo, questi generatori con lo stesso input, cioè x_0, daranno in uscita sempre lo stesso output, ovvero la sequenza $x_1, ..., x_n$: l'output assumerà quindi la denominazione di *pseudocasuale*, in quanto non sia completamente casuale, ma solo all'apparenza.

Di conseguenza, i valori forniti in output da questi generatori possono presentare anomalie o debolezze quali:

- valori di seed x_0 inadeguati, che risultano in periodi n molto brevi;
- sottosequenze di valori uguali generati a brevi distanze;

Nonostante questi non trascurabili svantaggi, i PRNG sono assai usati in una vasta gamma di applicazioni per la loro semplicità, leggerezza e portabilità.

4.1.1 Generatore lineare congruenziale

L'esempio più semplice di PRNG è il generatore lineare congruenziale (LCG), il quale, definito un seed breve x_0 in qualche maniera casuale, è tale che

$$x_i = (ax_{i-1} + b) \bmod n \qquad i \geq 1 \qquad (a, b, x_0 < n) \tag{4.5}$$

che, sotto opportune ipotesi su a, b genera una sequenza periodica e apparentemente casuale di numeri con periodo pari ad n [19].

Si può ben notare come con una realizzazione tramite LCG non siano generati output *completamente casuali*, in quanto, come si può vedere dall'equazione 4.5, l'elemento corrente della sequenza x_i dipende dal precedente x_{i-1} e tutti gli elementi sono funzione dell'elemento iniziale (seed) x_0

$$x_i, x_{i-1}, ..., x_1 = f(x_0) \tag{4.6}$$

Questa caratteristica è in linea con il concetto di base dei PRNG.

4.2 True Random Number Generators

Un True Random Number Generator (TRNG) è un dispositivo hardware che fornisce numeri casuali attraverso la misura di un fenomeno fisico, anziché tramite un algoritmo, e sono tipicamente realizzati da tre componenti principali:

- un trasduttore, che misura il fenomeno fisico e lo traduce in un segnale elettrico
- un amplificatore, che amplifica il segnale ottenuto dal trasduttore in modo che sia sufficientemente potente
- un convertitore analogico/digitale, per convertire in binario l'output in uscita dall'amplificatore, con in dotazione un clock che indicherà la cadenza con la quale verranno estratti campioni dal segnale analogico amplificato.

Un esempio di TRNG è rappresentato in figura 4.1

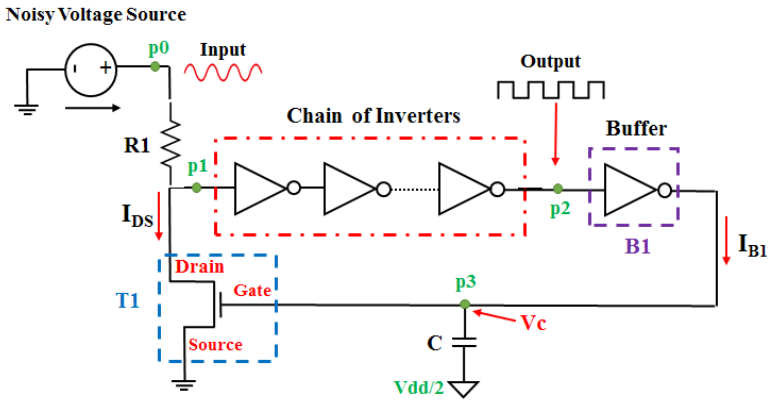


Figura 4.1: Esempio di TRNG [17]

L'output di questi generatori si contraddistinguono da quelli dei PRNG per la loro impredicibilità, dal momento che non sono basati su processi software iterativi con seed iniziale, ma da fenomeni fisici microscopici assai più imprevedibili.

Questa forte impredicibilità è molto attraente in crittografia, in quanto mette notevolmente a riparo l'output da possibili debolezze quali pattern ripetuti che faciliterebbero il crittoanalista ad organizzare un eventuale attacco sul generatore e quindi sul sistema crittografico che si sta impegnando di difendere.

La qualità dell'output dei TRNG, tuttavia, è pagata in termini in velocità: essi infatti risultano molto meno efficienti a generare bit casuali. Per questo motivo, vengono spesso impiegati nella generazione di seed, che sono stringhe tipicamente corte, da poi utilizzare per alimentare generatori CSPRNG.

I CSPRNG rimangono comunque generatori piuttosto lenti ma decisamente più rapidi dei TRNG: aumentando il numero di bit generati si perde in qualità dell'output, il ciò è un buon compromesso considerando il fatto che in crittografia si ha necessità di generare numeri composti da migliaia di bit che poi verranno utilizzati per generare chiavi crittografiche, come ad esempio i valori p, q dell'algoritmo RSA.

4.3 Cryptographically Secure Pseudo-Random Number Generators

Un Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) è un PRNG con particolari accortezze che lo rende sicuro da applicare in ambiti crittografici.

I CSPRNG sono stati ideati per far fronte al fatto che la casualità ricopre un ruolo fondamentale in crittografia, senza di essa, infatti, non si potrebbe parlare di crittografia in primo luogo. [1]

Alcune applicazioni molto importanti dei numeri casuali in crittografia includono:

- generazione di chiavi segrete da utilizzare nei cifrari
- generazione di *nonce*, cioè valori *usa e getta*, da utilizzare una volta sola

In crittografia l'output fornito in uscita da un generatore pseudocasuale qualunque non è *sufficientemente casuale*, è richiesto infatti che il generatore sia anche *crittograficamente sicuro*, ovvero che, oltre a soddisfare una serie di test statistici base necessari affinché la sequenza generata da sia sufficientemente esente da regolarità o predicibilità nella sua struttura, deve essere tale per cui:

- soddisfa il *test di prossimo bit* [9]
- garantisce la *backtracking resistance* [15]

Definizione 14 (Test di prossimo bit)**.** *Dati i primi n bit di una sequenza casuale ottenuta dal generatore, non deve esistere un algoritmo che in tempo polinomiale riesca a predire il bit* $n + 1$ *della sequenza, ovvero il prossimo bit, con probabilità strettamente maggiore di* $\frac{1}{2}$*.*

Definizione 15 (Backtracking resistance)**.** *Nel caso in cui lo stato interno del generatore venga parzialmente o anche totalmente compromesso, deve risultare impossibile risalire a valori generati in stati precedenti a quello compromesso.*

Queste sono condizioni molto stringenti che non tutti i PRNG riescono a soddisfare e che rallentano l'output di bit che riescono a generare, ma che sono di assoluta necessità per far si che si possano utilizzare in ambiti crittografici.

Inoltre, qualora si voglia utilizzare un CSPRNG si deve essere assolutamente sicuri che nessun elemento venga a conoscenza del seed oppure dello stato interno del generatore ad un qualsiasi istante poichè alla fine dei conti, un CSPRNG è anche un PRNG, e quindi è deterministico.

Questo è un motivo per il quale non solo il generatore deve essere realizzato propriamente ma anche il seed che riceverà dovrà essere generato correttamente e mantenuto assolutamente segreto, mai ceduto.

Capitolo 5

Il generatore Dual EC-DRBG

Il Dual EC-DRBG (*Elliptic Curve - Deterministic Random Bit Generator*) è un generatore di numeri pseudo-casuali crittograficamente sicuro (CSPRNG) basato su curve ellittiche, proposto dal NIST attraverso la pubblicazione SP 800-90A nel 2006 [4] e divenuto immediatamente standard.

5.1 Specifiche iniziali

EC-DRBG è un generatore basato su curve ellittiche prime, quindi necessiterà di una curva: in particolare, il NIST consiglia di utilizzare una tra le curve standard P-256, P-384 oppure P-521 [13], rispettivamente che lavorano con punti di coordinate a 256 bit, 384 bit oppure 521 bit, tutte verificanti l'equazione

$$y^2 \equiv x^3 - 3x + b \pmod{p} \tag{5.1}$$

ove b e p sono parametri che dipendono dalla curva stessa.

Dopodiché, per ogni curva standard il NIST fornisce anche due punti denominati P e Q generati dall'NSA[1] tali che:

- P è pari al generatore G, punto specifico di ogni curva tipicamente utilizzato come punto iniziale per gli algoritmi crittografici, che dipende dalla curva scelta;
- Q è un punto che anche esso dipende dalla curva scelta, ove il NIST non specifica in che modo sia stato ottenuto.

I due punti in questione sono necessari al generatore per poter funzionare: in linea teorica, dovrebbero essere due punti casuali della curva scelta, in pratica potrebbero non esserlo, e verrà esposto il perché.

[1] National Security Agency (https://www.nsa.gov/)

5.2 Struttura del generatore

La struttura del generatore è piuttosto semplice:

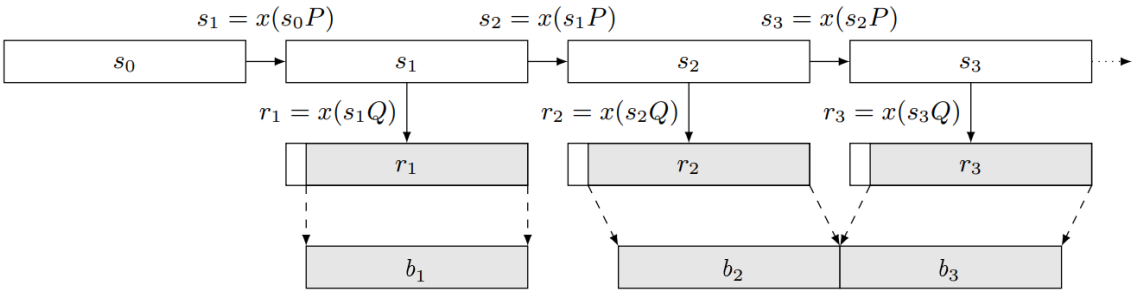


Figura 5.1: Dual EC-DRBG [6]

L'algoritmo alla base del generatore, in quanto PRNG, è innescato da un seed iniziale ottenuto in qualche maniera casuale, ad esempio tramite un TRNG: questo seed rappresenterà lo stato iniziale, denominato s_0.

Successivamente, si definiscano le seguenti funzioni:

- $x(P)$: ritorna la coordinata x del punto $P = (P_x, P_y)$ appartenente ad una curva ellittica, ovvero
$$x(P) = P_x \tag{5.2}$$
- $\varphi(x)$: scarta i 16 bit più significativi dalla rappresentazione binaria del numero x, in figura rappresentato dall'estrazione in colore grigio dai blocchi r_i, cioè
$$\underbrace{x_n\ x_{n-1}\ x_{n-2}\ \ldots\ x_{n-15}}_{\text{scartati}}\ x_{n-16}\ x_{n-17}\ \ldots\ x_1\ x_0 \qquad x_i \in \{0,1\} \tag{5.3}$$

A partire dallo stato iniziale si ottiene lo stato successivo s_1 andando ad estrarre la coordinata x del punto che si ottiene moltiplicando scalarmente lo stato iniziale s_0 con il punto P della curva:
$$s_1 = x(s_0P) \tag{5.4}$$
Il precedente risultato verrà combinato nella stessa maniera di s_0 tramite il punto Q per ottenere il primo blocco r_1 da cui dover estrarre i bit pseudo-casuali:
$$r_1 = x(s_1Q) \tag{5.5}$$
Dal quale si scartano i 16 bit più significativi tramite la funzione $\varphi(x)$, ottenendo il primo blocco di bit pseudo-casuali in output b_1
$$b_1 = \varphi(r_1) = \varphi(x(s_1Q)) \tag{5.6}$$
In quanto PRNG, ad ogni successivo ciclo viene ricalcolato lo stato interno s_i, che verrà utilizzato come seed all'iterazione successiva per ottenere un nuovo valore s_{i+1} per continuare il processo iterativo.

Come si può vedere dall'equazione 5.4 e dalla figura 5.1, lo stato successivo del generatore dipende da quello attuale

$$s_{i+1} = f(s_i) \tag{5.7}$$

mentre il blocco di bit pseudo-casuali b_i generati ad una certa iterazione i dipendono dallo stato interno in tale iterazione

$$b_i = g(s_i) \tag{5.8}$$

Con queste due proprietà si può dimostrare la proprietà di backtracking resistance del generatore, ma non solo: la presunta sicurezza dell'algoritmo risiede nel fatto che le frecce raffigurate siano a senso unico, ovvero rappresentano una funzione one-way

- è computazionalmente facile calcolare la moltiplicazione scalare di un punto di una curva ellittica, tramite algoritmo dei raddoppi ripetuti
- è computazionalmente difficile risalire allo scalare che ci ha dato il punto in output, conoscendo il punto originale e l'output: occorrerebbe risolvere il problema del logaritmo discreto su curve ellittiche (ECDLP, 3.31)

Lo stato del generatore s_i ad ogni iterazione dovrà essere quindi mantenuto segreto, altrimenti sarebbe computazionalmente facile, tramite l'algoritmo dei raddoppi ripetuti, calcolare tutti gli stati successivi ad i e conseguentemente tutti i bit pseudo-casuali b_i funzione di essi.

5.3 Descrizione e dimostrazione dell'attacco

A primo avviso, il generatore si direbbe che sia sicuro, in quanto protetto dal problema ECDLP, che è computazionalmente difficile.

In realtà nel 2007, due impiegati della Microsoft, Dan Shumow and Niels Ferguson, hanno dimostrato che con solo 256 bit di output, pari cioè ad un singolo blocco r_i, è possibile risalire allo stato interno del generatore [16].

Per poter applicare l'attacco in questione, detto attacco *Shumow-Ferguson*, si deve prima supporre che esista una *backdoor*, una relazione tra i due punti impiegati nel generatore, P e Q, cioè che siano tali che:

$$P = eQ \tag{5.9}$$

è anche necessaria la conoscenza del blocco di bit pseudo-casuali generati alla i-esima iterazione, che spesso sono mandati in chiaro nelle applicazioni ove si usa il CSPRNG:

$$b_i = \varphi(x(s_iQ)) \tag{5.10}$$

siccome per ottenere b_i si scartano i 16 bit più significativi della coordinata x del punto s_iQ, è sufficiente andare a generare tutte le possibili combinazioni dei 16 bit scartati, pari a

$$2^{16} = 65536 \text{ combinazioni} \tag{5.11}$$

per ognuna concatenarle a b_i e sostituire la concatenazione ottenuta, r_{i_x}, nell'equazione della curva ellittica impiegata nel generatore

$$r_{i_y}^2 = r_{i_x}^3 - 3r_{i_x} + b \pmod p \tag{5.12}$$

le coppie (r_{i_x}, r_{i_y}) che verificano la precedente equazione non saranno 2^{16} ma circa la metà, quindi circa 2^{15}, in quanto circa la metà dei valori di r_{i_x} sono residui quadratici: vi sarà però solo una tra le 2^{15} coppie $(\overline{r}_{i_x}, \overline{r}_{i_y})$ rispettante la precedente equazione che sarà tale per cui

$$R = (\overline{r}_{i_x}, \overline{r}_{i_y}) = s_iQ \tag{5.13}$$

ove s_i è ancora non noto, ma la precedente relazione esiste sicuramente: a questo punto, moltiplicando scalarmente il punto R trovato per lo scalare e, si ottiene

$$eR = e(s_iQ) = s_i(eQ) = s_iP \tag{5.14}$$

del quale si prenderà la coordinata x

$$x(s_iP) = s_{i+1} \tag{5.15}$$

ottenendo lo stato interno all'iterazione successiva e sbloccando la possibilità di potersi calcolare tutti gli stati successivi e i relativi bit generati, dall'iterazione $i+1$ inclusa in poi

$$b_{i+1} = \varphi(x(s_{i+1}Q))$$

$$s_{i+2} = x(s_{i+1}P) \longrightarrow b_{i+2} = \varphi(x(s_{i+2}Q))$$
$$s_{i+3} = x(s_{i+2}P) \longrightarrow b_{i+3} = \varphi(x(s_{i+3}Q))$$
$$\dots$$

a questo punto, il generatore è stato violato.

5.3.1 Osservazioni

Per poter applicare l'attacco, è necessaria l'esistenza della relazione espressa nell'equazione 5.9, tuttavia non sappiamo se lo scalare e esista o se il NIST abbia messo a disposizione i punti P e Q tali per cui esista tale relazione, e non possiamo neanche ad oggi dimostrarlo in pratica poiché l'unico modo che abbiamo ad oggi è, noti P e Q, quello di enumerare tutti i possibili e e per ognuno provare a calcolare

$$P = eQ \tag{5.16}$$

e verificare che P sia pari a G del generatore della curva utilizzata, questo corrisponde ad andare a risolvere il problema ECDLP che è un problema computazionalmente difficile.

La dimostrazione afferma infatti che P e Q *potrebbero* essere stati scelti dal NIST appositamente per inserire la backdoor, la dimostrazione però non può confermare che ci sia, a meno che non si riesca a trovare un algoritmo, se esiste, che in tempo polinomiale riesca a risolvere ECDLP, che ad oggi è sconosciuto, è altresì particolarmente importante evidenziare che la possibile esistenza della backdoor non è dovuto alle curve ellittiche impiegate, ma al modo in cui il generatore stesso è realizzato.

5.3.2 Conseguenze

La comunità tecnico-scientifica ha sospettato fortemente fin da subito la presenza di una backdoor: per ovviare, il NIST in una successiva revisione ha introdotto la possibilità di scegliere i propri parametri P e Q quando si implementa EC-DRBG nel proprio sistema, fornendo delle linee guida sul come andare a generare tali parametri in modo da massimizzarne la sicurezza, tuttavia il NIST stesso sconsiglia questa pratica in quanto neghi la validazione FIPS 140 [11], un insieme di standard definiti dal governo degli Stati Uniti che indica i minimi requisiti di sicurezza per moduli crittografici.

Il sospetto diventa ancora più sensato sapendo che nel 2004 fu rivelato da un articolo di Reuters che l'NSA ha firmato un accordo segreto da $10mln con RSA Security, importante azienda statunitense che si preoccupa di gestire standard crittografici, per l'adozione di EC-DRBG all'interno delle proprie librerie crittografiche [10], impiegate per realizzare numerosi algoritmi impiegati anche ad oggi, tra cui SSL e TLS.

Inoltre, nel 2013 un articolo di The Guardian riporta che alcuni documenti dell'NSA rivaleti da Edward Snowden dimostrano che l'NSA ha spinto a rendere il ge-

neratore fin da subito standard in modo da esserne l'unico mantenitore, e conferma che EC-DRBG effettivamente contiene una backdoor che favorisce l'NSA. [2]

Il NIST nega la presenza di una backdoor, ma il fatto che nel 2014 esso stesso abbia rimosso Dual EC-DRBG dai generatori standard dopo 8 anni consecutivi che era categorizzato come tale [14], praticamente ne conferma la presunta veridicità.

Capitolo 6

Proof of Concept

In questo capitolo verrà illustrato il codice del Proof of Concept (PoC), scritto in linguaggio Python versione 3.9.1, completo di commenti.

6.1 Classi e funzioni di utility

Il seguente blocco di codice contiene classi e funzioni di utility utilizzate per l'attacco, principalmente composto da funzioni helper per operazioni su curve ellittiche, tutte realizzate dall'autore della tesi, ad esclusione dell'algoritmo Tonelli-Shanks per il calcolo della radice quadrata in modulo p che è stata ottenuta da questa repository.

utils.py

```
class point:
    def __init__(self, x = 'O', y = 'O'):
        self.x = x
        self.y = y

    def is_punto_infinito(self):
        return (str(self)) == 'O'

    def __str__(self):
        if self.x == 'O' and self.y == 'O': return f'O'
        else: return f'({self.x}, {self.y})'

class prime_elliptic_curve:
    def __init__(self, p:int, a:int, b:int, G:point, name: str):
        self.a = a
        self.b = b
        self.p = p
        self.G = G
        self.keysize = len(bin(p)[2:])
        self.name = name

# CURVE A DISPOSIZIONE
# y^2 = x^3 + ax + b mod p
# prese da https://neuromancer.sk/std/ (NIST)

P192 = prime_elliptic_curve(
    p = 0xfffffffffffffffffffffffffffffffeffffffffffffffff,
    a = 0xfffffffffffffffffffffffffffffffeffffffffffffffc,
```

```
    b = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1,
    G = point(0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012,
              0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811),
    name = "P-192"
)

P224 = prime_elliptic_curve(
    p = 0xffffffffffffffffffffffffffffffff000000000000000000000001,
    a = 0xfffffffffffffffffffffffffffffffefffffffffffffffffffffffe,
    b = 0xb4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4,
    G = point(0xb70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21,
              0xbd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34),
    name = "P-224",
)

P256 = prime_elliptic_curve(
    p = 0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff,
    a = 0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc,
    b = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b,
    G = point(0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d8...,
              0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb640683...),
    name = "P-256"
)

P384 = prime_elliptic_curve(
    p = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe...,
    a = 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe...,
    b = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875a...,
    G = point(0xaa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082...,
              0x3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b...),
    name = "P-384"
)

P521 = prime_elliptic_curve(
    p = 0x01ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff...,
    a = 0x01ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff...,
    b = 0x0051953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef1...,
    G = point(0x00c6858e06b70404e9cd9e3ecb662395b4429c648139053fb521f828af...,
              0x011839296a789a3bc0045c8a5fb42c7d1bd998f54449579b446817afb...),
    name = "P-521"
)

# algoritmo di euclide esteso
# risolve l'identita' di Bezout ax + by = gcd(a,b)
# ritorna gcd(a,b), x, y
def EE(a: int, b: int):
    if b == 0:
        return [a,1,0]

    d_, x_, y_ = EE(b, a % b)

    dxy = [d_, y_, x_ - a//b * y_]
    return dxy

# calcola l'inverso moltiplicativo di a mod b
# cioe' risolve l'equazione x = a^-1 mod b
# a, b devono essere coprimi!
def inverso_moltiplicativo(a: int, b: int):
    gcdab, x, y = EE(a, b)
    if gcdab != 1:
        print(f'ERRORE: non posso calcolare l\'inverso moltiplicativo di {a} mod
    {b} poiche\' {a} e {b} non sono coprimi!')
        return 0
    else: return x
```

```
92
93  # controlla che il punto P appartenga alla curva ellittica prima Ep(a,b)
94  def appartiene_alla_curva(P: point, curva_ellittica:prime_elliptic_curve):
95      if P.is_punto_infinito(): return True
96
97      cubica = (pow(P.x, 3) + curva_ellittica.a * P.x + curva_ellittica.b) %
        curva_ellittica.p
98
99      ysquared = pow(P.y, 2, curva_ellittica.p)
100
101     if ysquared == cubica:
102         P.x %= curva_ellittica.p
103         return True
104     return False
105
106 # algoritmo tonelli-shanks per il calcolo della radice in modulo p
107 # source: https://gist.github.com/nakov/60d62bdf4067ea72b7832ce9f71ae079
108 def modular_sqrt(a: int, p: int):
109
110     def legendre_symbol(a: int, p: int):
111         """ Compute the Legendre symbol a|p using
112             Euler's criterion. p is a prime, a is
113             relatively prime to p (if p divides
114             a, then a|p = 0)
115             Returns 1 if a has a square root modulo
116             p, -1 otherwise.
117         """
118         ls = pow(a, (p - 1) // 2, p)
119         return -1 if ls == p - 1 else ls
120
121     """ Find a quadratic residue (mod p) of 'a'. p
122         must be an odd prime.
123         Solve the congruence of the form:
124             x^2 = a (mod p)
125         And returns x. Note that p - x is also a root.
126         0 is returned is no square root exists for
127         these a and p.
128         The Tonelli-Shanks algorithm is used (except
129         for some simple cases in which the solution
130         is known from an identity). This algorithm
131         runs in polynomial time (unless the
132         generalized Riemann hypothesis is false).
133     """
134     # Simple cases
135     #
136     if legendre_symbol(a, p) != 1:
137         return 0
138     elif a == 0:
139         return 0
140     elif p == 2:
141         return p
142     elif p % 4 == 3:
143         return pow(a, (p + 1) // 4, p)
144
145     # Partition p-1 to s * 2^e for an odd s (i.e.
146     # reduce all the powers of 2 from p-1)
147     #
148     s = p - 1
149     e = 0
150     while s % 2 == 0:
151         s //= 2
152         e += 1
153
154     # Find some 'n' with a legendre symbol n|p = -1.
```

```
155     # Shouldn't take long.
156     #
157     n = 2
158     while legendre_symbol(n, p) != -1:
159         n += 1
160 
161     # Here be dragons!
162     # Read the paper "Square roots from 1; 24, 51,
163     # 10 to Dan Shanks" by Ezra Brown for more
164     # information
165     #
166 
167     # x is a guess of the square root that gets better
168     # with each iteration.
169     # b is the "fudge factor" - by how much we're off
170     # with the guess. The invariant x^2 = ab (mod p)
171     # is maintained throughout the loop.
172     # g is used for successive powers of n to update
173     # both a and b
174     # r is the exponent - decreases with each update
175     #
176     x = pow(a, (s + 1) // 2, p)
177     b = pow(a, s, p)
178     g = pow(n, s, p)
179     r = e
180 
181     while True:
182         t = b
183         m = 0
184         for m in range(r):
185             if t == 1:
186                 break
187             t = pow(t, 2, p)
188 
189         if m == 0:
190             return x
191 
192         gs = pow(g, 2 ** (r - m - 1), p)
193         g = (gs * gs) % p
194         x = (x * gs) % p
195         b = (b * g) % p
196         r = m
197 
198 # verifica che x sia un residuo quadratico e le calcola la radice in modulo p
199 # ritorna int(0) se non ammette radice nel campo
200 def eval_cubica(x:int, curva_ellittica:prime_elliptic_curve):
201     ysquared = (pow(x, 3) + curva_ellittica.a * x + curva_ellittica.b) %
        curva_ellittica.p
202 
203     return modular_sqrt(ysquared, curva_ellittica.p)
204 
205 # calcola S = P + Q
206 # accetta due point e ritorna un point
207 # la somma la effettua in una curva elitticha prima Ep(a,b)
208 # gestisce il caso se il punto e' il punto all'infinito
209 def somma_di_punti(P: point, Q: point, curva_ellittica:prime_elliptic_curve):
210     if curva_ellittica.p < -1 or curva_ellittica.p == 0:
211         print(f'ERRORE: non posso fare la somma con p negativo o uguale a zero!')
212         return -1
213 
214     S = point()
215 
216     if appartiene_alla_curva(P, curva_ellittica) == False:
```

```
217         print(f'ERRORE: il punto P = {P} non appartiene alla curva ellittica
    specificata!')
218         return -1
219 
220     if appartiene_alla_curva(Q, curva_ellittica) == False:
221         print(f'ERRORE: il punto Q = {Q} non appartiene alla curva ellittica
    specificata!')
222         return -1
223 
224     if Q.is_punto_infinito():
225         return P
226 
227     if P.is_punto_infinito():
228         return Q
229 
230     # caso in cui P = -Q
231     if curva_ellittica.p - P.y == Q.y:
232         return S
233 
234     if P.x == Q.x and P.y == Q.y:
235         lambdaa = ((3 * (P.x ** 2) + curva_ellittica.a) *
    inverso_moltiplicativo(2 * P.y, curva_ellittica.p)) % curva_ellittica.p
236     else:
237         lambdaa = ((Q.y - P.y) * inverso_moltiplicativo(Q.x - P.x,
    curva_ellittica.p)) % curva_ellittica.p
238 
239     S.x = lambdaa ** 2 - P.x - Q.x
240     S.x %= curva_ellittica.p
241 
242     S.y = -Q.y + lambdaa * (Q.x - S.x)
243     S.y %= curva_ellittica.p
244 
245     return S
246 
247 # calcola Q = kP ove P appartiene alla curva ellittica prima Ep(a,b)
248 def raddoppi_ripetuti(P: point, k: int, curva_ellittica:prime_elliptic_curve):
249     if k <= 0:
250         print(f'ERRORE: non posso fare i raddoppi se k <= 0')
251         return -1
252 
253     if curva_ellittica.p <= 0:
254         print(f'ERRORE: non posso fare i raddoppi se p <= 0')
255         return -1
256 
257     if appartiene_alla_curva(P, curva_ellittica) == False:
258         print(f'ERRORE: il punto P = {P} non appartiene alla curva ellittica
    specificata!')
259         return -1
260 
261     bink = bin(k)[::-1][:-2]
262     maxk = len(bink)
263     points = [P]
264 
265     for i in range(1, maxk):
266         new = somma_di_punti(points[-1], points[-1], curva_ellittica)
267         points.append(new)
268 
269     result = -1
270     for i, digit in enumerate(bink):
271         if(digit == '1'):
272             if result == -1: result = points[i]
273             else: result = somma_di_punti(result, points[i], curva_ellittica)
274 
275     return result
```

6.2 Semplice realizzazione EC-DRBG

Nel seguente blocco di codice è implementata una classe con alcune funzioni base che simula il comportamento del generatore EC-DRBG, che servirà per dimostrare l'attacco.

EC_DRBG.py

```python
from utils import point, prime_elliptic_curve, raddoppi_ripetuti
import secrets # libreria per la casualita', crittograficamente sicura

class EC_DRBG:
    def __init__(self, P: point, Q: point, prime_elliptic_curve:
    prime_elliptic_curve, seed = secrets.randbits(128)):
        self.__current_state = seed
        self.P = P
        self.Q = Q
        self.prime_elliptic_curve = prime_elliptic_curve

    # rimuove i 16 bit piu' significativi da x
    def __phi(self, x):
        # notare che l'output dipende dalla grandezza della curva
        return bin(x)[2:].zfill(self.prime_elliptic_curve.keysize)[16:]

    # genera (self.prime_elliptic_curve.keysize - 16) bit pseudocasuali per
    iterazione
    def generate_bits(self, iterations = 1):
        output = ''

        for _ in range(iterations):
            # calcolo stato successivo
            next_state = raddoppi_ripetuti(self.P, self.__current_state,
    self.prime_elliptic_curve).x
            # aggiornamento stato successivo
            self.__current_state = next_state
            # generazione bit
            bits = self.__phi(raddoppi_ripetuti(self.Q, next_state,
    self.prime_elliptic_curve).x)
            # concatenazione
            output += bits

        return output
```

6.3 Applicazione dell'attacco

Nel seguente blocco di codice viene implementato l'attacco vero e proprio, ove il programma notifica costantemente l'utente di ciò che sta calcolando.

EC_DRBG_vuln.py

```
from time import time
from EC_DRBG import *
from utils import *

print("\n-- SIMULAZIONE ATTACCO AL GENERATORE EC-DRBG TRAMITE BACKDOOR --\n")

# richiesta input all'utente
while(True):
    scelta = input("Scegliere la curva ellittica da impiegare sul DRBG (P-192, P-224, P-256, P-384, P-521): ")

    if(scelta == P192.name):
        curva_scelta = P192
        break
    elif(scelta == P224.name):
        curva_scelta = P224
        break
    elif(scelta == P256.name):
        curva_scelta = P256
        break
    elif(scelta == P384.name):
        curva_scelta = P384
        break
    elif(scelta == P521.name):
        curva_scelta = P521
        break
    else:
        print("Inserire una curva ellittica tra quelle elencate!\n")

# prendiamo Q pari al punto base G della curva per semplicita' (essendo un punto che ne appartiene)
Q = curva_scelta.G

# introduciamo la backdoor, cioe' calcoliamo P = eQ
e = 0xdeadbeef
P = raddoppi_ripetuti(Q, e, curva_scelta)

# supponiamo che il generatore di cui vogliamo conoscere lo stato utilizzi P, Q con la backdoor (P = eQ)
generatore_vulnerabile = EC_DRBG(P, Q, curva_scelta)

# attenzione: non stiamo usando i P, Q dati dal NIST, poiche' non sappiamo se ci sia veramente una relazione tra P, Q che fornisce il NIST

# generiamo un po' di bit
niterazioni = 10
print(f"\nEseguo {niterazioni} iterazioni dell'EC-DRBG...")
generatore_vulnerabile.generate_bits(niterazioni)

# supponiamo di ottenere i bit generati alla i-esima iterazione (che e' facile, tipicamente sono mandati in chiaro nelle applicazioni in cui vengono usati)
bi = generatore_vulnerabile.generate_bits()
# generiamo anche i bit alla iterazione successiva, da usare nel confronto
bi_plus_one = generatore_vulnerabile.generate_bits()

```

```
# applichiamo l'attacco shumow-ferguson e dimostriamo di saper generare correttamente i bit da qui in avanti
# ipotizziamo di essere il NIST: generatore_vulnerabile e' una istanza del generatore di cui non conosciamo il seed che pero' sappiamo utilizza i P, Q e la curva da noi consigliate
start_time = time()

# e' piu' veloce prima trovare gli R candidati (poiche' sono 2^15 anziche' 2^16)
candidatiR = []
# enumeriamo tutti i possibili 16 bit piu' significativi che sono stati scartati da ri, concateniamoli ad esso ottenendo rix e verifichiamo che esista una coordinata riy associata ad rix
# alla fine otterremo che circa la meta' degli rix ammettono radici nel campo (quindi 2^15 con 16 bit da enumerare)
niterazioni = pow(2, 16)
for guess in range(0, niterazioni):
    print(f"Concateno e verifico residuo quadratico per {guess} di {niterazioni}...", end='\r')
    # concateno
    rix = int(bin(guess)[2:].zfill(16) + str(bi), 2)

    # se e' un residuo quadratico, aggiungiamo alla lista
    riy = eval_cubica(rix, curva_scelta)
    if(riy != 0):
        candidato_R = point(rix, riy)
        candidatiR.append(candidato_R)

print(f"Concateno e verifico residuo quadratico per {niterazioni} di {niterazioni}...")

# ci sara' sicuramente il punto R relativo allo stato successivo, andiamo a controllare quale sia (alla peggio sono 2^15 iterazioni)
ncandidatiR = len(candidatiR)
i = 0
for R in candidatiR:
    print(f"Verifico i candidati punti R della curva ellittica {curva_scelta.name}, candidato {i} di {ncandidatiR}...", end='\r')
    i += 1
    # calcoliamo il possibile si_plus_one P = eR
    candidato_si_plus_one = raddoppi_ripetuti(R, e, curva_scelta).x
    # calcoliamo il blocco dei bit associati ad esso facendo si_plus_one Q
    candidato_bi_plus_one = bin(raddoppi_ripetuti(Q, candidato_si_plus_one, curva_scelta).x)[2:].zfill(curva_scelta.keysize)[16:]

    # se sono uguali a quelli del DRBG, con assoluta certezza lo stato utilizzato per ottenerli sara' lo stato interno successivo
    if(candidato_bi_plus_one == bi_plus_one):
        # generiamo una copia del DRBG, utilizzando come seed lo stato appena trovato
        generatore_clonato = EC_DRBG(P, Q, curva_scelta, candidato_si_plus_one)
        print(f"Verifico i candidati punti R della curva ellittica {curva_scelta.name}, candidato {i} di {ncandidatiR}...")
        print(f"\nTrovato lo stato successivo in {round(time() - start_time, 2)} secondi!")
        print(f"> s[i+1] = {hex(candidato_si_plus_one)}")
        print(f"> R = ({hex(R.x)}, {hex(R.y)})")
        break

# verifiche aggiuntive, non necessarie ma per scaramanzia
print(f"\nVerifica iterazioni successive:")

n = 5
for i in range(1, n+1):
    print(f"\n-- iterazione i+{i+1} --")
```

```
100
101     print(f"I due blocchi di bit generati sono uguali?:
        {generatore_vulnerabile.generate_bits() ==
        generatore_clonato.generate_bits()}")
102
103 print(f"\nAttacco eseguito correttamente.")
```

6.4 Benchmark

In questa sezione si riportano dati di benchmark per l'attacco Shumow-Ferguson, su un Desktop PC dotato di CPU AMD Ryzen 7 5800X 8-core 4.6 GHz e RAM 2x8GB DDR4 3200 MHz.

Il benchmark è stato effettuato su un campione di 20 esecuzioni per ciascuna curva ellittica (P-192, P-224, P-256, P-384 e P-521), per un totale di 100 esecuzioni, con seed casuale ad ogni istanza dell'EC-DRBG, lasciando i punti P, Q e la loro relazione invariati, ove l'output è stato estratto tramite il seguente script in linguaggio batch

benchmark.bat

```
@echo off
setlocal enabledelayedexpansion

set curves=P-192 P-224 P-256 P-384 P-521

for %%i in (%curves%) do (
  echo Testing curve %%i
  echo. > "%%i benchmark tests.txt"

  for /L %%j in (1, 1, 20) do (
    echo starting iteration number %%j...

    python "EC_DRBG_vuln.py" %%i >> "%%i benchmark tests.txt"

    echo iteration number %%j complete
  )
  echo tests for curve %%i complete
)

echo all tests complete!

pause
endlocal
```

I test, per semplicità, sono eseguiti sequenzialmente e non in parallelo, il risultato di ogni test è poi combinato per il calcolo della media e della varianza di ciascuna curva utilizzando il seguente programma

stat.py

```
from statistics import mean, variance

curves = ["P-192", "P-224", "P-256", "P-384", "P-521"]
test_values = []

print("\n-- ANALISI DEI TEST PER L'ATTACCO SHUMOW-FERGUSON --\n")

for curve in curves:
    print("> " + curve)
    # apertura del file
    with open(curve + " benchmark tests.txt", "r") as file:

        content = file.readlines()

```

```
        # lettura riga per riga
        for line in content:
            # individuazione degli indici di inizio e fine
            start_index = line.find("in ") + 2
            end_index = line.find("secondi", start_index)

            if start_index != -1 and end_index != -1:
                test_value = float(line[start_index:end_index].strip())
                test_values.append(test_value)

    media = round(mean(test_values), 1)
    varianza = round(variance(test_values), 1)

    print("Media dei risultati: " + str(media) + " sec")
    print("Varianza dei risultati: " + str(varianza))
    print("\n")
```

Ottenendo infine dei risultati per ogni curva riassunti nella seguente tabella:

Curva	**Media** (secondi)	**Varianza**
P-192	158.0	14642.8
P-224	242.8	30760.2
P-256	280.6	38043.5
P-384	403.1	1.58×10^5
P-521	629.2	5.25×10^5

Tabella 6.1: Benchmark dell'attacco Shumow-Ferguson

Si può notare come all'aumentare dei bit del modulo la media aumenta, tuttavia anche la varianza subisce un incremento notevole: questo significa che l'aumento dei bit del modulo non mette praticamente per niente a riparo dall'attacco, ne aumenta il costo ma in maniera quasi impercettibile.

È di particolare importanza notare che i numeri raffigurati in tabella 6.1 sono molto piccoli per essere un attacco ad un PRNG, basti pensare che le curve in esame sono tali per cui il costo dell'attacco ECDLP sarebbe pari ad $O(2^{\frac{p}{2}})$, ove p è i bit del modulo, questi invece sono numeri che dimostrano chiaramente che l'attacco sia di costo sicuramente non esponenziale.

Capitolo 7

Conclusioni

Da tutto quello che è stato esposto fin'ora, possiamo trarre le seguenti conclusioni riguardo EC-DRBG:

- non è del tutto confermato che sia presente una backdoor ad oggi, però i parametri P e Q potrebbero benissimo essere stati scelti dall'NSA a tal scopo;
- il generatore, indipendentemente dai bit della curva, scarta sempre e solamente i 16 bit più significativi dal blocco r_i, il che è non è affatto sufficiente in termini di difficoltà dell'enumerazione: per ovviare è sufficiente che il generatore scarti più bit dal blocco r_i

Bit scartati	**Enumerazioni**	**Bit in output per iterazione**
16	2^{16}	240
$\frac{p}{3}$	2^{87}	169
$\frac{p}{2}$	2^{128}	128
$\frac{3}{4}p$	2^{172}	84

Tabella 7.1: Enumerazioni in funzione di bit scartati EC-DRBG (P-256)

  aumentando i bit scartati il generatore fornisce in output meno bit per iterazione, quindi risulterebbe più lento, ma aumenterebbero le enumerazioni necessarie per l'attacco, ovvero il costo, quindi la sicurezza del generatore;

- è possibile generarsi i propri P e Q, ma questa è una pratica in generale rischiosa se non fatta in modo appropriato;
- anche senza backdoor, è possibile, seppur computazionalmente difficile, risalire allo stato interno s_i da s_iQ enumerando i bit scartati come spiegato e risolvendo una singola istanza del problema ECDLP;

in conclusione EC-DRBG, nella sua interezza, è un generatore di numeri pseudo-casuali insicuro e assolutamente da non impiegare come CSPRNG.

7.1 Ringraziamenti

Volevo dedicare questo piccolo spazio per ringraziare di cuore tutti coloro che hanno creduto in me sin dall'inizio e mi hanno dato la spinta per andare avanti, in particolare i miei genitori e parenti, che mi hanno insegnato a lottare, lottare, lottare e mai arrendersi, i miei amici, che mi hanno sempre supportato anche nei momenti più difficili pur nonostante tutte le volte che gli ho detto di no per avere più ore per studiare e il mio fido destriero, la gatta, che non mi ha mai abbandonato.

Vorrei anche ringraziare tutti i fantastici professori che ho incontrato nel mio percorso, di cui ultimo ma non per importanza il Dott. Gaspare Ferraro, che mi ha seguito con enorme interesse, e tutti i compagni di uni con cui ho condiviso risate, lacrime ma anche tante, tantissime soddisfazioni.

Appendice A

Curve ellittiche consigliate

Non tutte le curve ellittiche sono utilizzabili in crittografia: per via delle proprietà delle curve stesse, ci sono curve più deboli, ovvero facilmente attaccabili, e curve meno deboli, curve più efficienti e curve meno efficienti.

Il NIST nel 1999 ha stilato una pubblicazione, il FIPS 186, la cui più moderna revisione è il FIPS 186-4 [13], dove ha consigliato l'uso di una serie di curve ellittiche ad oggi inattaccate, ovvero tali che ad oggi non esistono algoritmi in grado di comprometterne la sicurezza.

Alcune tra le curve in questione sono d'interesse per la Proof of Concept, cioè le curve prime P-192, P-224, P-256, P-384 e P-521, ove il numero ne indica la dimensione in bit del modulo p, più precisamente:

Curva	**Modulo** p
P-192	$2^{192} - 2^{64} - 1$
P-224	$2^{224} - 2^{96} - 1$
P-256	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
P-384	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
P-521	$2^{521} - 1$

Ognuna delle curve sopra illustrate rispettano l'equazione

$$y^2 = x^3 - 3x + b \pmod{p} \tag{A.1}$$

Ove i parametri b e il generatore G dipendono dalla curva stessa.

Bibliografia

[1] April, B. Randomness and cryptography with yevgeniy dodis. *The Courant News 14* (2019), 4–5.

[2] Ball James, Borger Julian, G. G. Revealed: how us and uk spy agencies defeat internet privacy and security.

[3] Barker, E., and Dang, Q. Nist special publication 800-57 part 3 revision 1: Recommendation for key management: Application-specific key management guidance. Tech. rep., National Institute of Standards and Technology, 22th January 2015.

[4] Barker, E. B., and Kelsey, J. M. Recommendations for random number generation using deterministic random bit generators. Special Publication 800-90A, National Institute of Standards and Technology, June 2006.

[5] Dana Neustadter, Sr. Product Marketing Manager, S. True random number generators for heightened security in any soc. Tech. rep., Synopsys, 2019.

[6] Daniel J. Bernstein, Tanja Lange, R. N. Dual ec: A standardized back door. Tech. rep.

[7] Ferraro, G. Crittografia su curve ellittiche.

[8] Inc., R. S. Rsa security releases rsa encryption algorithm into public domain. Tech. rep., 6th September 2000.

[9] Katz, Jonathan; Lindell, Y. Introduction to modern cryptography. crc press.

[10] Menn, J. Exclusive: Secret contract tied nsa and security industry pioneer.

[11] of Standards, N. I., and Technology. Security requirements for cryptographic modules. Tech. rep., 25th May 2001.

[12] of Standards, N. I., and Technology. The case for elliptic curve cryptography. Tech. rep., 15th Jan 2009.

[13] of Standards, N. I., and Technology. Digital signature standard (dss). Tech. rep., 19th July 2013.

[14] of Standards, N. I., and Technology. Nist removes cryptography algorithm from random number generator recommendations. Tech. rep., 21th April 2014.

[15] of Standards, N. I., and Technology. Glossary: Backtracking resistance. *Computer Security Resource Center* (2023).

[16] Shumow, D., and Ferguson, N. On the possibility of a back door in the nist sp800-90 dual ec prng. Tech. rep., Microsoft, August 2007.

[17] Tehranipoor, F., Wortman, P., Nima, K., Wei, Y., and Chandy, J. Dvft: A lightweight solution for power supply noise based trng using a dynamic voltage feedback tuning system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems PP* (02 2018).

[18] Treccani, E. Definizione di crittografia, 2023.

[19] Vasudevan, V., and Andersen, D. G. System support for speculative execution in distributed machine learning. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (2016), p. 4.