ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

A proposito di Crittografia a chiave asimmetrica e numeri primi: tecniche note e proposta di un nuovo test di primalità euristico e deterministico

Relatore:
Chiar.mo Prof.
Ozalp Babaoglu

Presentata da:
Francesco Sovrano

Sessione I
Anno Accademico 2015/2016

A chiunque abbia un briciolo di follia

Introduzione

Con questa tesi si spiegherà l'intrinseca connessione tra la matematica della teoria dei numeri e l'affidabilità e sicurezza dei crittosistemi asimmetrici moderni. I principali argomenti trattati saranno la *crittografia a chiave pubblica* ed il problema della *verifica della primalità*.
Nel capitolo 1 si capirà cos'è la crittologia, distinguendola nelle seguenti branche

- Crittografia, l'arte di proteggere le informazioni
- Crittoanalisi, l'arte di analizzare e carpire informazioni crittografate

Verranno inoltre presentati i principi alla base della crittografia moderna e le principali tipologie di attacchi crittoanalitici, spiegando cosa voglia dire segretezza perfetta ed in che modo sia possibile ottenerla.
Nel capitolo 2 si entrerà più nel dettaglio di cosa sia la crittografia, descrivendo le differenze tra asimmetria e simmetria delle chiavi, mostrando i vantaggi del combinare pro e contro di entrambe le tecniche ed entrando nel dettaglio di alcuni protocolli e utilizzi principali della crittografia asimmetrica:

- Comunicazione sicura: il protocollo DHM
- Firma digitale
- Certificati digitali

Nel capitolo 3 verrà fatta maggiore luce su cosa sia un crittosistema asimmetrico, mostrando algoritmi per

- Comunicare in modo confidenziale: *RSA*

- Scambiare in modo sicuro chiavi private su un canale insicuro: *ElGamal*
- Firmare messaggi: *DSA*
- Certificare identità e chiavi pubbliche: *ECQV*

Verrà inoltre introdotta l'importanza dei numeri primi come punto in comune tra ognuna delle tecniche appena elencate.
La tesi proseguirà con la spiegazione di quale sia la natura dei problemi alla base della sicurezza dei crittosistemi asimmetrici più diffusi, dimostrando l'importanza che riveste in questo contesto il problema della verifica della primalità. Nel capitolo 4 verranno identificate le caratteristiche peculiari alla base della sicurezza dei crittosistemi asimmetrici più diffusi, spiegando cosa sono la teoria della complessità computazionale e la teoria dei numeri e mostrando quali siano i problemi matematici più usati nella crittografia a chiave pubblica:

- Il problema della fattorizzazione
- Il problema del logaritmo discreto in gruppi ciclici e in sottogruppi ciclici di curve ellittiche su campi finiti

Sempre nello stesso capitolo verrà argomentata e classificata la sicurezza dei problemi citati pocanzi, presentando i migliori algoritmi noti allo stato dell'arte per affrontarli.
Nel capitolo 5 verrà mostrato come il calcolo quantistico possa influire sulla sicurezza intrinseca dei crittosistemi asimmetrici ad oggi più diffusi, introducendo brevemente cos'è la crittografia post-quantistica e su quali problemi matematici ha le sue fondamenta.
Nel capitolo 6 verrà spiegato cos'è un test di verifica della primalità, illustrando la differenza fra determinismo e probabilismo e descrivendo alcuni tra i test di primalità più efficienti ed usati:

- AKS
- ECPP
- Il test di Miller
- Il test di Fermat

- Il test di Miller-Rabin

Nel capitolo 7 verrà presentata una nuova tecnica per migliorare l'affidabilità del test di Fermat mediante un nuovo algoritmo *deterministico* per fattorizzare gli pseudoprimi di Carmichael euristicamente in tempo $\tilde{O}(\log^3 n)$.
Nel capitolo 8 verrà mostrato come, dall'algoritmo descritto alla fine del capitolo 7, sia possibile ottenere un nuovo test di primalità deterministico ed euristico con complessità $\tilde{O}(\log^2 n)$ e la cui probabilità di errore tende a 0 con n che tende ad infinito. Sempre nello stesso capitolo verrà descritta una nuova famiglia di interi chiamati numeri Esploratori. Sulla base di argomentazioni empiriche ed euristiche si ritiene che tali numeri dominino il caso pessimo del nuovo test di primalità, che verrà quindi chiamato *test degli Esploratori* in loro onore.
Nel capitolo 9 verranno fatti i dovuti ringraziamenti alle istituzioni ed alle persone che hanno aiutato nella ricerca sui nuovi algoritmi proposti in questa tesi.
Nell'appendice A verrà spiegato il significato di alcuni termini utilizzati all'interno della tesi.
Nell'appendice B verrà descritta la notazione d'ordine adottata.
Nell'appendice C verrà spiegato il modello di calcolo del costo computazionale adottato per le principali operazioni aritmentiche.
Nell'appendice D verrà descritto il metodo di fattorizzazione di Fermat, ritenuto utile per contestualizzare meglio alcune proprietà dei numeri Esploratori e degli algoritmi di fattorizzazione dei numeri di Carmichael.
Nell'appendice E, verrà descritto un algoritmo probabilistico e polinomiale per fattorizzare i numeri di Carmichael usando il test di Miller-Rabin.
Nell'appendice F verranno mostrati alcuni esempi numerici relativi ai nuovi algoritmi descritti nei capitoli 7 e 8.

Indice

Elenco delle figure

Elenco dei codici C++

Elenco delle tabelle

Capitolo 1

La crittologia

La crittologia è il campo di ricerca e di applicazione che riguarda le scritture segrete ed in particolare le scritture cifrate. [18]
Più nel dettaglio, la crittologia è suddivisa in:

- Crittografia
- Crittoanalisi

1.1 Crittografia

La crittografia è una tecnica di rappresentazione di un messaggio in una forma tale che l'informazione in esso contenuta possa essere recepita solo dal destinatario. [18]
Più nel dettaglio, la crittografia riguarda la progettazione e lo sviluppo di protocolli e tecniche che proteggano il contenuto di un messaggio privato in modo che non sia letto da terzi o contraffatto.
Da un punto di vista temporale, la crittografia è suddividibile in

- Crittografia antica: sinonimo di *cifratura* di un'informazione
- Crittografia moderna: confidenzialità delle informazioni (permessa anche dalla cifratura), integrità dei dati, autenticazione, impossibilità di ripudiare la propria identità o le proprie azioni

La crittografia moderna è il frutto dell'intersezione fra le discipline più disparate, tra cui

- Matematica
- Informatica
- Ingegneria elettrica e quantistica

La crittografia ha trovato larga applicazione in campo militare, diplomatico e commerciale ed il suo sviluppo è stato fortemente condizionato dall'evoluzione delle tecnologie di comunicazione adottate in questi settori. La crittografia viene applicata in molti contesti, tra cui

- Commercio elettronico
- Password di calcolatori e altri dispositivi
- Transazioni bancarie
- Bitcoin e altre valute virtuali

1.1.1 I principi di Kerckhoffs

Si può dire che *alla base della crittografia moderna* ci siano i principi di Kerckhoffs. Nel 1883 Auguste Kerckhoffs enunciò sei principi progettuali per la creazione di cifrari militari sicuri:

1. Il sistema deve essere praticamente, se non matematicamente, indecifrabile.
2. Il sistema non deve essere segreto, dev'essere in grado di cadere nelle mani del nemico senza inconvenienti.
3. La chiave segreta deve essere comunicabile senza l'aiuto di note scritte e dev'essere facilmente modificabile dai corrispondenti.
4. Il sistema deve essere applicabile alla corrispondenza telegrafica.
5. Il sistema deve essere portatile ed il suo utilizzo ed uso non deve richiedere il concorso di più persone.

6. È necessario che il crittosistema sia facile da usare e che non richieda la conoscenza e l'uso di una lunga serie di regole.

Si noti come l'insieme di regole appena descritte sia in contrapposizione con i principi della sicurezza tramite oscuramento delle informazioni: nel caso pessimo, tenere segreto il funzionamento di un crittosistema non lo rende più sicuro.

1.2 Crittoanalisi

La crittoanalisi è la parte della crittologia che si occupa dell'interpretazione dei testi cifrati, detti anche crittogrammi. [18] In particolare, è lo studio delle tecniche per decifrare sistemi o scritti che siano stati crittografati. Dunque, un crittoanalista è un esperto nell'analizzare ed attaccare codici e cifrari.
Generalmente, un crittogramma è una combinazione opportuna del testo in chiaro e della chiave di cifratura. Usando la chiave di decifratura, un crittogramma può essere trasformato nel testo in chiaro corrispondente.

1.2.1 Principali tipologie di attacco

Per identificare le tipologie di attacco si devono identificare le informazioni di una conversazione cifrata che un crittoanalista può conoscere:

- Le funzioni di cifratura e decifratura usate per la conversazione cifrata
- A volte, alcune informazioni raccolte ascoltando la conversazione cifrata

Le principali categorie di attacco sono quindi:

- *Attacco a forza bruta*; il crittoanalista conosce solo gli algoritmi usati per cifrazione e decifrazione e quindi prova tutte le possibili chiavi esistenti
- *Attacco al testo cifrato*; il crittoanalista sfrutta una collezione di crittogrammi utilizzati in passato dall'attaccato, in modo da ricostruire il testo in chiaro usando ricorrenze nei crittogrammi

- *Attacco su testo in chiaro noto*; il crittoanalista conosce alcuni testi in chiaro ed i loro corrispondenti crittogrammi, tali conoscenze possono essere sfruttate per risalire alla chiave di cifratura e/o decifratura

- *Attacco su testo in chiaro predefinto*; il crittoanalista è in grado di far generare all'attaccato crittogrammi partendo da testi in chiaro noti, tali conoscenze possono essere sfruttate per risalire alla chiave di cifratura e/o decifratura

1.3 Esiste un cifrario perfetto?

Un'idea sbagliata abbastanza comune è pensare che ogni metodo di crittazione possa essere rotto. Claude Shannon, un matematico Americano, ingegnere elettrico e crittografo conosciuto come il padre della teoria dell'informazione, provò che il cifrario One-Time Pad non può essere rotto; questo nel caso in cui la chiave di cifratura sia veramente random, mai riusata e grande almeno quanto il messaggio stesso.
Dunque, per rispondere alla domanda si deve distinguere segretezza perfetta da segretezza computazionale.
Se per segretezza perfetta si intedesse l'impossibilità da parte di terzi di carpire il contenuto di un messaggio segreto di dimensioni finite, allora essa non sarebbe possibile. Infatti, si supponga che un messaggio M sia cifrato usando una chiave k appartenente all'insieme K di tutte le chiavi possibili. Se $|K|$ è il numero di chiavi possibili e K è un insieme finito, allora provare tutte le chiavi è sufficiente per riuscire a scoprire il contenuto di M.
Perciò, col termine *segretezza perfetta* si indica una segretezza la cui probabilità di essere violata non sia maggiore di $\frac{1}{|K|}$. Tale segretezza garantisce che un'informazione sia comunicata in modo tale che la probabilità di scoprirne il contenuto sia uguale a quella di indovinarlo per caso. Si ha quindi che l'unico attacco effettuabile contro un cifrario perfetto è quello a forza bruta.
Al contrario, col termine *segretezza computazionale* si intende che la confidenzialità è garantita solo se le risorse dell'avversario sono limitate.
Un esempio di cifrario che garantisca segretezza perfetta è il One-Time Pad, men-

tre un esempio di cifrario (debole) che garantisca segretezza computazionale è Data Encryption Standard (DES).

1.3.1 OTP

One-Time Pad (o cifrario di Vernam) è un cifrario a sostituzione polialfabetica con le seguenti caratteristiche

1. La chiave è lunga quanto il testo (pad)
2. La chiave è ottenuta in maniera totalmente casuale
3. La chiave è utilizzabile una sola volta (one time)

Come funziona OTP?
Siano

- $\oplus$, l'operatore logico di disgiunzione esclusiva (XOR)
- $m = m_1 m_2 \dots m_n$, un messaggio in chiaro lungo n bit
- $k = k_1 k_2 \dots k_n$, una chiave casuale lunga n bit
- $c = c_1 c_2 \dots c_n$, il crittogramma di m lungo n bit

Per ottenere il crittogramma c si calcoli $c_i = m_i \oplus k_i$.
Per ottenere il messaggio in chiaro m si calcoli $m_i = c_i \oplus k_i$.
C'è qualche differenza tra l'usare l'operatore di AND (congiunzione) o OR (disgiunzione) invece dell'operatore di XOR?
La risposta è sì, in quanto

- L'operatore di AND ha il 75% di possibilità di dare in output 0
- L'operatore di OR ha il 75% di possibilità di dare in output 1
- L'operatore di XOR ha il 50% di possibilità di dare in output 1 o 0

Al contrario di AND e OR, l'uso dell'operatore di XOR garantisce la segretezza perfetta e di seguito ne viene data una prova formale.
Siano

1. C l'insieme dei testi cifrati
2. K l'insieme delle chiavi
3. M l'insieme dei testi in chiaro
4. $E(k, m)$ l'algoritmo di cifratura applicato sul messaggio m con chiave k

Affinchè E sia perfettamente sicuro si richiede che, per ogni crittogramma $c \in C$ e per ogni coppia di messaggi in chiaro $\bar{m}_0, \bar{m}_1 \in M$ di uguale dimensione $|\bar{m}_0| = |\bar{m}_1|$, la seguente equazione sia vera:

$$Pr[E(k, \bar{m}_0) = c] = Pr[E(k, \bar{m}_1) = c] \tag{1.1}$$

dove $Pr[E(k, m) = c]$ è la *probabilità* che il testo in chiaro m cifrato con la chiave k e la funzione di cifratura E sia uguale al crittogramma c.
Nel caso di OTP, supponendo che le chiavi siano casuali, dato $m \in M$ si ha che la funzione di cifratura E garantisce segretezza perfetta in quanto per ogni m:

$$Pr[E(k, m) = c] = \frac{\#k.E(k, m) = c}{|K|} = \frac{1}{|K|} \tag{1.2}$$

Capitolo 2

Tipologie di crittografia

Prima del ventesimo secolo, la crittografia riguardava principalmente schemi linguistici e lessicografici. Da allora l'enfasi si è spostata verso complessità computazionale, teoria dell'informazione, statistica, teoria dei numeri, ecc..
Attualmente, la crittografia è suddividibile in due principali tipologie

- Crittografia simmetrica
- Crittografia asimmetrica

2.1 Simmetrica

Con il termine crittografia simmetrica ci si riferisce a metodi di cifratura per i quali la chiave di decifratura è privata e deve essere concordata preventivamente dalle parti interessate a comunicare. Prima del Giugno 1976, questa era l'unica tipologia di crittografia pubblicamente conosciuta.
Esistono due categorie di cifrari a chiave simmetrica (o privata):

- *Cifrari a flusso*, in cui i bit del testo in chiaro vengono combinati uno alla volta con i bit corrispondenti di un flusso casuale o apparentemente casuale di dati detto chiave di cifratura. Tali cifrari trasformano, con una regola variabile al progredire del testo, uno o pochi bit alla volta del messaggio da cifrare e da decifrare [45]

- *Cifrari a blocchi*, in cui viene contemporaneamente cifrato un intero blocco di bit di lunghezza fissata usando una chiave privata predefinita. Tali cifrari trasformano, con una regola fissa ed uno alla volta, blocchi di messaggio formati da molti bit [45]

I cifrari a flusso risultano i più veloci in quanto la cifratura è facilmente ed immediatamente applicabile al singolo bit, non è infatti richiesto conoscere i bit successivi. Un esempio di cifrario a flusso con chiave completamente casuale è il One-Time Pad (OTP). A differenza degli algoritmi a flusso che cifrano un singolo elemento alla volta, i cifrari a blocchi sono più lenti, in quanto per procedere con la cifratura devono essere a conoscenza di un intero blocco ed applicare un complicato insieme di operazioni. Però, rispetto ai cifrari a flusso con chiave apparentemente casuale, i cifrari a blocchi risultano complessivamente più sicuri. Un esempio di cifrario a blocchi è il Data Encryption Standard (DES).

2.1.1 Pro e contro

Alcuni dei pro e dei contro nell'utilizzo della crittografia simmetrica sono

- Pro

    1. Se il canale usato per condividere la chiave privata è veramente sicuro, allora la sicurezza intrinseca della crittografia simmetrica *può essere* anche *perfetta*
    2. Cifratura e decifratura sono generalmente *processi molto veloci*

- Contro

    1. La chiave privata richiede un *canale sicuro* per essere scambiata
    2. Per ogni possibile interlocutore è richiesta la *memorizzazione di una differente chiave privata*

2.2 Asimmetrica

Storicamente, si ritiene che la data di nascita della crittografia asimmetrica sia il 1976. Infatti, in quell'anno Whitfield Diffie e Martin Hellman proposero la nozione di crittografia a chiave pubblica presentando un protocollo di comunicazione sicura che non richieda lo scambio preliminare della chiave privata. Si ha dunque che la crittografia asimmetrica è caratterizzata dall'uso di due chiavi distinte, una privata e l'altra pubblica. I principali usi della crittografia asimmetrica sono:

- La comunicazione sicura di informazioni (chiavi o quant'altro) su un canale insicuro
- Firme digitali, per autenticare inequivocabilmente il mittente e verificare l'integrità informativa del messaggio
- Certificati digitali, per autenticare il mittente e verificare che la sua chiave pubblica sia valida

2.2.1 Comunicazione sicura: Lo scambio di chiavi DHM

Lo scambio di chiavi Diffie-Hellman-Merkle (DHM) è un protocollo, originalmente ideato da Merkle e successivamente raffinato e proposto da Diffie e Hellman; si tratta di uno dei primi esempi pratici di scambio di chiave pubblica nel contesto della crittografia.
Il protocollo DHM permette di trasmettere un crittogramma computazionalmente sicuro usando un canale insicuro senza che sia avvenuto uno scambio preliminare della chiave di cifratura. [19] Ma come?
Una messaggio cifrato in maniera sicura viene generato nel seguente modo dai corrispondenti

1. Il destinatario possiede una chiave segreta e ne comunica al mittente una trasformazione iniziale (il lucchetto o chiave pubblica)
2. Il mittente ottiene il lucchetto dal destinatario e lo combina con il proprio messaggio segreto, originando così un messaggio cifrato

3. Il messaggio cifrato viene inviato al destinatario che grazie alla propria chiave privata sarà in grado di decifrarlo

4. Le trasformazioni della chiave privata (lucchetto e messaggio cifrato) fanno uso di operazioni non segrete che possono essere effettuate in maniera semplice ma che siano estremamente difficili da invertire

5. La trasformazione da messaggio cifrato a messaggio in chiaro dev'essere, in pratica, computazionalmente possibile solo conoscendo la chiave privata

Non è così possibile per un ascoltatore malizioso invertire le operazioni di trasformazione dei segnali scambiati dai corrispondenti e scoprire il contenuto del messaggio segreto comunicato o la chiave segreta.
In pratica, quanto richiesto per rendere sicuro lo scambio di chiavi appena proposto è un algoritmo in grado di generare le trasformazioni con le caratteristiche descritte nei punti 4 e 5; tale algoritmo verrà in seguito definito *algoritmo di asimmetria*.
L'algoritmo di asimmetria originalmente proposto da Diffie, Hellman e Merkle fa uso di alcune proprietà dei gruppi moltiplicativi di interi modulo p, dove p è un numero primo. Nei prossimi capitoli ci si soffermerà maggiormente sui dettagli matematici di altre tecniche più efficaci.
Si noti come la differenza sostanziale tra le varie tecniche di cifratura a chiave pubblica è proprio l'algoritmo di asimmetria. Per questo motivo è possibile considerare DHM il protocollo di scambio delle chiavi alla base della crittografia asimmetrica moderna.

2.2.2 La firma digitale

Una firma digitale è usata per verificare che:

- Un particolare documento digitale sia autentico ed integro

- Il firmatario sia chi dice di essere

Quindi, una firma digitale permette al destinatario di verificare che l'informazione ricevuta non sia contraffatta ed impedisce al firmatario di ripudiare la propria firma.
Il protocollo alla base del corretto utilizzo di una firma digitale è leggermente diverso

da quello usato per la comunicazione sicura di informazioni. Infatti, nel caso delle firme si ha che:

1. Il firmatario (mittente) possiede una chiave segreta ed una sua trasformazione iniziale (chiave pubblica).

2. Il firmatario usa la chiave segreta per cifrare (firmare) il messaggio da inviare al destinatario. A volte, prima di essere firmato, il messaggio in chiaro viene trasformato tramite una funzione hash crittografica opportunamente concordata dalle parti (vedi sezione 3.3).

3. Il destinatario ottiene dal mittente il messaggio in chiaro, il messaggio firmato e la chiave pubblica.

4. Il destinatario usa la chiave pubblica per verificare che il messaggio in chiaro ed il messaggio firmato coincidano.

5. Come nel caso dello scambio delle chiavi DHM, le trasformazioni della chiave privata fanno uso di operazioni non segrete che possono essere effettuate in maniera semplice ma che siano estremamente difficili da invertire.

Nel caso in cui il messaggio sia confidenziale è opportuno che il messaggio in chiaro ed il messaggio firmato vengano comunicati facendo uso del protocollo DHM e di un algoritmo di asimmetria idoneo.

2.2.3 Il certificato digitale

Un certificato digitale viene usato, tipicamente dai siti web, per autenticare il mittente, verificare che la sua chiave pubblica sia valida e quindi aumentare l'attendibilità nei confronti dei propri utenti o interlocutori.
Normalmente un certificato contiene:

- Informazioni sulla chiave pubblica da certificare

- Informazioni sull'identità dell'utente associato alla chiave pubblica

- La firma digitale di un ente preposto che garantisca per la veridicità dei contenuti del certificato

In tale contesto, esistono due principali tipologie di schemi noti allo stato dell'arte:

- L'*infrastruttura a chiave pubblica* (PKI), in cui il firmatario è un'autorità certificante (CA). Le CA di solito sono compagnie ben conosciute che offrono tale servizio a pagamento.
- La *rete della fiducia* (web of trust), in cui il firmatario è
  - Il proprietario della chiave; in tal caso il certificato è detto autoreferenziale
  - Altri utenti della rete ritenuti fidati da chi verifichi il certificato

A differenza delle firme digitali l'affidabilità dei certificati digitali è fortemente dipendente dall'affidabilità del firmatario, il quale garantisce per la chiave pubblica e l'identità dell'utente certificato.
Esistono due principali categorie di certificati:

- Espliciti
- Impliciti

2.2.3.1 Espliciti

In uno schema di certificazione esplicita la firma dell'ente certificante viene esplicitamente verificata, ottenendo direttamente dall'ente la chiave pubblica di verifica.
Si ha quindi che la chiave pubblica, l'identità dell'ente certificato e la firma digitale dell'ente certificante sono elementi distinti. In questo contesto la generazione della coppia di chiavi ed il certificato di rilascio sono due processi indipendenti. Un utente può infatti presentare una chiave pubblica arbitraria ad un ente certificante per la certificazione.
La diretta conseguenza di tutto ciò è che se l'ente certificante non fosse a conoscenza dell'esistenza del certificato esplicito o tale certificato non fosse valido, allora la verifica del suddetto certificato darebbe esito negativo.
I certificati espliciti sono caratterizzati dall'avere dimensioni relativamente grandi. Ad

esempio, di solito un certificato standard X.509 ha dimensione prossima a 1 KB.
Si noti come quelli espliciti siano gli schemi di certificazione più usati. Un esempio famoso di protocollo basato sulla certificazione esplicita è HTTPS, uno tra i più usati nelle comunicazioni web sicure.

2.2.3.2 Impliciti

In crittografia, i certificati impliciti (o certificati proiettile) contengono le stesse informazioni dei certificati espliciti ma sono caratterizzati dal fatto che non richiedono validazione esplicita della firma dell'ente certificante, in quanto la coppia di chiavi pubblica/privata viene generata insieme all'ente certificante (EC) e non esclusivamente dall'ente certificato (utente).
I certificati proiettile sono più veloci e più piccoli rispetto a quelli espliciti, in quanto si impone che la loro dimensione sia pari a quella della chiave pubblica.
Quando un utente richiede ad un EC un certificato implicito per una chiave pubblica, tale chiave pubblica ed anche la chiave privata sono il risultato casuale di operazioni congiunte di utente e EC. Quindi, a differenza degli schemi a certificazione esplicita, non è permesso ad un utente richiedere un certificato implicito per una chiave pubblica predefinita.
Una diretta conseguenza di quanto descritto è che, una volta che un certificato implicito viene rilasciato, non è possibile ottenere un altro certificato implicito per la stessa chiave pubblica da un EC differente.
Riassumendo, l'utilizzo di uno schema di certificazione implicito risulta ottimale in tutte quelle situazioni in cui si debbano rispettare requisiti dimensionali e prestazionali vincolanti e per cui la verifica esplicita del certificante sia trascurabile, come ad esempio: sitemi di comunicazione e-mail, accessi logici e fisici, transazioni sicure nel constesto dell'Internet of Things, etc..
Più nel dettaglio, i certificati proiettile sono consigliati nei seguenti contesti:

1. Reti di sensori 'ad hoc': in quanto sono ambienti molto limitati nella disponibilità di risorse e possono avere frequenti nuovi utenti entranti, oltre che continui cambi nelle relazioni di fiducia tra gli utenti stessi

2. Comunicazioni postali: in quanto richiedono certificati di dimensione ridotta (francobolli)

3. Comunicazioni tra dispositivi periferici: nel caso di un drive USB collegato ad un PC sicuro, il PC potrebbe usare uno schema di sicurezza che certifichi in tempi rapidi il drive per l'uso temporaneo

2.2.4 Pro e contro

Alcuni dei pro e dei contro nell'utilizzo della crittografia asimmetrica sono

- Pro

    1. Non è richiesto un canale sicuro per lo scambio delle chiavi
    2. Non è richiesta la memorizzazione delle chiavi private di tutti i potenziali interlocutori, esse infatti possono essere concordate in maniera sicura nelle prime fasi della comunicazione

- Contro

    1. Non sono note tecniche crittografiche asimmetriche che siano perfettamente sicure
    2. Cifratura e decifratura sono generalmente più lente dei propri corrispettivi nella crittografia simmetrica

2.3 Ibrida

La crittografia ibrida è frutto dell'unione tra crittografia simmetrica ed asimmetrica. Più nel dettaglio

1. Fa uso della cifratura asimmetrica per stabilire una chiave di sessione condivisa
2. Usa la cifratura simmetrica per scambiare i messaggi cifrati con la chiave di sessione

Il fatto di dover creare ogni volta una chiave di sessione permette di risparmiare spazio, non è infatti richiesto memorizzare una chiave privata per ogni possibile interlocutore. L'utilizzo della cifratura simmetrica per lo scambio dei messaggi aumenta le performance di invio e ricezione delle informazioni.

2.3.1 Esempi

Alcuni esempi famosi di crittografia ibrida sono

- Pretty Good Privacy (PGP); crittosistema con lo scopo di permettere privacy ed autenticazione per lo scambio di dati nella rete
- Transport Secure Layer (TSL); protocollo crittografico alla base della sicurezza del protocollo HTTPS

Capitolo 3

Crittosistemi asimmetrici

Col termine crittosistema ci si riferisce ad un insieme di algoritmi richiesti per implementare un metodo di cifratura e decifratura. Di solito, un crittosistema è composto da

1. Un algoritmo di generazione della chiave
2. Un algoritmo di cifratura
3. Un algoritmo di decifratura

La nozione formale di crittosistema è la seguente

Definizione 1. *Un crittosistema è una quintupla* (P, C, K, E, D) *tale che*

1. *P,C e K sono insiemi finiti per cui*
   - *P è lo spazio dei messaggi in chiaro (non cifrati)*
   - *C è lo spazio dei messaggi cifrati*
   - *K è lo spazio delle chiavi*
2. $E = \{E_k | k \in K\}$ *è una famiglia di funzioni* $E_k : P \to C$ *usate per la cifratura*
   $D = \{D_k | k \in K\}$ *è una famiglia di funzioni* $D_k : C \to P$ *usate per la decifratura*
3. *Per ogni chiave* $k_1 \in K$ *esiste una chiave* $k_2 \in K$ *tale per cui per ogni* $p \in P$*:*

$$D_{k_2}(E_{k_1}(p)) = p \tag{3.1}$$

*Un crittosistema è detto **simmetrico** se $k_1 = k_2$ o quantomeno se k_2 può essere ottenuto da k_1 in modo relativamente facile.*

*Un crittosistema è detto **asimmetrico** se $k_1 \neq k_2$ e se in pratica è computazionalmente difficile calcolare k_2 a partire da k_1. In questo contesto, si ha che k_2 è la chiave privata e k_1 è la chiave pubblica.*

Inoltre, un crittosistema è detto semanticamente sicuro quando un attaccante non è in grado di distinguere tra di loro due crittogrammi anche conoscendo o scegliendo i messaggi in chiaro (non cifrati) corrispondenti.
In questo capitolo verranno descritti i seguenti crittosistemi asimmetrici:

- *RSA*, usato principalmente per comunicare in modo confidenziale
- *ElGamal*, usato principalmente per scambiare in modo sicuro chiavi private su un canale insicuro
- *DSA*, uno standard per firme digitali
- *ECQV*, per la creazione e l'uso di certificati impliciti

3.1 RSA

RSA è uno dei primi crittosistemi a chiave pubblica ad essere stato largamente usato per *trasmissioni sicure di dati* e prende il nome dai suoi inventori:

- Ron Rivest
- Adi Shamir
- Leonard Adleman

i quali lo pubblicarono per primi nel 1977. L'algoritmo di asimmetria di RSA si basa sulla difficoltà pratica del fattorizzare il prodotto di due numeri primi sufficientemente grandi. RSA funziona nel modo seguente:

1. Generazione delle chiavi

Si scelgono due numeri primi distinti p e q

Si calcola $n = p \cdot q$

Si calcola $\phi(n) = (p-1)(q-1)$, dove ϕ è la funzione toziente di Eulero che calcola il numero di coprimi minori di n

Si sceglie un intero e in modo che sia coprimo con $\phi(n)$ e che $1 < e < \phi(n)$

Si trova $d \equiv e^{-1} \mod \phi(n)$: l'inverso moltiplicativo di e modulo $\phi(n)$

La chiave pubblica è così formata dalla coppia $< e, n >$

La chiave privata è d

2. Cifatura

Si ottiene la chiave pubblica del destinatario e cioè la coppia di interi e e n

Usando un opportuno schema, si codifica il messaggio M da trasferire in un intero m opportuno, con $0 \leq m < n$, in modo che il massimo comun divisore di m e n sia 1

Si calcola il testo cifrato c nel modo seguente: $c \equiv m^e \mod n$. Questa operazione è efficientemente effettuabile mediante l'algoritmo di esponenziazione modulare per quadrati

3. Decifatura

Si ottiene c dal mittente

Si calcola $c^d \equiv m^{ed} \equiv m \mod n$

Dato m è possibile ottenere M usando la funzione inversa dello schema usato per codificarlo in m

Il crittosistema appena mostrato è corretto, ma perchè?
Dato che per costruzione, per qualche intero positivo h, è vero che

$$\begin{aligned} ed &\equiv 1 \mod \phi(qp) \\ &= h \cdot (p-1) \cdot (q-1) + 1 \end{aligned} \tag{3.2}$$

per il piccolo teorema di Fermat (th. 3) si ha che

$$\begin{aligned} m^{ed} &= m^{ed-1} \cdot m \\ &= m^{h\cdot(p-1)\cdot(q-1)} \cdot m \\ &\equiv 1 \cdot m \mod p, q \end{aligned} \tag{3.3}$$

Quindi, è vero che $m^{ed} \equiv m \mod qp$.

3.1.1 Schema da adottare per la codifica preliminare del messaggio

Se lo schema adottato per la codifica preliminare del messaggio M fosse la funzione identità, allora RSA sarebbe vulnerabile ai seguenti attacchi

- Usando il metodo di Coppersmith [46], messaggi cifrati mediante piccoli valori dell'esponente e (es.: $e = 3$) per $m < n^{\frac{1}{e}}$ possono essere facilmente decifrati trovando tutte le radici intere del messaggio cifrato c
- Dato che RSA usa un algoritmo di cifratura deterministico, un attaccante potrebbe eseguire con successo un attacco con testo in chiaro scelto. Come? Cifrando messaggi simili con la chiave pubblica fino a trovarne due che abbiano lo stesso crittogramma
- Dato che il prodotto tra due o più potenze aventi gli stessi esponenti è uguale ad una potenza avente per base il prodotto delle basi e per esponente lo stesso esponente, si ha che il prodotto di due crittogrammi RSA è uguale alla cifratura del prodotto dei rispettivi testi in chiaro. La proprietà appena descritta permette di effettuare un attacco su testo in chiaro predefinto

Per evitare i problemi appena elencati, lo schema adottato prevede l'aggiunta ad m di un'imbottitura casuale prima della cifratura. Tale imbottitura garantisce che m non sia troppo piccolo o che abbia caratteristiche che rendano insicuro il suo crittogramma.
Dato che lo schema di imbottitura aggiunge bit a m, la dimensione del messaggio di partenza M dev'essere relativamente ridotto o suddiviso in blocchi di dimensione predefinita.

Attualmente, Optimal Asymmetric Encryption Padding (OAEP) risulta essere lo schema di imbottitura più robusto agli attacchi.
Riassumendo, l'algoritmo di asimmetria e quindi la sicurezza intrinseca di RSA si basa su due problemi matematici simili:

- La fattorizzazione
- Il problema RSA; cioè essere in grado di trovare il valore di $\sqrt[e]{c} \mod n$ senza conoscere i fattori primi di n

Non si conosce ancora se i problema sopracitati siano equivalenti o meno.

3.2 ElGamal

L'algoritmo ElGamal fu descritto nell'articolo [20, A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms] da Taher Elgamal nel 1985 e oggigiorno viene principalmente *usato per accordare le chiavi private* in un contesto di crittografia ibrida.
L'algoritmo di asimmetria di ElGamal si basa sulla difficoltà pratica del calcolare il logaritmo discreto in un particolare gruppo ciclico [48]. ElGamal funziona nel modo seguente:

1. Generazione delle chiavi

   Si genera una descrizione efficiente di un gruppo ciclico G di ordine q e con generatore $g < q$ scelto casualmente. Dove q è la potenza di un numero primo sufficientemente grande.

   Si sceglie casualmente un intero x tale che $1 < x < q - 1$

   Si calcola $h = g^x$

   La chiave pubblica è così formata da h e le descrizioni di G, q, g

   La chiave privata è x

2. Cifatura

Si ottiene la chiave pubblica del destinatario e cioè h e le descrizioni di G, q, g

Si sceglie casualmente un intero y tale che $1 < y < q - 1$ e si calcola $c_1 = g^y$

Si calcola $s = h^y$

Si mappa il messaggio segreto M in un suo corrispettivo elemento m all'interno del gruppo G

Si calcola $c_2 = m \cdot s$

Si invia al destinatario il crittogramma composto da c_1 e c_2

3. Decifatura

Si ottengono c_1 e c_2 dal mittente

Si calcola $s = c_1^x$

Sfruttando il teorema di Lagrange, si calcola facilmente $s^{-1} = c_1^{q-x}$ e cioè l'inverso moltiplicativo di s all'interno del gruppo G

Si calcola $m = c_2 \cdot s^{-1}$ e da m si risale facilmente al messaggio in chiaro M

L'algoritmo di decifratura è in grado di risalire a m in quanto

$$\begin{aligned} c_2 \cdot s^{-1} &= m \cdot h^y \cdot g^{-xy} \\ &= m \cdot g^{xy} \cdot g^{-xy} \\ &= m \end{aligned} \tag{3.4}$$

Riassumendo, l'algoritmo di asimmetria e quindi la sicurezza intrinseca di ElGamal si basa sul seguente problema:

- Il problema del logaritmo discreto in un gruppo ciclico

3.3 DSA

Il Digital Signature Algorithm è una variante di ElGamal ed è uno standard FIPS per le *firme digitali*.
L'algoritmo di asimmetria di DSA si basa sulla difficoltà pratica del calcolare il logaritmo discreto in un particolare gruppo ciclico. DSA funziona nel modo seguente:

1. Generazione delle chiavi

   Si sceglie H, una funzione hash crittografica approvata (es.: SHA-2)

   Si sceglie un numero primo q di Q bit, dove Q dev'essere minore od uguale alla lunghezza dell'output di H

   Si sceglie un numero primo p di P bit, dove $p - 1$ è un multiplo di q e P è un multiplo di 64

   Si scelgono h e g per cui $1 < h < p - 1$, $g = h^{\frac{p-1}{q}} \mod p$ e $g > 1$. Dato che la maggior parte dei valori di h danno un g accettabile, calcolare un g opportuno è un'operazione relativamente semplice

   Si sceglie casualmente un intero x per cui $0 < x < q$

   Si calcola $y = g^x \mod p$

   La chiave pubblica è l'ennupla $< y, p, q, g >$

   La chiave privata è x

2. Firma

   Per ogni messaggio da firmare m si genera casualmente un intero k per cui $0 < k < q$

   Si calcola $r = (g^k \mod p) \mod q$. Nel caso in cui $r = 0$, si sceglie casualmente un altro k

   Si calcola $s = k^{-1}(H(m) + xr) \mod q$. Nel caso in cui $s = 0$, si sceglie casualmente un altro k

   La firma per m è la coppia $< r, s >$

3. Verifica

   Si ottengono dal mittente la chiave pubblica $< y, p, q, g >$ ed il messaggio m

   Si controlla che $0 < r < q$ e $0 < s < q$; in caso contrario la firma non è valida per m

   Si calcola $w = s^{-1} \mod q$

Si calcola $u_1 = H(m) \cdot w \mod q$

Si calcola $u_2 = r \cdot w \mod q$

Si calcola $v = (g^{u_1} y^{u_2} \mod p) \mod q$

Se $v = r$ allora la firma è valida per m

L'algoritmo appena descritto è corretto nel contesto della firma digitale, in quanto

$$\begin{aligned} r &= (g^k \mod p) \mod q \\ &= (g^{u_1} \cdot y^{u_2} \mod p) \mod q \\ &= v \end{aligned} \tag{3.5}$$

Infatti, se $g = h^{\frac{p-1}{q}} \mod p$, allora per il priccolo teorema di Fermat (th. 3) è vero che

$$\begin{aligned} g^q &\equiv h^{p-1} \mod p \\ &\equiv 1 \mod p \end{aligned} \tag{3.6}$$

da cui segue che g ha ordine q modulo p [47]. Quindi

$$\begin{aligned} k &\equiv H(m) \cdot s^{-1} + xrs^{-1} \mod q \\ &\equiv H(m) \cdot w + xrw \mod q \end{aligned} \tag{3.7}$$

e conseguentemente

$$\begin{aligned} g^k &\equiv g^{H(m)\cdot w} \cdot g^{xrw} \mod p \\ &\equiv g^{H(m)\cdot w} \cdot y^{rw} \mod p \\ &\equiv g^{u_1} \cdot y^{u_2} \mod p \end{aligned} \tag{3.8}$$

Un algoritmo efficiente per il calcolo dell'inverso modulare $s^{-1} \mod q$ è l'algoritmo esteso di Euclide, in alternativa è possibile sfruttare il piccolo teorema di Fermat calcolando $s^{q-2} \mod q$.
Infine, si noti che i valori di n e m rappresentano la misura principale della forza crittografica della chiave, più sono grandi e più la chiave è robusta.
Riassumendo, l'algoritmo di asimmetria e quindi la sicurezza intrinseca di DSA si basa sul seguente problema:

- Il problema del logaritmo discreto in un gruppo ciclico

3.4 ECQV

Elliptic Curve Qu-Vanstone (ECQV) è un algoritmo crittografico asimmetrico per la generazione di certificati impliciti e basato sulla Crittografia a Curve Ellittiche (ECC). ECC venne indipendentemente proposta per la prima volta nel 1985 da Neal Koblitz e Victor S. Miller e si basa sul problema del logaritmo discreto in sottogruppi ciclici di curve ellittiche su campi finiti. Questa tecnica permette di ridurre drasticamente la dimensione di chiavi di cifratura e certificati, mantenendo livelli di sicurezza equivalenti a quelli dei suoi predecessori. Per maggiori dettagli su ECC e su cosa sia una curva ellittica, si veda la sezione 4.2.2.
In seguito, col termine EC si farà riferimento all'ente certificante e col termine Utente si farà riferimento all'utente che fa richiesta di un certificato implicito a EC. Quanto segue descrive il funzionamento di ECQV:

1. Inizializzazione

   EC sceglie una curva ellittica verificabile E definita sul campo generato da un'opportuna potenza q di un numero primo, per cui:

   - G è il generatore dei punti base
   - n è l'ordine di G, generalmente un numero primo

   Usando una propria chiave privata d_{EC}, con $1 \leq d_{EC} \leq n-1$, EC genera $Q_{EC} = d_{EC}G$

   Q_{EC} è la chiave pubblica di EC, mentre E è l'insieme dei suoi parametri di dominio

2. Richiesta del certificato

   Utente ottiene la chiave pubblica Q_{EC} di EC e l'insieme dei suoi parametri di dominio E

   Utente sceglie casualmente un intero $1 \leq k_U \leq n-1$ ed invia $R_U = k_U G$ a EC

3. Generazione del certificato

   EC sceglie casualmente un intero $1 \leq k \leq n-1$

EC calcola $P_U = R_U + kG$ e cioè i dati di ricostruzione della chiave pubblica di Utente

EC calcola $Cert_U = Encode(P_U, ID_U)$, dove *Encode* è una funzione di codifica opportuna e ID_U sono le informazioni che identifichino univocamente Utente

EC calcola $e = H_n(Cert_U)$, dove H_n è una funzione hash crittografica opportuna (es.: SHA-3)

EC calcola $r = ek + d_{EC} \mod n$ e cioè la chiave privata di ricostruzione dei dati

EC invia la coppia $< r, Cert_U >$ a Utente

4. Generazione delle chiavi

Utente conosce la chiave pubblica Q_{EC} di EC, l'insieme dei suoi parametri di dominio E e l'intero k_U che ha preventivamente scelto ed ottiene da EC il certificato $Cert_U$ e la chiave privata del certificato r

Utente estrae P_U da $Cert_U$ usando la funzione inversa di *Encode*

Utente calcola $e = H_n(Cert_U)$, dove H_n è una funzione hash crittografica opportunamente concordata con EC (es.: SHA-3)

La chiave privata di Utente è $d_U = ek_U + r \mod n$

La chiave pubblica di Utente è $Q_U = eP_U + Q_{EC}$

Comunemente, come funzione *Encode* per la codifica di $< P_U, ID_U >$ si usa uno dei seguenti schemi [21]:

- Campi di dimensione finita: una semplice codifica minimalista che privilegia un consumo efficiente della banda. Il certificato è composto da campi di dimensione finita. Si richiede che almeno uno di questi campi contenga P_U

- Schema di codifica minimale ASN.1: a differenza dello schema precedente richiede l'utilizzo di più campi

- Codifica ASN.1 conforme a X.509: richiede di includere sufficienti informazioni per poter permettere al certificato ECQV di essere ri-codificato come certificato standard X.509

Riassumendo, l'algoritmo di asimmetria e quindi la sicurezza intrinseca di ECQV si basa sul seguente problema:

- Il problema del logaritmo discreto in un sottogruppo ciclico di una curva ellittica su un campo finito

3.5 Cosa accomuna i crittosistemi appena descritti

Tralasciando per un momento il fatto che sono tutti crittosistemi asimmetrici, si può notare che sono raggruppabili in due grandi famiglie principali

- Crittosistemi basati sul problema della fattorizzazione: RSA
- Crittosistemi basati sul problema del logaritmo discreto: ElGamal, DSA, ECQV

Entrambe le famiglie appena enunciate sono caratterizzate dall'uso estensivo di alcune proprietà dei *numeri primi* e richiedono la generazione di numeri primi sufficientemente grandi per il loro corretto funzionamento ed utilizzo.

Capitolo 4

Cosa rende sicura la crittografia asimmetrica?

Come descritto nei precedenti capitoli, l'algoritmo di asimmetria è la vera base della sicurezza di un crittosistema asimmetrico.
Tutti gli algoritmi di asimmetria proposti in questa tesi dipendono da particolari problemi della Teoria dei Numeri con complessità computazionali peculiari. Quanto si evince è che Teoria dei Numeri e Crittografia siano inestricabilmente collegate.

4.1 La teoria della complessità computazionale

La teoria della complessità computazionale è una branca della teoria della computazione nell'Informatica Teorica che si focalizza nella classificazione di problemi computazionali in base alla loro difficoltà intrinseca.
Un problema computazionale è inteso come un obiettivo che sia, in via di principio, facilmente ottenibile da un calcolatore con sufficienti risorse a sua disposizione. In altre parole, quanto enunciato è equivalente a dire che un problema computazionale può essere risolto mediante l'uso meccanico di passaggi matematici predefiniti (un algoritmo).
Un problema è detto essere intrinsecamente difficile se la sua soluzione richiede una quantità significativa di risorse a prescindere dall'algoritmo usato. La teoria della complessità computazionale formalizza questa intuizione introducendo modelli matematici

per studiare i problemi computazionali e quantificare le risorse richieste per risolverli. Le misure di complessità solitamente adottate sono

- Tempo
- Spazio

4.1.1 Crittografia asimmetrica e teoria della complessità computazionale

Quanto richiesto da un crittosistema asimmetrico è un problema relativamente facile da creare ma computazionalmente complicato da risolvere senza le giuste informazioni. Nel contesto della crittografia asimmetrica, con '*problema facile da creare*' si intende che la funzione di cifratura/verifica di un crittosistema deve avere complessità polinomiale in tempo nella dimensione dell'input. Se m è la codifica numerica di un messaggio in chiaro, allora la sua dimensione in bit è esattamente $\lfloor \log_2 m \rfloor + 1$. Si ha quindi che la complessità in tempo della funzione di cifratura/verifica dev'essere asintotica a $O(\log^c m)$, per qualche costante $c > 0$.
Invece, con '*problema computazionalmente complicato da risolvere senza le giuste informazioni*' si intende che, usando una macchina deterministica, la funzione di decifratura/firma deve avere complessità polinomiale in tempo nella dimensione dell'input se e solo se si conosce la chiave di decifratura. In caso non si conosca la chiave privata, per un calcolatore deterministico non-quantistico, l'algoritmo di decifratura/firma deve avere complessità almeno esponenziale ($g \geq 1$) o sub-esponenziale ($0 < g < 1$) nella dimensione di m e quindi dev'essere dell'ordine di $\Omega(2^{\Theta(\log^g m)})$ per qualche costante $g > 0$.

4.2 La teoria dei numeri

La teoria dei numeri è una branca della matematica pura che si occupa principalmente dello studio dei numeri interi.
Per comprendere meglio il ruolo centrale giocato dai numeri primi nel contesto della teoria dei numeri, si enuncia il teorema fondamentale dell'aritmetica

Teorema 1 (Teorema fondamentale dell'aritmetica)**.** *Ogni numero naturale maggiore di* 1 *o è un numero primo o si può esprimere come prodotto di numeri primi. Tale rappresentazione è unica, se si prescinde dall'ordine in cui compaiono i fattori.*

Ma, cos'è un numero primo?

Definizione 2 (Numero primo)**.** *Un primo è un numero naturale maggiore di* 1 *che non ha divisori positivi ad eccezione di* 1 *e se stesso.*

Se 1 fosse un numero primo, allora il teorema fondamentale dell'aritmetica non potrebbe essere vero, in quanto la fattorizzazione in numeri primi non sarebbe unica:

$$\begin{aligned} 110 &= 1 \cdot 2 \cdot 5 \cdot 11 \\ &= 1 \cdot 1 \cdot 2 \cdot 5 \cdot 11 \\ &= 1 \cdot ... \cdot 1 \cdot 2 \cdot 5 \cdot 11 \end{aligned} \tag{4.1}$$

Un qualsiasi fattore di un numero intero n è detto *non triviale* se è diverso da 1 e da n.
Nelle prossime sottosezioni verranno presentati i due principali problemi matematici utilizzati dagli algoritmi di asimmetria proposti nel capitolo precedente.

4.2.1 Il problema della fattorizzazione

In teoria dei numeri, la fattorizzazione di interi è la decomposizione di un numero composto nel prodotto di interi più piccoli. Se questi interi più piccoli dovessero essere numeri primi, allora il processo appena descritto si chiamerebbe fattorizzazione in numeri primi.
Grazie al teorema fondamentale dell'aritmetica si può affermare che per ogni intero composto esiste una fattorizzazione in primi che sia unica. Ad esempio

$$9699690 = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \tag{4.2}$$

Attualmente, non sono noti algoritmi non-quantistici efficienti per la fattorizzazione di interi arbitrariamente grandi. Inoltre, tutt'oggi non si sa molto sulla classe di complessità di questo problema, in quanto non si è a conoscenza del fatto che possa esistere o meno un algoritmo che lo risolva con complessità polinomiale in tempo.
Allo stato dell'arte attuale, gli algoritmi di fattorizzazione più efficienti sono

- Per input $n \leq 10^{100}$ è il Quadratic Sieve (QS) che ha la seguente complessità *euristica* sub-esponenziale nella dimensione dell'input: $O(e^{\sqrt{\ln n \cdot \ln \ln n}\cdot(1+o(1))})$
- Per input $n > 10^{100}$ è il General Number Field Sieve (GNFS) che ha la seguente complessità *euristica* sub-esponenziale nella dimensione dell'input: $O(e^{(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}(\sqrt[3]{\frac{64}{9}}+o(1))})$

Un crittosistema che si basa sul problema della fattorizzazione è RSA.

4.2.1.1 RSA vs Fattorizzazione

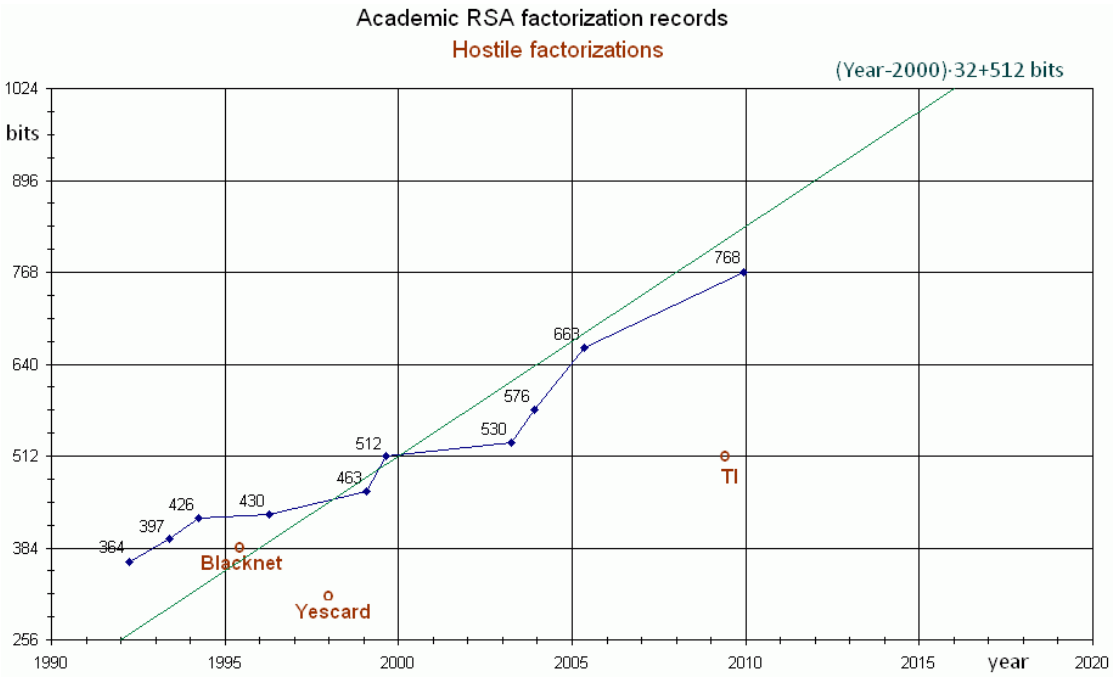

Figura 4.1: Record accademici di fattorizzazione di chiavi pubbliche RSA

Al 2009, il record accademico di fattorizzazione è una chiave pubblica RSA di 768 bit. [22]
Attualmente, chiavi RSA a 512 bit non sono più ritenute essere computazionalmente sicure. [23]
Al 2015, la minaccia più valida per i sistemi che usino RSA a 1024 bit non è la fattorizzazione della chiave pubblica, ma piuttosto la vulnerabilità delle infrastrutture in-

formatiche ad attacchi fisici o di altra natura (hacking, certificati digitali, ingegneria sociale, etc..). Quindi si ritiene che, almeno per qualche decennio, chiavi RSA a 2048 bit (o di dimensione anche maggiore) siano sufficientemente sicure da reali minacce di fattorizzazione della chiave pubblica.

4.2.2 Il problema del logaritmo discreto e le curve ellittiche

Definizione 3 (Logaritmo discreto in un gruppo ciclico)**.** *Dati $a, g, k, q, p, n \in \mathbb{N}$, si ha che k è detto logaritmo discreto di a rispetto alla base g modulo $q = p^n$, per qualche primo p e qualche intero $n > 0$, se*

$$a \equiv g^k \mod q \tag{4.3}$$

Riuscire a trovare k che soddisfi tale equazione equivale a risolvere il problema del logaritmo discreto di a rispetto alla base g modulo q.

Ad esempio, 3 è detto logaritmo discreto di 15 rispetto alla base 7 modulo 41 in quanto $15 \equiv 7^3 \mod 41$.
Come nel caso della fattorizzazione, non si conoscono metodi efficienti e generici per risolvere il problema del logaritmo discreto su computer non-quantistici.
Allo stato dell'arte attuale, gli algoritmi più efficienti per risolvere il problema del logaritmo per qualunque p sono sub-esponenziali

- *Number Field Sieve for Discrete Logarithms*: usato quando p è grande rispetto a q, ha complessità in tempo $O(e^{(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}(\sqrt[3]{\frac{64}{9}}+o(1))})$

- *Function Field Sieve*: usato quando q è grande rispetto a p, ha complessità in tempo $O(e^{(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}(\sqrt[3]{\frac{32}{9}}+o(1))})$

- *Joux*: usato quando q è grande rispetto a p, per qualche $c > 0$ ha complessità in tempo $O(e^{(\ln n)^{\frac{1}{4}+\varepsilon}(\ln \ln n)^{\frac{3}{4}-\varepsilon}(c+o(1))})$

- *Number Field Sieve in High Degree*: usato quando q e p hanno dimensione simile, ha complessità in tempo $O(e^{(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}(c+o(1))})$ con $c > 0$

Dunque, uno degli algoritmi sopracitati potrebbe essere usato per attaccare i crittosistemi ElGamal e DSA.
Si è però notato che, nel caso in cui il problema del logaritmo discreto sia definito in opportuni sottogruppi ciclici di curve ellittiche su campi finiti, allora gli algoritmi sub-esponenziali appena nominati perdono di efficacia. Si dà quindi la seguente definizione

Definizione 4 (Logaritmo discreto in un sottogruppo ciclico di una curva ellittica su un campo finito)**.** *Sia $\mathbb{F}_q$ un gruppo ciclico definito da $q = p^n$, per qualche primo p e qualche intero $n > 0$. Sia E una curva ellittica definita su $\mathbb{F}_q$. Dato $B \in E$, risolvere il problema del logaritmo discreto su E in base B significa trovare (nel caso esista) un intero z tale per cui $zB = P$, dato un punto $P \in E$. [25]*

Un noto esempio di utilizzo del problema appena descritto è ECC. Ma, cos'è una curva ellittica definita su $\mathbb{F}_q$?

Definizione 5. *Una curva ellittica definita su $\mathbb{F}_q$ è un'equazione a due incognite $x, y \in \mathbb{N}$ e a coefficienti $a, b \in \mathbb{N}$ per cui*

$$y^2 \equiv x^3 + ax + b \mod q \tag{4.4}$$

Allo stato dell'arte attuale, gli algoritmi più efficienti per risolvere il problema del logaritmo discreto in un sottogruppo ciclico di una curva ellittica su un campo finito sono esponenziali:

- *Algoritmo Pohlig-Hellman*: con complessità in tempo $O(\sqrt{q})$
- *Algoritmo Rho di Pollard*: con complessità in tempo $O(\sqrt{q})$

Il fatto che gli algoritmi più efficienti non siano sub-esponenziali permette, ai crittosistemi basati su ECC come ECQV, di poter usare chiavi pubbliche più piccole e robuste.

4.2.2.1 ECC vs RSA/DSA

Dato che l'algoritmo di fattorizzazione e quello di risoluzione del problema del logaritmo discreto in un gruppo ciclico hanno complessità asintotiche pressochè uguali,

la dimensione computazionalmente sicura per una chiave pubblica di ElGamal e DSA è pressochè la stessa di una chiave pubblica RSA. Questo non si può dire di ECC.

Symmetric Key Size (bits)	RSA and Diffie-Hellman Key Size (bits)	Elliptic Curve Key Size (bits)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Table 1: NIST Recommended Key Sizes

Figura 4.2: Confronto tra dimensioni e sicurezza di chiavi simmetriche, chiavi pubbliche RSA/DH e chiavi pubbliche ECC

Al 2009, il record accademico di attacco ad uno schema ECC conclusosi con successo è su una chiave pubblica di 112 bit. [26]

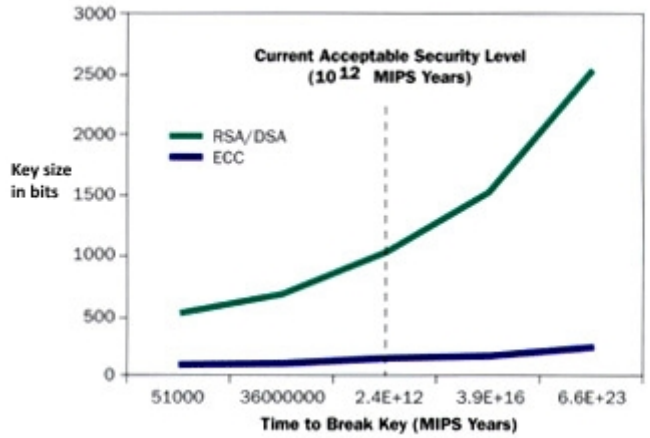

Figura 4.3: Livelli di sicurezza accettabili per chiavi pubbliche RSA/DSA e ECC

Capitolo 5

Crittografia e calcolo quantistico

I calcolatori quantistici sono differenti dai calcolatori elettronici digitali e binari basati sui transistor. Infatti i computer quantistici si basano più sulla fisica quantistica che sulla fisica elettrica.
Un calcolatore digitale richiede che i dati siano codificati in bit e cioè unità di memoria che possono assumere solo due stati: o 0, o 1. I bit vengono poi memorizzati sotto forma di energia elettrica o magnetica, generalmente su dischi magnetici o all'interno di condensatori.
Al contrario, un calcolatore quantistico usa i qubit, i quali sono sistemi meccanico-quantistici a due stati. La differenza principali con i bit è che per le leggi della meccanica quantistica i qubit possono essere in più stati sovrapposti contemporaneamente.
Al 2016, lo sviluppo di calcolatori quantistici è ai suoi albori. [28] [29]

5.1 Gli algoritmi di Shor

Nel 1994, Peter Shor presentò all'interno dell'articolo [27, Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer] due algoritmi randomizzati per risolvere, in *tempo polinomiale* nella dimensione dell'input e su un computer quantistico ipotetico, il problema della fattorizzazione e del logaritmo discreto.
Contrariamente a quanto ci si potrebbe aspettare, i crittosistemi basati su ECC risultano

essere più vulnerabili rispetto a RSA ad un attacco basato sull'algoritmo di Shor. Questo perchè la dimensione delle chiavi pubbliche usate in ECC è significativamente minore rispetto a quelle usate in RSA.
Nel 2003, J. Proos e C. Zalka mostrarono nel loro articolo [30, Shor's discrete logarithm quantum algorithm for elliptic curves] come un computer quantistico richieda approssimativamente 4096 qubit per rompere una chiave pubblica RSA di 2048 bit, mentre per rompere una chiave pubblica ECC equivalentemente sicura di 224 bit lo stesso tipo di calcolatore richieda tra i 1300 ed i 1600 qubit.
Nel 2015, la NSA ha annunciato di star pianificando per il breve futuro lo sviluppo e la diffusione di un nuovo crittosistema standardizzato per calcolatori non-quantistici che sia resistente agli attacchi quantistici. [31]

5.2 Crittografia post-quantistica

Con il termine crittografia post-quantistica ci si riferisce ad algoritmi crittografici che siano ritenuti sufficientemente sicuri contro attacchi condotti con computer quantistici.
Al contrario dei crittosistemi asimmetrici, la maggior parte degli algoritmi crittografici simmetrici sono considerati essere relativamente sicuri da attacchi quantistici.
Allo stato attuale l'unica minaccia relativamente valida alla crittografia simmetrica è l'algoritmo di Grover [41], il quale permette di condurre attacchi quantistici in maniera estremamente efficiente. Dato che lo speed-up garantito dall'algoritmo di Grover è solamente quadratico, raddoppiare la dimensione della chiave simmetrica è però sufficiente a mitigare efficacemente questo genere di attacchi.
Alcune delle tecniche crittografiche post-quantistiche asimmetriche ritenute più valide sono:

- Crittografia basata sui Lattici: i cui algoritmi di asimmetria si basano sul problema del lattice definito su un campo F generato dalla potenza di un numero primo opportuno. Un esempio famoso è il problema di machine learning dell'imparare dagli errori, anche noto come Learning with Errors (LWE). [42]

- Crittografia multivariata: i cui algoritmi di asimmetria si basano sul trovare soluzioni di polinomi a più variabili definiti su un campo F generato dalla potenza di un numero primo opportuno (es.: MQQ-ENC [43]).

- Crittografia su curve ellittiche isogene supersingolari: i cui algoritmi di asimmetria si basano su una variante di ECC che fa uso di curve ellittiche isogene e supersingolari.

Ognuna delle tecniche pocanzi elencate si basa, per il suo corretto funzionamento, sull'utilizzo di numeri primi opportunamente scelti.
Si fa notare che c'è una grossa differenza tra *crittografia post-quantistica* e *crittografia quantistica*. Infatti la crittografia post-quantistica si occupa dello studio di problemi matematici che siano resistenti ad attacchi condotti con computer quantistici, mentre la crittografia quantistica si occupa delle tecniche che permettano di usare i fenomeni quantistici per ottenere segretezza ed identificare eventuali intercettazioni delle informazioni segrete.
Nel contesto della crittografia quantistica, lo schema Quantum Key Distribution (QKD) per condividere una chiave privata è in grado di garantire segretezza perfetta (come One-Time Pad) anche su un canale insicuro [44]. QDK non è però una tecnica crittografica asimmetrica, in quanto non viene usata alcuna chiave pubblica ma soltanto alcune leggi della fisica quantistica.

Capitolo 6

Test di primalità

Nei capitoli precedenti si è dimostrata la rilevante importanza dei numeri primi nel contesto della crittografia asimmetrica.
Per la generazione di numeri primi opportuni è necessario un algoritmo efficiente di verifica della primalità di un numero. Un test di primalità è un algoritmo che, applicato ad un numero intero, ha lo scopo di determinare se sia primo o meno. Non va confuso con un algoritmo di fattorizzazione, che invece ha lo scopo di determinare i fattori primi di un numero: quest'ultima operazione è infatti generalmente più lunga e complessa.
Non è però escluso che fattorizzare determinati numeri risulti estremamente efficiente e che sia quindi equivalente a testarne la primalità in tempo ottimo. Ad esempio, l'algoritmo AKS fa uso di algoritmi di fattorizzazione preliminari per identificare potenze perfette di numeri primi.
Esistono due principali tipologie di test di primalità

- Probabilistici
- Deterministici

Sia i test probabilistici che quelli deterministici possono essere euristici. Ma cosa vuol dire euristico?
I test euristici sono test che sembrano funzionare bene in pratica, ma la cui correttezza o complessità asintotica (nel caso pessimo) non sono state rigorosamente dimostrate.

L'algoritmo ECPP è un esempio di test deterministico ed euristicamente polinomiale, infatti non si sa con certezza quale sia la sua complessità nel caso pessimo. [15]

6.1 Test probabilistici

Si supponga che T sia un algoritmo che prenda in input un qualche intero x. Se per lo stesso input x l'output di $T(x)$ cambiasse, allora si direbbe che T è un algoritmo non deterministico.
Nel caso di algoritmi probabilistici (detti anche randomizzati) il calcolo delle probabilità è applicato non tanto ai dati in ingresso, quanto piuttosto ai dati in uscita. Gli algoritmi probabilistici sono algoritmi non deterministici la cui computazione dipende dai valori prodotti da un generatore di numeri casuali. [17]
Molti test di primalità famosi sono probabilistici. Un classico test di primalità randomizzato non dà mai conferma del fatto che un numero sia primo e quindi è possibile che qualche numero composto passi il test, tali numeri composti sono detti pseudoprimi. Ripetendo il test più volte, la probabilità di errore del test può essere però resa piccola a piacere. [15]
Di seguito verranno descritti alcuni dei test di primalità probabilistici più famosi:

- Il test di Fermat
- Il test di Miller-Rabin

6.1.1 Test di Fermat

Il test di Fermat prende il nome da Pierre de Fermat, un avvocato e matematico francese del 1600.
Il funzionamento del test di Fermat si basa sul seguente teorema

Teorema 2 (V1 - Piccolo teorema di Fermat)**.** *Se p è un numero primo, allora per ogni intero a è vero che*

$$a^p \equiv a \mod p \tag{6.1}$$

Questo significa che se si prende un qualunque numero a, lo si moltiplica per se stesso p volte e si sottrae a, il risultato è divisibile per p. Tale teorema è spesso espresso nella forma equivalente:

Teorema 3 (V2 - Piccolo teorema di Fermat)**.** *Se p è primo e a è un intero coprimo con p, allora per ogni intero a è vero che*

$$a^{p-1} \equiv 1 \mod p \tag{6.2}$$

Va notato che la prima versione del teorema è in un certo senso più generale, in quanto è valida per numeri interi arbitrari, come 0 o multipli di p, che invece non rientrano nelle ipotesi della seconda versione. [2]
Il test di Fermat funziona nel modo seguente

1. Si sceglie un valore opportuno per k e si pone $i = 0$
2. Si sceglie casualmente un intero a tale che $2 < a < n - 2$
3. Si pone $i = i + 1$
4. Si verifica che eq.(6.2) sia rispettata
    - Se $a^{n-1} \not\equiv 1 \mod n$, allora n è un numero composto
    - Se $i < k$, allora si riparte dal punto 2
    - Altrimenti si termina il test dicendo che n è probabilmente primo

Usando l'esponenziazione modulare per quadrati e l'algoritmo di Fürer per la moltiplicazione, la complessità dell'algoritmo è $\tilde{O}(k \cdot log^2 n)$ dove n è il numero di cui testare la primalità e k è il numero di volte per cui ripetere la verifica di eq.(6.2).
Si può dimostrare che la probabilità che l'algoritmo appena descritto sbagli, confondendo un numero composto per primo, è approssimativamente $\frac{1}{2^k}$. [32]
Un numero n è detto *pseudoprimo di Fermat per a* se n è un numero composto che supera il test di Fermat per la base a e cioè se rispetta la relazione di congruenza definita in eq.(6.2).
Ad esempio, 341 è uno pseudoprimo di Fermat per 2 in quanto $2^{340} \equiv 1 \mod 341$.
Esistono dei numeri composti che sono pseudoprimi di Fermat per ogni base, questi numeri sono detti pseudoprimi di Carmichael.

6.1.1.1 I numeri di Carmichael

I numeri di Carmichael rivestono un'importanza notevole in questo contesto perché passano in ogni caso il test di Fermat, pur essendo composti. La loro esistenza impedisce di utilizzare questo test per certificare con sicurezza la primalità di un numero.
In teoria dei numeri, un numero di Carmichael è un intero positivo composto n che soddisfa la seguente congruenza per ogni intero b:

$$b^n \equiv b \mod n \tag{6.3}$$

Una caratterizzazione dei numeri di Carmichael è stata fornita nel 1899 da Korselt [4] :

Teorema 4 (Criterio di Korselt)**.** *Un intero positivo composto n è un numero di Carmichael se e solo se è privo di quadrati e per ogni divisore primo p di n è vero che* $p - 1$ *divide* $n - 1$.

Usando il criterio di Korselt si può facilmente verificare che 561 è un numero di Carmichael. Infatti, $561 = 3 \cdot 11 \cdot 17$ è privo di quadrati e $\frac{560}{2} = 280$, $\frac{560}{10} = 56$ e $\frac{560}{16} = 35$.
Un'ulteriore proprietà dei numeri di Carmichael che deriva dal criterio di Korselt è la seguente:

Teorema 5. *Sia n un numero di Carmichael composto da k fattori primi distinti* p_i *con* $1 \leq i \leq k$. *Per ogni i è sempre vero che* $p_i - 1$ *divide* $(\prod_{j \neq i} p_j) - 1$, *con* $1 \leq j \leq k$.

6.1.1.2 Un'utile variante del test di Fermat

Un semplice corollario che deriva da th.(2) è il seguente:

Teorema 6. *Dato* $a \geq 1$ *con* $a \in \mathbb{N}$ *e p numero primo, per il th.(2) è vero che*

$$(a^{p^\alpha} \mod p^\alpha) \equiv a \mod p \tag{6.4}$$

Infatti, nel caso in cui $a = 1$ si ha banalmente il piccolo teorema di Fermat. Nel caso in cui $a = 2$, per qualche intero $\aleph > 0$ si ha che

$$\begin{aligned} a^{p^2} &\equiv a^{pp} \mod p^2 \\ &= \aleph \cdot p^2 + a^{pp} \\ &\equiv a^{pp} \mod p \\ &\equiv a^p \mod p \\ &\equiv a \mod p \end{aligned} \tag{6.5}$$

segue per induzione che

$$\begin{aligned} a^{p^\alpha} &\equiv a^{p^{\alpha-1}} \mod p \\ &\equiv a \mod p \end{aligned} \tag{6.6}$$

Ciò implica che è possibile effettuare correttamente il test di Fermat anche usando qualsiasi potenza di p come esponente e modulo. Tale corollario verrà usato in seguito per garantire il corretto funzionamento degli algoritmi euristici proposti negli ultimi capitoli.

6.1.2 Il test di Miller-Rabin

Il test di Miller-Rabin deve il suo nome a Gary L. Miller e Michael O. Rabin e può essere visto come una derivazione diretta del test di Fermat.
Si consideri th.(3), se p è un numero primo si ha che per ogni a opportuno

$$\begin{aligned} 0 &\equiv a^{p-1} - 1 \mod p \\ &\equiv (a^{\frac{p-1}{2}} - 1)(a^{\frac{p-1}{2}} + 1) \mod p \end{aligned} \tag{6.7}$$

Se $p - 1 = 2^s \cdot d$, allora è possibile ripetere s volte il procedimento di fattorizzazione usato in eq.(6.7). Dunque, per th.(3) e per il Lemma di Euclide [35] una delle seguenti equivalenze dev'essere vera

$$a^d - 1 \equiv 0 \mod p \tag{6.8}$$

oppure

$$a^{2^r \cdot d} + 1 \equiv 0 \mod p \tag{6.9}$$

per qualche intero r tale che $0 \leq r \leq s-1$.
Inoltre, se $a^d - 1 \not\equiv 0 \mod n$ e $a^{2^r \cdot d} - 1 \equiv 0 \mod n$ per $0 < r \leq s-1$, allora

$$a^{2^r \cdot d} - 1 = (a^{2^{r-1} \cdot d} - 1) \cdot (a^{2^{r-1} \cdot d} + 1) \tag{6.10}$$

e quindi n è composto in quanto

- o $1 \leq GCD(n, (a^{2^{r-1} \cdot d} - 1)) \leq n$
- o $1 \leq GCD(n, (a^{2^{r-1} \cdot d} + 1)) \leq n$

per ulteriori dettagli matematici si legga [27, Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer].
Perciò il test di Miller-Rabin funziona nel modo seguente

1. Si sceglie un valore opportuno per k e si pone $i = 0$
2. Si calcolano d e s per cui $n - 1 = 2^s \cdot d$
3. Si sceglie casualmente un intero a tale che $2 < a < n - 2$
4. Si pone $i = i + 1$
5. Se $a^d - 1 \equiv 0 \mod n$, allora si va al punto 8
6. Se $a^{2^r \cdot d} - 1 \equiv 0 \mod n$ per qualche intero r tale che $0 < r \leq s-1$, allora n è composto ed il test termina
7. Se per ogni r, con $0 \leq r \leq s-1$, si ha che $a^{2^r \cdot d} + 1 \not\equiv 0 \mod n$, allora n è composto ed il test termina
8. - Se $i < k$, allora si riparte dal punto 3
   - Altrimenti si termina il test dicendo che n è probabilmente primo

Usando l'esponenziazione modulare per quadrati e l'algoritmo di Fürer per la moltiplicazione, la complessità dell'algoritmo è $\tilde{O}(k \cdot log^2 n)$.
Nel 1980, Michael O. Rabin dimostrò che la probabilità che l'algoritmo appena descritto sbagli, confondendo un numero composto per primo, è approssimativamente $\frac{1}{4^k}$ per

ogni n composto (anche se si tratta di un numero di Carmichael). [33]
Inoltre, per n sufficientemente grande, Damgård, Landrock and Pomerance dimostrarono che, nel caso medio, la probabilità che un numero composto sia dichiarato probabilmente primo è significativamente più piccola di $\frac{1}{4^k}$. [34]
Dato che k è una costante, la complessità asintotica del test di Miller-Rabin è dunque $\tilde{O}(log^2 n)$.
Un numero n è detto *pseudoprimo forte per a* se n è un numero composto che supera il test Miller-Rabin per la base a.
Ad esempio, 2047 è uno pseudoprimo forte per 2 in quanto $2^{1023} \equiv 1 \mod 2047$.
L'insieme degli pseudoprimi forti è contenuto all'interno dell'insieme degli pseudoprimi di Fermat. Però, diversamente dagli pseudoprimi di Fermat, non esistono dei numeri composti che siano pseudoprimi forti per ogni base. Questa proprietà rende il test di Miller-Rabin molto più *affidabile* del test di Fermat.

6.2 Test deterministici

Il determinismo dal punto di vista ontologico indica il dominio della necessità causale in senso assoluto e nega quindi nel contempo l'esistenza del caso.
Un test deterministico è un algoritmo che, dato un paticolare input, produce sempre lo stesso output. I test probabilistici non sono deterministici.
Di seguito verranno descritti alcuni dei test di primalità probabilistici più famosi, come

- AKS
- ECPP
- Il test di Miller

6.2.1 AKS

Il test AKS fu pubblicato nel 2002 da M. Agrawal, N. Kayal e N. Saxena nell'articolo [36, PRIMES is in P] e si tratta del primo esempio conosciuto allo stato dell'arte di un algortimo *deterministico e non euristico* per verificare la primalità di un numero

in tempo polinomiale nella dimensione dell'input. La correttezza di AKS non dipende da alcuna ipotesi non provata e si basa sulla seguente generalizzazione del Piccolo Teorema di Fermat:

Teorema 7. *Un intero $n \geq 2$ è primo se e solo se è vera la seguente relazione di congruenza tra polinomi, per qualche $r > 0$ e per qualche a coprimo rispetto a n*

$$(x + a)^n \equiv (x^n + a) \mod x^r - 1, n \tag{6.11}$$

con x variabile libera.
Equivalentemente la relazione appena descritta può essere rappresentata mediante la seguente equazione

$$(x + a)^n - (x^n + a) = (x^r - 1)g + nf \tag{6.12}$$

per qualche polinomio f e g.

Quanto dimostrato dagli inventori dell'algoritmo AKS è che r può essere opportunamente ed efficientemente scelto per essere polinomiale nella dimensione di n, permettendo così una verifica polinomiale e deterministica della primalità di n.
La complessità asintotica in tempo della prima versione di AKS è $\tilde{O}(\log^{12} n)$. Nel 2005, Carl Pomerance e H. W. Lenstra Jr. dimostrarono la correttezza di una variante di AKS con complessità in tempo $\tilde{O}(\log^6 n)$. [37]
L'algoritmo AKS ha enorme importanza teorica ma *non viene usato in pratica*, in quanto le costanti computazionali nascoste dalla notazione asintotica della complessità rendono l'algoritmo in pratica sconveniente da utilizzare rispetto ai test probabilistici o rispetto a ECPP.

6.2.2 ECPP

Elliptic Curve Primality Proving (ECPP) è un test di primalità corretto per ogni intero in input. ECPP nasce nel 1986 da un'idea di S. Goldwasser e J. Kilian ed attualmente è in pratica il test di primalità deterministico più veloce conosciuto allo stato dell'arte.
La correttezza di ECPP si basa sul seguente teorema

Teorema 8. *Sia n un intero positivo e E l'insieme di punti definito dalla seguente curva ellittica*

$$y^2 = x^3 + ax + b \mod n \tag{6.13}$$

con $a, b \in \mathbb{N}$.
Sia m l'ordine del gruppo E. [47]
Se

- *Esiste un numero primo q tale da essere anche un fattore di m per cui* $q > (\sqrt[4]{n}+1)^2$
- *Esiste un* $P \in E$ *tale per cui*
  1. $mP = 0$
  2. $\frac{m}{q}P$ *è definito e diverso da* 0

allora n è primo

Da un punto di vista computazionale, la parte più complicata dell'algoritmo è quella di contare i punti $P \in E$ e trovarne uno che certifichi la primalità di n.
Dato che l'algoritmo richiede la scelta di un E opportuno e dato che non si sa se questo sia possibile in tempo polinomiale, nel caso pessimo non si conosce quale possa essere l'effettiva complessità asintotica di ECPP. In tal senso si dice che la complessità euristica asintotica di ECPP è $\tilde{O}(\log^5 n)$.
Esiste una variante basata su argomentazioni euristiche e proposta da J. O. Shallit che riduce la complessità di ECPP a $\tilde{O}(\log^4 n)$. [38]

6.2.3 Il test di Miller

Nel 1796 G. L. Miller pubblicò l'articolo [39, Riemann's Hypothesis and Tests for Primality] col quale presentò un nuovo test di primalità euristico e deterministico che ora porta il suo nome e da cui derivò il test di Miller-Rabin.
Il test di Miller basa la propria correttezza sull'ipotesi generalizzata di Riemann (GRH), che se dimostrata proverebbe che il seguente algoritmo è vero

1. Si pone $a = 1$

2. Si calcolano d e s per cui $n - 1 = 2^s \cdot d$
3. Se $a > 2\log^2 n$, allora n è sicuramente primo
4. Si pone $a = a + 1$
5. Se $a^d - 1 \equiv 0 \mod n$, allora si va al punto 3
6. Se $a^{2^r \cdot d} - 1 \equiv 0 \mod n$ per qualche intero r tale che $0 < r \leq s - 1$, allora n è composto ed il test termina
7. Se per ogni r, con $0 \leq r \leq s-1$, si ha che $a^{2^r \cdot d} + 1 \not\equiv 0 \mod n$, allora n è composto ed il test termina
8. Altrimenti si riparte dal punto 3

Sostanzialmente il test di Miller è come un test di Miller-Rabin effettuato su al più $2\log^2 n$ basi diverse scelte deterministicamente.
Usando l'esponenziazione modulare per quadrati e l'algoritmo di Fürer per la moltiplicazione, la complessità dell'algoritmo appena descritto è $\tilde{O}(log^4 n)$.

6.3 Determinismo vs probabilismo

Complessivamente, i migliori test di primalità probabilistici risultano molto più efficienti dei migliori test di primalità deterministici.
Di seguito un breve riassunto

- AKS è deterministico e non euristico, ha complessità $\tilde{O}(\log^6 n)$
- ECPP è deterministico ed in grado di certificare la primalità di un numero in tempo euristico $\tilde{O}(\log^5 n)$ o addirittura $\tilde{O}(\log^4 n)$
- Il test di Miller è deterministico con complessità $\tilde{O}(\log^4 n)$, ma la sua correttezza si basa sull'ipotesi di Riemann generalizzata ed è quindi euristico
- Il test di Fermat è probabilistico con complessità $\tilde{O}(\log^2 n)$

- Il test di Miller-Rabin è probabilistico con complessità $\tilde{O}(\log^2 n)$

Date le elevate performance richieste ai crittosistemi asimmetrici per garantire comunicazioni in tempo reale con basso consumo di risorse, si ha che il test di primalità più usato in pratica nel contesto della crittografia a chiave pubblica è il test di Miller-Rabin. Si ricorda che il test di Miller-Rabin è estremamente performante, non ha pseudoprimi per tutte le basi e la sua probabilità di errore può essere ridotta a piacere.
Quanto si può dedurre dalle precedenti considerazioni è che trovare nuovi test di primalità che siano più efficienti ed affidabili permetterebbe di migliorare le prestazioni e l'affidabilità di molti crittosistemi asimmetrici noti allo stato dell'arte.

Capitolo 7

Una nuova tecnica per migliorare l'affidabilità del test di Fermat fattorizzando gli pseudoprimi di Carmichael

Gli pseudoprimi sono di rilevante importanza nel contesto della crittografia a chiave pubblica in quanto la loro presenza *potrebbe compromettere sicurezza ed efficacia* dei crittosistemi asimmetrici che facciano uso di test di primalità probabilistici.
Una delle pecche principali del test di Fermat è il fatto che per certe famiglie di input sia completamente inefficace. Gli pseudoprimi di Carmichael sono infatti quegli interi in grado di imbrogliare *sempre* il test di Fermat. 6.1.1.1
Quindi, trovare *un metodo deterministico ed efficiente per fattorizzare i numeri di Carmichael permetterebbe di migliorare l'affidabilità del test di Fermat* rendendola comparabile con quella del test di Miller-Rabin. Ecco perchè in questo capitolo viene mostrato un algoritmo deterministico in grado di fattorizzare tutti gli pseudoprimi di Carmichael, euristicamente in tempo $\tilde{O}(\log^3 n)$, poi modificato sfruttando alcune proprietà del test di Miller *per ottenere, nel prossimo capitolo, un nuovo test di primalità deterministico* ed euristico con complessità $\tilde{O}(\log^2 n)$ e la cui probabilità di errore tende a 0 con n che tende ad infinito.

Per comprendere il funzionamento dei nuovi algoritmi proposti in questi ultimi capitoli, verranno innanzitutto descritte le seguenti tecniche:

- L'algoritmo di Euclide
- L'esponenziazione modulare per quadrati
- Il metodo di moltiplicazione Russo

7.1 L'algoritmo di Euclide

L'algoritmo di Euclide (GCD) è un algoritmo per trovare il massimo comun divisore tra due numeri interi ed è uno degli algoritmi più antichi che si conoscano.
L'algoritmo di Euclide non richiede la fattorizzazione completa dei due interi di cui cercare il divisore comune, di seguito ne viene presentata una semplice implementazione polinomiale che esegue al più $o(\log_2 \min(a, b))$ operazioni di modulo.

Algoritmo 7.1 : Algoritmo di Euclide

```
int GCD( int a, int b )
{
        int r;
        while( b!=0 ) // repeat until b is 0
        {
                r = a%b;
                a=b;
                b=r;
        }
        return a;
}
```

Il calcolo del massimo comun divisore è un passo essenziale in molti algoritmi di fattorizzazione di interi, come l'algoritmo Rho di Pollard, l'algoritmo di Shor, il metodo di fattorizzazione di Fermat (vedi appendice D) e la fattorizzazione a curve ellittiche di

Lenstra.
Esistono alcune implementazioni efficienti dell'algoritmo di Euclide che, tramite l'uso dell'algoritmo di Fürer per la moltiplicazione, hanno complessità asintotica $\tilde{O}(\log \min(a, b))$. [3]

7.2 L'esponenziazione modulare per quadrati

Si definiscono $b, c, p \in \mathbb{N}^+$.
Ogni operazione di esponenziazione $a^b \mod p$ è scomponibile nelle seguenti sotto-operazioni:

- Se b è dispari, allora

$$\begin{aligned} a^b &\equiv a^{2\frac{b-1}{2}+1} \mod p \\ &\equiv a \cdot (a^{\frac{b-1}{2}}) \cdot (a^{\frac{b-1}{2}}) \mod p \end{aligned} \tag{7.1}$$

- Se b è pari, allora

$$\begin{aligned} a^b &\equiv a^{2\frac{b}{2}} \mod p \\ &\equiv (a^{\frac{b-1}{2}}) \cdot (a^{\frac{b-1}{2}}) \mod p \end{aligned} \tag{7.2}$$

Supponendo di conoscere già il valore di $(a^{\frac{b-1}{2}})$, si ha quindi che in eq.(7.1) si devono effettuare due moltiplicazioni, mentre in eq.(7.2) si deve effettuare una sola moltiplicazione. Perciò, nel caso pessimo in cui $b = 2^d - 1$, per qualche $d \in \mathbb{N}^+$, si ha che il numero di moltiplicazioni richieste è esattamente $2 \log_2 b$.
Il risultato è che usando la tecnica appena descritta e l'algoritmo di Fürer per la moltiplicazione è possibile eseguire l'operazione di esponenziazione modulare in tempo pari a $\tilde{O}(\log p \cdot \log b)$, questo perchè tutti gli interi usati per le operazioni di moltiplicazione sono al più grandi quanto $p - 1$ per via delle operazioni di modulo.
Di seguito viene presentata una possibile implementazione C++ dell'esponenziazione modulare per quadrati.

7.2 L'esponenziazione modulare per quadrati

Algoritmo 7.2 : Esponenziazione modulare per quadrati

```
int modPower( int a, int b, int p )
{
        switch ( b )
        {
                case 0:
                        return 1;
                case 1:
                        return a%p;
                default:
                        int remainder = 1;
                        while ( b > 1 ) // executed o( log b ) times
                        {
                                if ( IsOdd(b) ) // b is odd
                                {
                                        remainder = (remainder*a)%p;
                                }
                                a = (a*a)%p;
                                b = floor(b/2);
                        }
                        return (remainder*a)%p;
        }
}
```

7.3 Il metodo di moltiplicazione Russo

Si supponga di voler moltiplicare tra loro due interi a e b. Il metodo di moltiplicazione Russo equivale a rappresentare l'operazione $a \cdot b$ nel modo seguente

$$\begin{aligned} a \cdot b &= a \cdot \left(2\lfloor\frac{b}{2}\rfloor + (b \mod 2)\right) \\ &= a \cdot (b \mod 2) + 2a \cdot \left(2\lfloor\frac{b}{4}\rfloor + (\lfloor\frac{b}{2}\rfloor \mod 2)\right) \\ &= 2^{\lfloor\log_2 b\rfloor}a + \sum_{i=0}^{\lfloor\log_2 b\rfloor-1}\left(2^i a \cdot (\lfloor\frac{b}{2^i}\rfloor \mod 2)\right) \\ &= a \cdot \left(2^{\lfloor\log_2 b\rfloor} + \Sigma\left(b, \lfloor\log_2 b\rfloor\right)\right) \end{aligned} \tag{7.3}$$

dove $\Sigma(y,z) = \sum_{i=0}^{z-1}\left(2^i \cdot (\lfloor\frac{y}{2^i}\rfloor \mod 2)\right)$.
Nel caso della moltiplicazione modulare, tale scomposizione permette di ridurre al minimo possibili overflow dovuti ad eventuali limiti imposti alla rappresentazione di un numero, come nel caso dei tipi numerici nativi di C e C++.
Con il metodo Russo, se U è il massimo numero rappresentabile, allora è possibile elevare correttamente al quadrato modulare qualsiasi intero $c \leq \left\lfloor\frac{U}{2}\right\rfloor$.
Di seguito viene mostrata un'implementazione C++ della moltiplicazione Russa modulare che fa uso della ricorsione di coda.

Algoritmo 7.3 : Metodo di moltiplicazione Russo

```
int modMultiplier( int x, int y, int m, int tail=0, int deep=0 )
{
	if ( x == 0 || y == 0 )
	{
		return tail;
	}
	if ( x == 1 )
	{
		return (y + tail)%m;
	}
```

```
        if ( y == 1 )
        {
                return (x + tail)%m;
        }

        deep = deep + 1;
        if ( IsOdd(y) ) // y is odd
        {
                tail = (tail+x)%m;
        }

        int head = (x*2)%m;
        return modMultiplier( head, floor(y/2), m, tail, ++deep ); //
            tail recursion
}
```

Dato che la funzione *modMultiplier* viene ricorsivamente chiamata $\log_2 y$ volte e dato che, per via delle operazioni di modulo, tutti gli interi usati per le operazioni di moltiplicazione sono al più grandi quanto $m - 1$, facendo uso dell'algoritmo di Fürer per la moltiplicazione, l'implementazione appena mostrata della moltiplicazione modulare ha complessità pari a $\tilde{O}(\log m \cdot \log y)$.

N.B. La variabile *deep* è di per se inutile per ottenere il risultato finale della funzione così come è stata descritta, viene però usata per rendere più chiari i prossimi ragionamenti.

7.4 L'esponenziazione modulare, la moltiplicazione Russa e l'algoritmo di Euclide

Si definisca la funzione $\bar{\Omega}(x)$ come segue

$$\bar{\Omega}(x) = \lfloor \log_2 x \rfloor \tag{7.4}$$

dato che 2 è il più piccolo numero primo, $\bar{\Omega}(x)$ conta il massimo numero di diversi fattori primi possibili per l'input x.
Si consideri il test di Fermat implementato usando le due tecniche appena viste e cioè l'esponenziazione modulare per quadrati e la moltiplicazione Russa modulare. Tale test, in seguito chiamato test Russo di Fermat, esegue più volte l'operazione di elevamento al quadrato di un intero x usando la moltiplicazione Russa:

$$x^2 \equiv x \cdot \left(2^{\bar{\Omega}(x)} + \Sigma\left(x, \bar{\Omega}(x)\right)\right) \mod n \tag{7.5}$$

Ritornando alla funzione *modMultiplier* descritta poco fa, se gli si desse come input (x, x, n), allora sarebbe sempre vero che al variare di $deep$

$$head \equiv x \cdot 2^{deep} \mod n \tag{7.6}$$

$$tail \equiv x \cdot \Sigma\left(x, deep\right) \mod n \tag{7.7}$$

Quanto *empiricamente* osservato è che esiste sempre un'opportuna base a per cui

$$1 < GCD\left(n, x \cdot \left(2^k + \Sigma\left(x, k\right)\right)\right) < n \tag{7.8}$$

per qualche intero k tale che $0 < k < \bar{\Omega}(x)$.
Dunque, per ottenere un fattore di n basta modificare il test Russo di Fermat in modo che, per ogni nuovo valore assunto dalla variabile $head$, si calcoli

$$GCD\left(n, head + tail\right) \tag{7.9}$$

Dato che per costruzione x è coprimo rispetto a n, si ha che

$$GCD\left(n, x \cdot \left(2^k + \Sigma\left(x, k\right)\right)\right) = GCD\left(n, 2^k + \Sigma\left(x, k\right)\right) \tag{7.10}$$

Di seguito viene mostrata un'implementazione C++ di $findFactor$ e cioè la variante appena descritta del metodo Russo di moltiplicazione.

Algoritmo 7.4 : Tecnica per trovare un fattore non triviale di un numero di Carmichael

```
bool findFactor( int x, int n ) // Based on the Russian multiplication
{
```

```
        int pow_of_p = 1; // If x will be equal to 1, then pow_of_p will
            be exactly: 2^Omega(x) mod n
        // int head = x; // For the Russian multiplication: If x will be
            equal to 1, then head will be exactly: x*2^Omega(x) mod n = x*
            pow_of_p mod n
        int sigma = 0; // If x will be equal to 1, then sigma will be
            exactly: Sigma(x,Omega(x)) mod n
        // int tail = 0; // For the Russian multiplication: If x will be
            equal to 1, then tail will be exactly: x*Sigma(x,Omega(x)) mod
             n = x*sigma mod n
        while ( x > 1 )
        {
                if ( IsOdd(x) ) // x is odd
                {
                        sigma = (pow_of_p+sigma)%n;
                        // tail = (head + tail)%n;
                }
                pow_of_p = (pow_of_p*2)%n;
                // head = (head*2)%n;
                x = floor(x/2);

                if ( computeGCD( (pow_of_p+sigma)%n ) )
                {
                        return true;
                }
        }
        // The Russian multiplication is head + tail = x*2^Omega(x) mod n
             + x*Sigma(x,Omega(x)) mod n = (x*x) mod n
        return false;
}

bool computeGCD( int a, int n )
{
```

```
        int gcd = GCD( n, a ); // Euclidean algorithm
        if ( gcd>1 && gcd<n )
        {
                return true;
        }
        return false;
}
```

La complessità di $findFactor$ è $\tilde{O}(\log x \cdot \log n)$, in quanto esegue al più $\log_2 x$ volte l'algoritmo di Euclide su input al più grandi quanto $n-1$.
Nella prossima sezione verrà discusso con argomentazioni euristiche un algoritmo in grado di trovare, in tempo polinomiale, un x opportuno che garantisca una fattorizzazione deterministica ed euristicamente polinomiale di un generico numero di Carmichael.

7.5 Una nuova tecnica per fattorizzare i numeri di Carmichael

Quanto osservato empiricamente è che nel caso in cui n sia un un numero di Carmichael, usando th.(6) insieme al test di Fermat effettuato mediante l'esponenziazione modulare per quadrati e combinato con il nuovo algoritmo $findFactor$, si è in grado di trovare facilmente un'opportuna base a che permette la fattorizzazione di n in tempo polinomiale nella dimensione di n.
Il nuovo test di Fermat così ottenuto verifica la seguente congruenza

$$(a^{n^\alpha} \mod n^\alpha) \equiv a \mod n \tag{7.11}$$

eseguendo l'algoritmo $findFactor$ sul risultato di ogni elevamento al quadrato.
Di seguito verranno descritti due diversi algoritmi polinomiali di scelta della base a al variare dell'esponente α e per ognuno di essi verrà mostrata una possibile implementazione C++.

7.5.1 Prima versione

Scegliendo $\alpha = \bar{\Omega}(n)$, per tutti i numeri di Carmichael fino a $10^{21} \approx 2^{70}$ si è visto che:

- Per $a = n^{\bar{\Omega}(n)} \pm 2$, il nuovo algoritmo trova *sempre* un fattore non triviale di n
- Per $a = n^{\bar{\Omega}(n)} \pm 1$, il nuovo algoritmo nella maggior parte dei casi non trova un fattore non triviale di n

Dato che $n^{\bar{\Omega}(n)} + 2 \equiv 2 \mod n^{\bar{\Omega}(n)}$ si ha che $a = 2$ è una buona scelta di base in questo contesto. Si può però osservare che in questo modo le prime $2\bar{\Omega}(\bar{\Omega}(n))$ operazioni di elevamento al quadrato di a risultano inutili ai fini della fattorizzazione mediante l'algoritmo di Euclide, in quanto in numeri di Carmichael sono dispari e

$$2^{2^{2\bar{\Omega}(\bar{\Omega}(n))}} \leq 2^{\bar{\Omega}(n)^2} \leq n^{\bar{\Omega}(n)} \tag{7.12}$$

Dunque, un'ulteriore ottimizzazione nella scelta della base è porre $a = 2^{\bar{\Omega}(n^{\bar{\Omega}(n)})} = 2^{\bar{\Omega}(n)^2}$. Anche in questo caso, il test risulta efficace dando sempre una risposta corretta.

7.5.1.1 Il codice C++

Il codice proposto è un template C++ pensato per l'uso con la libreria open source NTL [13], senza la quale non sarebbe possibile generare gli $n^{\bar{\Omega}(n)}$ necessari per i test fino e oltre $n = 10^{21} \approx 2^{70}$.

Algoritmo 7.5 : *Header* del 1mo algoritmo di fattorizzazione per numeri di Carmichael

```
#include <math.h>
#include<iostream>
using namespace std;

// #define LOG2 log(2)
#define LOG2 0.693147180559945

#ifndef __Sovrano
#define __Sovrano
```

```
template <class INT, class FLOAT> // INT is an integer type. FLOAT is a
    floating point type.
// NTL::ZZ is suggested as INT type, double is suggested as FLOAT type
class Sovrano
{
        public:
                Sovrano( ); // O( 1 ) constructor
                bool findCarmichaelFactor( const INT& n ); // O~( log^5 n )
                INT getFactor( ) const { return factor; } // O( 1 )
        protected:
                void initialize( const INT& n ); // O~( log n )
                void modPower( INT base, INT exponent, const INT& modulus )
                    ; // O~( log exponent * log modulus * log real_number )
                bool findFactor( INT y ); // O~( log y * log real_number )
                bool computeGCD( const INT& m ); // O~( log MIN(real_number
                    ,m) )
                bool hasFoundValidFactor( ) const { return has_found_factor
                    ; } //O( 1 )
        private:
                static const INT int0, int1, int2, int3, int4; // Improves
                    template performance when using non standard types as
                    INT. Prevents constructing new INT objects each time.
                INT remainder; // modPower local variables
                INT gcd; // computeGCD local variables
                INT pow_of_p, tail; // findFactor local variables
                bool has_found_factor; // initialize variables
                FLOAT max_omega;
                INT factor, real_number, pow_number, exponent, base; //
                    initialize variables
};

#endif
```

Algoritmo 7.6 : *Body* del 1mo algoritmo di fattorizzazione per numeri di Carmichael

```
#include "Sovrano.h"

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int0(0); // Equal to 0. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int1(1); // Equal to 1. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int2(2); // Equal to 2. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int3(3); // Equal to 3. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int4(4); // Equal to 4. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
```

```
Sovrano<INT, FLOAT>::Sovrano( ):factor(int1),has_found_factor(false){};

template <class INT, class FLOAT>
void Sovrano<INT, FLOAT>::initialize( const INT& n ) // O~( log n )
{
	real_number = n;
	has_found_factor=false;
	factor=int1; // factor=1
	max_omega = floor(log(n)/LOG2); // floor( log_2 n ) -> max. count
	    of possible prime factors of n
	pow_number = power( n, max_omega ); // O~( log n )
	base = power( int2, max_omega*max_omega ); // O~( log n )
	exponent = pow_number; // For the Fermat Test
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::findCarmichaelFactor( const INT& n ) // O~(
   log^5 n )
{
	initialize( n ); // O~( log n )

	/* Trial division */
	if ( computeGCD( int3 ) ) // O( 1 )
	{
		return true; // n is multiple of 3
	}

	modPower( base, exponent, pow_number ); // O~( log^5 n )
	return hasFoundValidFactor();
}

template <class INT, class FLOAT>
```

```
void Sovrano<INT, FLOAT>::modPower( INT base, INT exponent, const INT&
    modulus ) // O~( log exponent * log modulus * log real_number )
{
       if ( exponent <= int1 ) // exponent <= 1
       {
              return;
       }

       remainder = int1;
       while ( exponent > int1 ) // while factor > 1 -> O( log2 exponent
           )
       {
              if ( IsOdd( exponent ) )
              {
                     remainder = MulMod( base, remainder, modulus ); // O
                         ~( log base )
                     if ( findFactor(remainder) ) // O~( log remainder *
                         log real_number )
                     {
                            return;
                     }
              }

              base = MulMod(base,base,modulus); // O~( log base )
              if ( findFactor(base) ) // O~( log base * log real_number )
              {
                     return;
              }

              exponent /= int2; // exponent = floor(exponent/2) ->
                  exponent is always an integer
       }
```

```
        base = MulMod(base,remainder,modulus); // O~( log base )
        if ( findFactor(base) ) // O~( log base * log real_number )
        {
                return;
        }
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::findFactor( INT y ) // O~( log y * log
    real_number )
{
        pow_of_p=int1, tail=int0;
        while ( y > int1 ) // y > 1
        {
                if ( IsOdd(y) ) // y is odd
                {
                        tail = AddMod(pow_of_p,tail,real_number);
                }
                pow_of_p = MulMod(pow_of_p,int2,real_number);
                y /= int2; // y = floor(y/2) -> y is always an integer

                if ( computeGCD( AddMod(pow_of_p,tail,real_number) ) ) // O
                    ~( log MIN(real_number,(pow_of_p+tail)%real_number) )
                    -> Euclidean GCD algorithm
                {
                        return true;
                }
        }
        return false;
}

template <class INT, class FLOAT>
```

```
bool Sovrano<INT, FLOAT>::computeGCD( const INT& m ) // O~( log MIN(
    real_number,m) )
{
        gcd = GCD( real_number, m ); // O( log MIN(real_number,m) )
        if ( gcd>int1 && gcd<real_number ) // gcd>1 && gcd<real_number
        {
                factor=gcd;
                has_found_factor=true;
                return true;
        }
        return false;
}
```

7.5.1.2 La complessità asintotica

La complessità in tempo dell'algoritmo di fattorizzazione appena descritto è $\tilde{O}(\log^5 n)$, in quanto:

- **hasFoundValidFactor**() ha complessità $O(1)$
- **computeGCD**(m) esegue l'algoritmo di Euclide una volta e ha quindi complessità $\tilde{O}(\log min(n,m)) = \tilde{O}(\log n)$, in quanto $min(n,m) \leq n$.
- **findFactor**(y) ha complessità $\tilde{O}(\log y \cdot \log n)$
- **modPower**(base, exponent, modulus) ha complessità $\tilde{O}(\log exponent \cdot \log modulus \cdot \log n)$
- **initialize**(n) ha complessità $\tilde{O}(\log n)$
- **findCarmichaelFactor**(n) esegue modPower una sola volta e quindi ha complessità $\tilde{O}(\log^5 n)$, dato che: $\log base = O(\log^2 n) = \log exponent$

Gli esperimenti condotti mostrano che scegliere $base = a = 2^{\bar{\Omega}(n)^2}$ è sufficiente per permettere alla funzione findCarmichaelFactor di trovare un fattore non triviale di un numero di Carmichael. Perciò si ha che findCarmichaelFactor è euristicamente deterministico con complessità $\tilde{O}(\log^5 n)$.

7.5.2 Seconda versione

Dato che $\bar{\Omega}(n)$ è il massimo numero possibile di fattori primi di n, cosa succederebbe se come valore di α si scegliesse $\omega(n)$ e cioè la quantità media dei fattori primi distinti che compongono n?
Il teorema di Hardy–Ramanujan, provato da Hardy e Ramanujan nel 1917, afferma che $\omega(n) = \log\log n$. [5]
In parole semplici, questo significa che la maggior parte dei numeri hanno circa $\log\log n$ fattori primi distinti.
Una più precisa versione del teorema afferma che

$$\left|\omega(n) - \log\log n\right| < \log\log^{\frac{1}{2}+\epsilon} n \tag{7.13}$$

per quasi tutti gli interi (ad eccezzione di una loro parte infinitesimale). Questo significa che, se $g(x)$ è uguale al numero di interi positivi n minori di x per i quali neq.(7.13) non è soddisfatta, allora

$$\lim_{x\to\infty} \frac{g(x)}{x} = 0 \tag{7.14}$$

In questa implementazione dell'algoritmo di fattorizzazione, scegliendo $\alpha = \omega(n)$ è stato empiricamente osservato che è richiesta un'analisi più approfondita del metodo di ricerca di una base a idonea per la fattorizzazione.

7.5.2.1 La scelta delle basi

Nel tentativo di scoprire quale sia l'insieme di basi più adatto ai fini del test è stata adottata la seguente tecnica:

1. Si sceglie una base di partenza β
2. Si pone $\delta = 0$
3. Si cerca una base $a = \beta \pm \delta$ idonea per la fattorizzazione tra tutte le basi che siano distanti δ da β
4. Nel caso in cui non si trovi alcuna base idonea, si pone $\delta = \delta + 1$ e si riparte dal punto 3

Quanto osservato per tutti i numeri di Carmichael fino a $10^{21} \approx 2^{70}$ è che:

- Scegliendo come parametri
  - $\alpha = \lceil \log \log n \rceil$
  - $\beta = 2^{\alpha\bar{\Omega}(n)}$

  il massimo valore di δ dipende dalla funzione rappresentata con il tratto continuo di colore *verde*, figura 7.1

- Scegliendo come parametri
  - $\alpha = \lceil 2 \log \log n \rceil$
  - $\beta = 2^{\alpha\bar{\Omega}(n)}$

  il massimo valore di δ dipende dalla funzione rappresentata con il tratto continuo di colore *nero*, figura 7.1

- Scegliendo come parametri
  - $\alpha = \lceil 2 \log \log n \rceil$
  - $\beta = 2^{\alpha^2}$

  il massimo valore di δ dipende dalla funzione rappresentata con il tratto continuo di colore *blu*, figura 7.1

- Scegliendo come parametri
  - $\alpha = \lceil \log \log^2 n \rceil$
  - $\beta = 2^{\alpha\bar{\Omega}(n)}$

  il massimo valore di δ dipende dalla funzione rappresentata con il tratto continuo di colore *arancione*, figura 7.1 (vedi la tabella F.2 in appendice F per maggiori dettagli sul valore numerico dei punti evidenziati)

- Scegliendo come parametri

- $\alpha = \lceil \log \log^2 n \rceil$
- $\beta = 2^{\alpha^2}$

il massimo valore di δ dipende dalla funzione rappresentata con il tratto continuo di colore *rosso*, figura 7.1 (vedi la tabella F.1 in appendice F per maggiori dettagli sul valore numerico dei punti evidenziati)

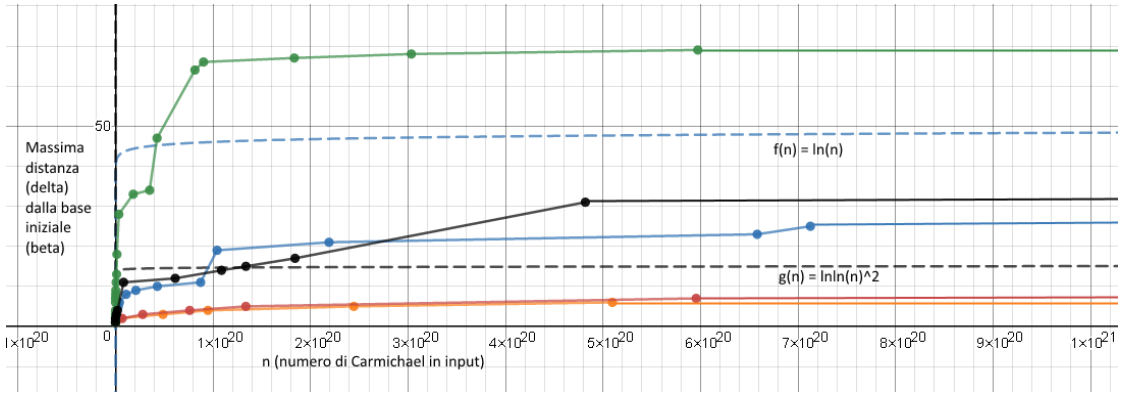

Figura 7.1: Scelta delle basi nella fattorizzazione dei numeri di Carmichael: comportamento asintotico della crescita del massimo δ al variare dei parametri per tutti i numeri di Carmichael fino a $10^{21} \approx 2^{70}$

Quanto osservato è che

- All'aumentare del valore di α diminuisce il valore massimo raggiunto da δ
- Quando $\alpha = \lceil 2 \log \log n \rceil$, scegliere 2^{α^2} come β iniziale mediamente produce risultati migliori rispetto a scegliere $2^{\alpha\bar{\Omega}(n)}$. Per quanto riguarda $\alpha = \lceil \log \log^2 n \rceil$, i valori massimali di δ sono pressochè identici per le due diverse scelte.

Il valore di $\omega(n)$ è uguale a $\log \log n$ per un generico n, in questo caso però n non è un intero qualunque ma è sempre un numero di Carmichael. Quanto congetturato è che se n è un numero di Carmichael allora $\omega(n) = O((\log \log n)^2)$, ecco perchè scegliere $\alpha \approx \log \log^2 n$ dà un risultato migliore del scegliere $\alpha \approx \log \log n$

7.5.2.2 Il codice C++

Di seguito viene mostrata una possibile implementazione C++ dell'algoritmo descritto pocanzi e con parametri:

- $\alpha = \lceil \log \log^2 n \rceil$
- $\beta = 2^{\alpha^2}$

Come nel caso precedente, si tratta di un template pensato per l'uso con la libreria NTL [13].

Algoritmo 7.7 : *Header* del 2do algoritmo di fattorizzazione per numeri di Carmichael

```
#include <math.h>
#include <iostream>
#include <fstream>
using namespace std;

// #define LOG2 log(2)
#define LOG2 0.693147180559945

#ifndef __Sovrano
#define __Sovrano

template <class INT, class FLOAT> // INT is an integer type. FLOAT is a
    floating point type.
// NTL::ZZ is suggested as INT type, double is suggested as FLOAT type
class Sovrano
{
        public:
                Sovrano( ); // O( 1 ) constructor
                bool findCarmichaelFactor( const INT& n ); // euristically
                    O~( log^3 n )
                INT getFactor( ) const { return factor; } // O( 1 )
```

```
        protected:
                void initialize( const INT& n ); // O~( log n )
                // bool findPath( const INT& base, const INT&
                    distance_bound ); // O~( distance_bound * log
                    real_number * log pow_number * log exponent )
                bool findPath( const INT& base ); // O~( MAX(distance) *
                    log real_number * log pow_number * log exponent )
                bool FermatTest( const INT& base ); // O~( log exponent *
                    log pow_number * log real_number )
                void modPower( INT base, INT exponent, const INT& modulus )
                    ; // O~( log exponent * log modulus * log real_number )
                bool findFactor( INT y ); // O~( log y * log real_number )
                bool computeGCD( const INT& m ); // O~( log MIN(real_number
                    ,m) )
                bool hasFoundValidFactor( ) const { return has_found_factor
                    ; } //O( 1 )
        private:
                static const INT int0, int1, int2, int3, int4; // Improves
                    template performance when using non standard types as
                    INT. Prevents constructing new INT objects each time.
                bool has_found_factor; // initialize variables
                FLOAT mean_omega; // initialize variables
                INT factor, real_number, pow_number, exponent, base,
                    around_size; // initialize variables
                INT distance; // findExplorerPath variables
                INT remainder; // modPower local variables
                INT gcd; // computeGCD local variables
                INT pow_of_p, tail; // findFactor local variables
};

#endif
```

Algoritmo 7.8 : *Body* **del 2do algoritmo di fattorizzazione per numeri di Carmichael**

```
#include "Sovrano.h"

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int0(0); // Equal to 0. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int1(1); // Equal to 1. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int2(2); // Equal to 2. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int3(3); // Equal to 3. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int4(4); // Equal to 4. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
Sovrano<INT, FLOAT>::Sovrano( ):factor(int1),has_found_factor(false){};
```

```
template <class INT, class FLOAT>
void Sovrano<INT, FLOAT>::initialize( const INT& n ) // O~( log n )
{
	real_number = n;
	has_found_factor=false;
	factor=int1; // factor=1
	around_size = mean_omega = ceil( pow(log(log(real_number)),2) );
	pow_number = power( n, mean_omega ); // O~( log n )
	base = power( int2, mean_omega*mean_omega ); // O~( log n )
	exponent = pow_number; // For the Fermat Test
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::findCarmichaelFactor( const INT& n ) //
    euristically O~( log^3 n )
{
	initialize( n ); // O~( log n )

	/* Trial division */
	if ( computeGCD( int3 ) ) // O( 1 )
	{
		return true; // n is multiple of 3
	}

	// findPath( base, around_size ); // O~( log^3 n )
	findPath( base ); // O~( log^3 n ) -> euristically, MAX(distance)
	    is O( loglog^2 n )
	return hasFoundValidFactor();
}

template <class INT, class FLOAT>
```

```
// bool Sovrano<INT, FLOAT>::findPath( const INT& base, const INT&
    distance_bound ) // O~( distance_bound * log real_number * log
    pow_number * log exponent )
bool Sovrano<INT, FLOAT>::findPath( const INT& base ) // O~( MAX(
    distance) * log real_number * log pow_number * log exponent )
{
       distance = int0; // distance = 0
       if ( !FermatTest( base ) ) // O~( log exponent * log pow_number *
            log real_number )
       {
              return true;
       }

       // for ( distance = int1; distance<distance_bound; ++distance )
          // this cycle is repeated o( distance_bound ) times
       for ( distance = int1; ; ++distance ) // this cycle is
          euristically repeated O( loglog^2 n ) times
       {
              if ( !FermatTest(base-distance) || !FermatTest(base+
                  distance) ) // O~( log exponent * log pow_number * log
                  real_number )
              {
                     return true;
              }
       }
       return false;
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::FermatTest( const INT& base ) // O~( log
    exponent * log pow_number * log real_number )
{
       modPower( base, exponent, pow_number );
```

```
        return !hasFoundValidFactor( ); // O( 1 )
}

template <class INT, class FLOAT>
void Sovrano<INT, FLOAT>::modPower( INT base, INT exponent, const INT&
    modulus ) // O~( log exponent * log modulus * log real_number )
{
        if ( exponent <= int1 ) // exponent <= 1
        {
                return;
        }

        remainder = int1;
        while ( exponent > int1 ) // while factor > 1 -> O( log2 exponent
            )
        {
                if ( IsOdd( exponent ) )
                {
                        remainder = MulMod( base, remainder, modulus ); // O
                            ~( log base )
                        if ( findFactor(remainder) ) // O~( log remainder *
                            log real_number )
                        {
                                return;
                        }
                }

                base = MulMod(base,base,modulus); // O~( log base )
                if ( findFactor(base) ) // O~( log base * log real_number )
                {
                        return;
                }
```

```
                exponent /= int2; // exponent = floor(exponent/2) ->
                    exponent is always an integer
        }

        base = MulMod(base,remainder,modulus); // O~( log base )
        if ( findFactor(base) ) // O~( log base * log real_number )
        {
                return;
        }
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::findFactor( INT y ) // O~( log y * log
    real_number )
{
        pow_of_p=int1, tail=int0;
        while ( y > int1 ) // y > 1
        {
                if ( IsOdd(y) ) // y is odd
                {
                        tail = AddMod(pow_of_p,tail,real_number);
                }
                pow_of_p = MulMod(pow_of_p,int2,real_number);
                y /= int2; // y = floor(y/2) -> y is always an integer

                if ( computeGCD( AddMod(pow_of_p,tail,real_number) ) ) // O
                    ~( log MIN(real_number,(pow_of_p+tail)%real_number) )
                    -> Euclidean GCD algorithm
                {
                        return true;
                }
        }
        return false;
```

```
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::computeGCD( const INT& m ) // O~( log MIN(
    real_number,m) )
{
        gcd = GCD( real_number, m ); // O( log MIN(real_number,m) )
        if ( gcd>int1 && gcd<real_number ) // gcd>1 && gcd<real_number
        {
                factor=gcd;
                has_found_factor=true;
                return true;
        }
        return false;
}
```

7.5.2.3 La complessità asintotica

La complessità in tempo dell'algoritmo di fattorizzazione appena descritto è euristicamente $\tilde{O}(\log^3 n)$, in quanto:

- **initialize**(n) ha complessità $\tilde{O}(\log n)$
- **hasFoundValidFactor**() ha complessità $O(1)$
- **computeGCD**(m) esegue l'algoritmo di Euclide una volta e ha quindi complessità $\tilde{O}(\log min(n,m)) = \tilde{O}(\log n)$, in quanto $min(n,m) \leq n$.
- **findFactor**(y) ha complessità $\tilde{O}(\log y \cdot \log n)$
- **modPower**(base, exponent, modulus) ha complessità $\tilde{O}(\log exponent \cdot \log modulus \cdot \log n)$
- **FermatTest**(base) esegue modPower una sola volta e ha quindi complessità $\tilde{O}(\log exponent \cdot \log modulus \cdot \log n)$

- **findPath**(base) euristicamente esegue FermatTest al più $\tilde{O}((\log\log n)^2)$ volte e quindi ha complessità $\tilde{O}(\log^3 n)$, questo perchè n è un numero di Carmichael
- **findCarmichaelFactor**(n) esegue findPath una sola volta e quindi euristicamente ha complessità $\tilde{O}(\log^3 n)$, dato che: $\log base = \tilde{O}(\log n) = \log exponent$

Si è osservato che esiste sempre una base a idonea e quindi un $\delta \geq 0$ per cui $a = \beta \pm \delta$. Quanto mostrato in immagine (7.1) permette di dire che la dimesione massima di δ cresce euristicamente come $\tilde{O}((\log\log n)^2)$ e quindi che la complessità in tempo dell'algoritmo di fattorizzazione appena proposto è euristicamente $\tilde{O}(\log^3 n)$.
Nel prossimo capitolo viene mostrato come derivare dall'algoritmo appena descritto un nuovo efficiente test di primalità.

Capitolo 8

Un nuovo test di primalità: *il test degli Esploratori*

Nei capitoli precedenti è stato mostrato quanto ed in che modo il problema della verifica della primalità influisca sulle performance e sulla sicurezza dei crittosistemi asimmetrici oggetto di questa tesi.
L'esperimento che segue mostra come sia possibile ottenere dall'algoritmo in sezione 7.5.2 *un nuovo test di primalità deterministico ed euristico con complessità polinomiale* $\tilde{O}(\log^2 n)$ e che possa competere con il test di Miller-Rabin. Questo nuovo algoritmo verrà chiamato in seguito test degli Esploratori.

8.1 Il test di Miller e la scelta delle basi

L'algoritmo di fattorizzazione descritto nel capitolo precedente si basa su una variante deterministica del test di Fermat. Dato che il test di Miller è, in un certo senso, una diretta estensione del test di Fermat, l'idea alla base del test degli Esploratori è proprio quella di usare il test di Miller insieme all'algoritmo di scelta delle basi mostrato in sezione 7.5.2.1.
Le differenze e le similitudini principali tra l'algoritmo di fattorizzazione descritto in sezione 7.5.2 ed il test degli Esploratori sono le seguenti:

- Algoritmo di fattorizzazione

  - Prende in input un numero di Carmichael n
  - Verifica che n non sia multiplo di 3
  - Fa uso di th.(6) calcolando

$$a^{n^\alpha} \mod n^\alpha \tag{8.1}$$

    ma senza verificare che $a^{n^\alpha} \mod n^\alpha$ sia congruente con $a \mod n$. Questo perchè quanto *congetturato* è che n^α è in grado di generare uno spazio sufficientemente grande da permettere alla funzione $findFactor$ di trovare un fattore non triviale di n. Dato che n è sempre un numero di Carmichael, si congettura che il valore ottimale di α sia $\omega(n) \approx \log\log^2 n$
  - Sceglie le basi a tramite l'algoritmo di scelta delle basi mostrato in sezione 7.5.2.1
  - Trova un fattore non triviale di n

- Test degli Esploratori
  - Prende in input un qualsiasi intero n
  - Verifica che n non sia multiplo di 2 o di 3
  - Verifica che $n \not\equiv 0 \mod \lfloor \sqrt{n} \rfloor$ e cioè che n non sia un quadrato perfetto od il prodotto di due interi distanti al più 2 tra loro. In caso contrario n è detto essere un *simquadro* ed è quindi composto.
  - Usa nel modo seguente il th.(6) combinato con le proprietà descritte all'inizio della sezione 6.1.2
    1. Se n è primo, allora $(a^{n^\alpha} \mod n^\alpha) \equiv a^n \mod n$
    2. Se n è primo e a è coprimo rispetto a n, allora $a^{n-1} \equiv 1 \mod n$
    3. Si calcolano d e s per cui $n - 1 = 2^s \cdot d$
    4. Se $a^d - 1 \not\equiv 0 \mod n$
       (a) Se $a^{2^r \cdot d} - 1 \equiv 0 \mod n$ per qualche intero r tale che $0 < r \leq s - 1$, allora n è composto

(b) Se per ogni intero r, tale che $0 \leq r \leq s-1$, si ha che $a^{2^r \cdot d} + 1 \not\equiv 0 \mod n$, allora n è composto

In questo caso risulta inutile usare la funzione $findFactor$ ad ogni elevamento modulare al quadrato

– Sceglie le basi a tramite l'algoritmo di scelta delle basi mostrato in sezione 7.5.2.1. Dato che n è un intero qualunque, si congettura che il valore ottimale di α sia $\omega(n) \approx \log\log n$, vedi th.(7.13)

– Verifica la primalità di n

In questo modo, l'unica differenza sostanziale tra il test degli Esploratori ed il test di Miller è la scelta deterministica della base idonea a che rimane come spiegato in sezione 7.5.2.
Le modifiche appena descritte danno vita ad una variante del test di Miller basata sulle proprietà degli pseudoprimi di Carmichael mostrate nel precedente capitolo. Il risultato è un nuovo test di primalità euristico, deterministico ed estremamente efficiente chiamato test degli Esploratori. Il corretto funzionamento di tale test per $\beta = 2^{\alpha^2}$ è stato verificato su *tutti* gli interi positivi fino a circa 2^{44}.
Di seguito viene descritta la famiglia di input che, in base ad evidenze empiriche, dominano la complessità del test degli Esploratori nel caso pessimo.

8.2 I numeri Esploratori

Cosa sono i numeri Esploratori? Sono i peggiori interi composti che sia possibile dare in input al test degli Esploratori. Per capirlo si definiscano le variabili $x, y, \bar{\beta}, \Delta \in \mathbb{N}^+$, tali per cui $x > y$.
Un Esploratore è un numero composto $n = x \cdot y$ tale per cui esiste una coppia $< x, y >$ che renda vere le seguenti proprietà:

$$\bar{\beta} = \frac{x-1}{y-1} - 1 \tag{8.2}$$

in altre parole, dev'essere vero che $y-1$ divide $x-1$. Da eq.(8.2) si deduce che

$$\Delta = \sqrt{\bar{\beta}^2 + 4n \cdot (\bar{\beta}+1)} \tag{8.3}$$

in altre parole, dev'essere vero che $\bar{\beta}^2 + 4n(\bar{\beta} + 1)$ è un quadrato perfetto.
Infatti, l'unica y positiva della parabola che si ottiene unendo la definizione di n con eq.(8.2) è:

$$y = \frac{\Delta + \bar{\beta}}{2(\bar{\beta} + 1)} \tag{8.4}$$

da cui segue che

$$\begin{aligned} x &= (y - 1) \cdot (\bar{\beta} + 1) + 1 \\ &= \frac{\Delta - \bar{\beta}}{2} \end{aligned} \tag{8.5}$$

Giunti a questo punto possiamo riscrivere n in funzione di $\bar{\beta}$:

$$\begin{aligned} n &= x \cdot y \\ &= \frac{\Delta^2 - \bar{\beta}^2}{4(\bar{\beta} + 1)} \end{aligned} \tag{8.6}$$

questa rappresentazione di n è riducibile alla notazione usata da Fermat nel suo famoso metodo di fattorizzazione (vedi appendice D) che ispirò altri metodi illustri tra cui quello di J.D. Dixon [7] ed il crivello quadratico di C. Pomerance [12].
Più in generale, se un numero intero n non è Esploratore, allora $\bar{\beta}$ e Δ perdono la proprietà di essere interi per qualche coppia $< x, y >$ rimanendo però razionali: $\bar{\beta}, \Delta \in \mathbb{Q}$.

8.2.1 Alcune proprietà

1. Esistono numeri Esploratori con 2 o più fattori. Ad esempio:
   - Un Esploratore con 2 fattori è $37 \cdot 73 = 2701$
   - Un Esploratore con 3 fattori è 1091917·2183833·3275749 = 7811234376213792889
   - Un Esploratore con 4 fattori è 3457·13441·39937·219649 = 407601364610119681
2. Il numero di fattori primi di un Esploratore è pari al grado dell'Esploratore stesso. In tal senso, con la notazione $\mathcal{E}_m$ si intenderà l'insieme degli Esploratori di grado m, dove $m \geq 2$ e $m \in \mathbb{N}$. Ad esempio, $\mathcal{E}_2$ è l'insieme degli Esploratori di grado 2 e quindi $3 \cdot 7 \in \mathcal{E}_2$, mentre $3 \cdot 7 \cdot 7 \notin \mathcal{E}_2$.

3. I numeri di Carmichael sono un sottoinsieme dei numeri Esploratori. Vedi th.(5).

4. Si è empiricamente osservato (e si congettura) che i numeri Esploratori in $\mathcal{E}_2$ con $\bar{\beta} = 1$ o $\bar{\beta} = 2$ regolano *sempre* la complessità asintotica del test degli Esploratori nel suo caso pessimo (vedi la tabella F.3 in appendice F). Tali numeri verranno in seguito chiamati numeri Originali.

8.3 La fattorizzazione dei numeri Originali ed il test degli Esploratori

Le proprietà caratterizzanti i numeri Originali sono:

- Sono numeri Esploratori in $\mathcal{E}_2$
- Hanno $\bar{\beta} = 1$ o $\bar{\beta} = 2$

Una caratteristica dei numeri Esploratori è che, usando eq.(8.3) insieme a eq.(8.5), possono essere facilmente fattorizzati conoscendo $\bar{\beta}$.
Nel caso dei numeri Originali si è a conoscenza del loro $\bar{\beta}$ e questo permette di ridurre la complessità asintotica del test degli Esploratori fattorizzando i numeri Originali.
In aggiunta, oltre ai numeri Originali si è deciso di fattorizzare anche tutti gli altri numeri Esploratori con $\bar{\beta} \leq \lceil (\log \log n)^{\hat{\gamma}} \rceil$, dove $\hat{\gamma}$ è una costante positiva. Euristicamente si ritiene che il $\hat{\gamma}$ ottimale sia circa 2.
Dato che il test degli Esploratori si basa sull'algoritmo di scelta delle basi descritto in sezione 7.5.2.1, di seguito si mostra il comportamento asintotico della crescita del massimo δ raggiunto dal test degli Esploratori nei seguenti casi:

1. Linea continua *viola*, figura 8.1 (vedi la tabella F.3 in appendice F per maggiori dettagli sul valore numerico dei punti evidenziati):
    - Su tutti gli interi composti e positivi fino a $12994178157973 \approx 2^{44} \approx 1.3 \cdot 10^{13}$
    - *Senza fattorizzare* i numeri Originali ed Esploratori

2. Linea continua *nera*, figura 8.1 (vedi la tabella F.4 in appendice F per maggiori dettagli sul valore numerico dei punti evidenziati):
   - Su tutti gli interi composti e positivi fino a $12994178157973 \approx 2^{44} \approx 1.3 \cdot 10^{13}$
   - *Fattorizzando* i numeri Originali e gli altri numeri Esploratori con $\bar{\beta} \leq \lceil (\log\log n)^2 \rceil$
3. Linea continua *blu*, figura 8.2 (vedi la tabella F.5 in appendice F per maggiori dettagli sul valore numerico dei punti evidenziati):
   - Su tutti i numeri Esploratori in $\mathcal{E}_2$, con $\bar{\beta} < 5$, fino a circa 2^{68}
4. Linea continua *verde*, figura 8.2 (vedi la tabella F.6 in appendice F per maggiori dettagli sul valore numerico dei punti evidenziati):
   - Su tutti i numeri di Carmichael fino a $10^{21} \approx 2^{70}$

In tutti i casi appena enumerati, i parametri adottati per l'algoritmo di scelta delle basi sono:

- $\alpha = \lceil \log\log n \rceil$, l'esponente di th.(6) usato per ottenere β
- $\beta = 2^{\alpha^2}$, la base di partenza per l'algoritmo di scelta delle basi

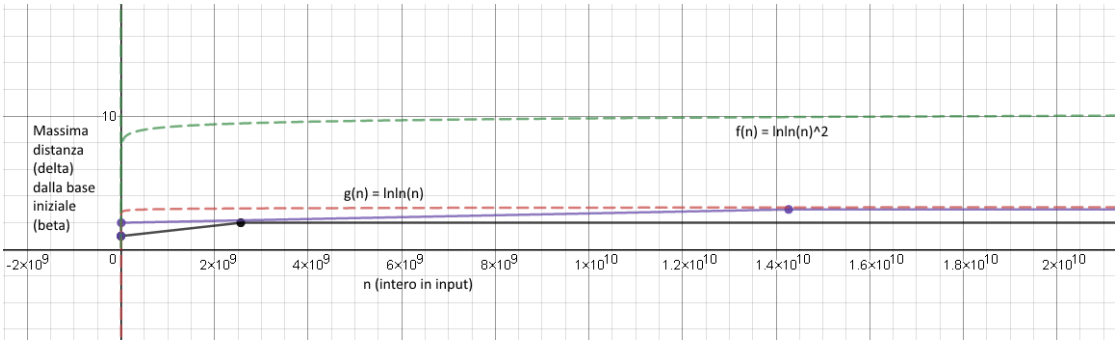

Figura 8.1: Test degli Esploratori: comportamento asintotico della crescita del massimo δ per tutti gli interi minori di circa $2^{44} \approx 1.3 \cdot 10^{13}$

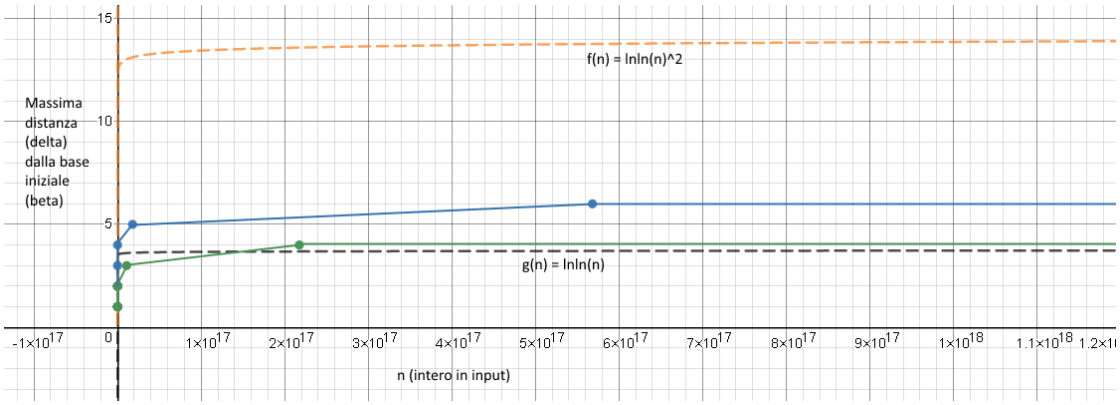

Figura 8.2: Test degli Esploratori: comportamento asintotico della crescita del massimo δ per alcuni insiemi significativi di numeri Esploratori minori di $10^{21} \approx 2^{70}$

8.4 Il codice C++

Di seguito viene mostrata una possibile implementazione C++ dell'algoritmo descritto pocanzi e con parametri:

- $\alpha = \lceil \log \log n \rceil$, l'esponente di th.(6) usato per ottenere β
- $\beta = 2^{\alpha^2}$, la base di partenza per l'algoritmo di scelta delle basi
- $\hat{\gamma} = 2$

Come negli altri casi, si tratta di un template pensato per l'uso con la libreria NTL [13].

Algoritmo 8.1 : *Header* del test di primalità degli Esploratori

```
#include <math.h>
using namespace std;

#ifndef __Sovrano
#define __Sovrano
```

```
template <class INT, class FLOAT> // INT is an integer type. FLOAT is a
    float type.
// NTL::ZZ is suggested as INT type, double is suggested as FLOAT type
class Sovrano
{
       public:
              Sovrano( ); // O( 1 ) -> constructor
              bool isPrime( const INT& n ); // O~( log^2 n )
              bool isSimSquare( ); // O~( log n ) -> check whether n is a
                   perfect square or a multiple of twin primes
              INT getFactor( ) const { return factor; } // O( 1 )
       protected:
              void initialize( const INT& n ); // O~( log n )
              bool findDelta( const INT& beta_bound ); // O~( beta_bound
                  * log n )
              bool StrongPrimalityTest( const INT& base ); // O~( log^2
                  real_number )
              bool findBase( const INT& base, const INT& base_bound ); //
                   O~( base_bound * log^2 n )
              bool hasFoundValidFactor( ) const { return has_found_factor
                  ; } //O( 1 )
              bool findValidFactor( const INT& m ); // O~( log MIN(m,n) )
              bool isValidBase( const INT& base ) const; // O( 1 )
       private:
              static const INT int0, int1, int2, int3, int4; // Improves
                  template performance when using non standard types as
                  INT. Prevents constructing new INT objects each time.
              bool has_found_factor; // initialize variables
              INT base_mod, d, r, step; // StrongPrimalityTest local
                  variables
              INT gcd; // findValidFactor local variables
              INT beta, delta, x; // findDelta local variables
```

```
            INT factor, real_number, real_number_minus_one, sqrt,
                pow_number, exponent, base, around_size; // initialize
                variables
            INT miller_base; // isPrime variables
            FLOAT mean_omega, real_loglog; // initialize variables
};

#endif
```

Algoritmo 8.2 : *Body* **del test di primalità degli Esploratori**

```
#include "Sovrano.h"

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int0(0); // Equal to 0. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int1(1); // Equal to 1. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int2(2); // Equal to 2. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
```

```
const INT Sovrano<INT, FLOAT>::int3(3); // Equal to 3. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
const INT Sovrano<INT, FLOAT>::int4(4); // Equal to 4. Improves template
    performance when using non standard types as INT. Prevents
   constructing new INT objects each time.

template <class INT, class FLOAT>
Sovrano<INT, FLOAT>::Sovrano( ):factor(int1),has_found_factor(false){};
   // O( 1 ) -> constructor

template <class INT, class FLOAT>
void Sovrano<INT, FLOAT>::initialize( const INT& n ) // O~( log n )
{
       real_number = n;
       real_number_minus_one = n - int1;
       has_found_factor=false;
       factor=int1; // factor=1
       real_loglog = log(log(n));
       mean_omega = ceil( real_loglog );
       around_size = ceil( real_loglog*real_loglog ); //to improve
       sqrt = SqrRoot( n ); // O~( log n ) -> Using Newton's method
       base = PowerMod( int2, mean_omega*mean_omega, n ); // O~( log n )
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::isPrime( const INT& n ) // O~( log^2 n )
{
       initialize( n ); // O~( log n )

       /* Trial division */
```

```
        if ( findValidFactor( int2 ) || findValidFactor( int3 ) ||
            isSimSquare( ) ) // O( 1 )
        {
                return false; // n is multiple of 2 or 3 or n is a sim-
                    square
        }

        /* Delta finder -> Removes most complex Explorer_2*/
        if ( findDelta( around_size ) ) // O~( mean_omega * log n ) = O~(
             log n )
        {
                return false;
        }

        /* Apply the little Fermat theorem on base b to speed up
            primality testing */
        if ( findBase( base, around_size ) ) // O~( around_size * log^2 n
             ) = O~( log^2 n )
        {
                return false;
        }

        return true;
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::isSimSquare( ) // O~( log real_number ) ->
    check whether real_number is a perfect square or a multiple of twin
    primes
{
        if ( real_number > int3 && real_number%sqrt == int0 ) // O~( log
            real_number )
        {
```

```
                factor=sqrt;
                has_found_factor=true;
                return true;
        }
        return false;
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::findValidFactor( const INT& m ) // O~( log MIN
    (m,real_number) )
{
        gcd = GCD( real_number, m ); // O~( log MIN(m,real_number) ), for
            more details see https://en.wikipedia.org/wiki/
           Computational_complexity_of_mathematical_operations
        if ( gcd>int1 && gcd<real_number ) // gcd>1 && gcd<real_number
        {
                factor=gcd;
                has_found_factor=true;
                return true;
        }
        return false;
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::findDelta( const INT& beta_bound ) // O~(
    beta_bound * log real_number )
{
        for ( beta = int1; beta<=beta_bound; ++beta ) // repeated o(
           beta_bound ) times
        {
                delta = SqrRoot( power(beta,2) + (beta+int1)*real_number*
                    int4 ); // O~( log real_number ) -> Using Newton's
                    method
```

```
                x = (delta - beta)/int2;
                if ( real_number%x==int0 ) // real_number%x == 0
                {
                        factor=x;
                        has_found_factor=true;
                        return true;
                }
        }
        return false;
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::findBase( const INT& base, const INT&
    base_bound ) // O~( base_bound * log^2 real_number )
{
        miller_base = int0;
        if ( !StrongPrimalityTest( base ) ) // O~( log^2 real_number )
        {
                return true;
        }
        for ( miller_base = int1; miller_base <= base_bound; ++
            miller_base ) // repeated o( base_bound ) times
        {
                if ( !StrongPrimalityTest( base+miller_base ) || !
                    StrongPrimalityTest( base-miller_base ) ) // O~( log^2
                    real_number )
                {
                        return true;
                }
        }
        return false;
}
```

```
template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::StrongPrimalityTest( const INT& base ) // O~(
    log^2 real_number )
{
       if ( findValidFactor(base) ) return false; // O~( log real_number
            ) -> base is not coprime to real_number -> real_number is not
            prime
       base_mod = base%real_number;
       if ( !isValidBase( base_mod ) ) return true; // O( 1 )

       d = real_number_minus_one/int2;
       r = int1;
       while ( !IsOdd(d) ) // repeated o( log2 real_number ) times ->
           until d is even
       {
              ++r;
              d /= int2;
       }

       step = PowerMod( base_mod, d, real_number ); // O~( log^2
           real_number )
       if ( step == int1 || step == real_number_minus_one ) return true;

       while ( r > int1 ) // repeated o( log2 real_number ) times
       {
              --r;
              //previous_step = step;
              step = PowerMod( step, int2, real_number ); // O~( log
                  real_number )
              if ( step == int1 ) return false; // Because: or 1 < GCD(n,
                  previous_step-1) < n, or 1 < GCD(n,previous_step+1) < n
              if ( step == real_number_minus_one ) return true;
       }
```

```
        return false;
}

template <class INT, class FLOAT>
bool Sovrano<INT, FLOAT>::isValidBase( const INT& base ) const // O( 1 )
{
        return base > int1;
}
```

8.4.1 La complessità asintotica

La complessità in tempo del test degli Esploratori è $\tilde{O}(\log^2 n)$, in quanto:

- **initialize**(n) ha complessità $\tilde{O}(\log n)$
- **isValidBase**(base) ha complessità $O(1)$
- **hasFoundValidFactor**( ) ha complessità $O(1)$
- **getFactor**( ) ha complessità $O(1)$
- **findValidFactor**(m) esegue una volta l'algoritmo di Euclide e quindi ha complessità $\tilde{O}(\log min(n,m)) = \tilde{O}(\log n)$, in quanto $m < n$.
- **isSimSquare**( ) verifica se n è un simquadro. isSimSquare ha complessità $\tilde{O}(\log n)$
- **StrongPrimalityTest**(base) è il test di primalità alla base del test di Miller e ha complessità $\tilde{O}(\log^2 n)$
- **findBase**(base, base_bound) esegue $base\_bound$ volte StrongPrimalityTest e ha quindi complessità $\tilde{O}(base\_bound \cdot \log^2 n)$
- **findDelta**(beta_bound) fattorizza i numeri Esploratori con $\bar{\beta} \leq beta\_bound$. Dunque esegue $beta\_bound$ volte un'operazione di radice su un numero di dimensione $O(\log n)$ e ha quindi complessità $\tilde{O}(beta\_bound \cdot \log n)$

- **isPrime**(n) esegue una volta l'algoritmo findDelta ed una volta l'algoritmo findBase. Dato che euristicamente si è visto che $base\_bound = \tilde{O}((\log\log n)^2)$ e dato che $beta\_bound = O(\log\log n)$, isPrime ha complessità $\tilde{O}(\log^2 n)$

Se il test degli Esploratori prendesse in input solo interi composti, allora si potrebbe dire che il test è deterministico con complessità euristica polinomiale in quanto non esistono pseudoprimi forti per tutte le basi. Il test degli Esploratori però può ricevere in input anche numeri primi e dato che un numero primo è tale per tutte le basi, si ha che la verifica della primalità (affinchè sia polinomiale) dev'essere esplicitamente bloccata dopo aver testato l'input su una quantità opportuna di basi diverse. Dato che si congettura che il numero di basi richieste sia al più $\tilde{O}((\log\log n)^2)$ usando l'algoritmo di scelta delle basi descritto in sezione 7.5.2.1, questo fa sì che il test degli Esploratori sia euristicamente deterministico con complessità polinomiale.

8.5 Conclusioni

Nel caso del test di Miller-Rabin la probabilità P_n che un numero composto n venga identificato come probabilmente primo è significativamente inferiore a $\frac{1}{4^k}$, dove k è il numero di volte per cui viene ripetuto il test su un input casuale. [34]
Nel caso in cui $k \to \infty$ con $n \to \infty$, si avrebbe che $P_n \to 0$, in quanto

$$\begin{aligned} \lim_{k\to\infty} P_n &= \lim_{k\to\infty} \frac{1}{4^k} \\ &= 0 \end{aligned} \tag{8.7}$$

Il test degli Esploratori è un caso particolare e deterministico del test di Miller-Rabin con $k = O((\log\log n)^\gamma)$, dove $\gamma > 0$ è una costante euristicamente ritenuta prossima a 2. Si ha quindi che

$$\begin{aligned} \lim_{n\to\infty} k &= \lim_{n\to\infty} (\log\log n)^\gamma \\ &= \infty \end{aligned} \tag{8.8}$$

Per cui, dato che almeno per tutti gli interi n fino a circa $2^{44} \approx 1.3 \cdot 10^{13}$ il test degli Esploratori non sbaglia mai, si può dire che ad $n \to \infty$ la probabilità che il test degli

Esploratori sbagli è esattamente 0.

Nel caso in cui si dimostrino le congetture presentate nei precedenti capitoli, si può dire che il test degli Esploratori sia *il test di primalità deterministico più veloce allo stato dell'arte*. Con le opportune modifiche descritte in appendice E, il test degli Esploratori risulta essere anche l'algoritmo di fattorizzazione deterministico per numeri di Carmichael più veloce allo stato dell'arte.

Capitolo 9

Ricerce future e ringraziamenti

Le mie attuali ricerche crittoanalitiche stanno convergendo verso un nuovo algoritmo deterministico per la fattorizzazione di un intero qualsiasi. I risultati iniziali sembrano essere promettenti.
Voglio ringraziare:

- Mr. **Richard G.E. Pinch** [14], per avermi gentilmente fornito la lista completa dei numeri di Carmichael fino a 10^{21}
- L'**Università di Bologna**, per avermi fornito l'accesso da remoto a 58 server di laboratorio usati per i test sugli algoritmi euristici presentati negli ultimi capitoli

Appendices

Appendice A

Legenda

Per comodità, alcune parole usate in questa tesi sono state abbreviate. Di seguito potrete trovare la lista delle abbreviazioni con i loro significati equivalenti.

- eq. ≡ equazione
- neq. ≡ disequazione
- th. ≡ teorema
- hyp. ≡ ipotesi
- pg. ≡ pagina
- fig. ≡ figura
- es. ≡ esempio

Appendice B

Notazioni d'ordine

Per descrivere il comportamento asintotico di una funzione $f : \mathbb{N} \to \mathbb{N}$ accettata da un calcolatore tradizionale, si è deciso di adottare le notazioni d'ordine descritte nella prossima tabella.
Si ricorda che, in un calcolatore tradizionale, ogni input è un insieme di bit ed è quindi univocamente rappresentabile mediante un numero intero in $\mathbb{N}$.

Classe	Definizione	Descrizione
$O(f)$	$\{g : \mathbb{N} \to \mathbb{N} \mid \exists c \forall n, g(n) \leq c \cdot f(n) + c\}$	La classe delle funzioni che crescono al più come f
$\tilde{O}(f)$	$\{g : \mathbb{N} \to \mathbb{N} \mid \exists c \forall n, g(n) \in O(f(n) \cdot \log^c f(n))\}$	La classe delle funzioni che crescono al più come f, ignorando eventuali fattori logaritmici
$o(f)$	$\{g : \mathbb{N} \to \mathbb{N} \mid \forall c \exists m \forall n \geq m, c \cdot g(n) + c \leq f(n)\}$	La classe delle funzioni che crescono meno rapidamente di f
$\Omega(f)$	$\{g : \mathbb{N} \to \mathbb{N} \mid f \in O(g)\}$	La classe delle funzioni che crescono almeno quanto f
$\Theta(f)$	$O(f) \cap \Omega(f)$	La classe delle funzioni che crescono come f

Tabella B.1: Notazioni d'ordine

Appendice C

Il modello di calcolo del costo computazionale

Per misurare la complessità asintotica degli algoritmi proposti in questa tesi, si è deciso di non adottare il modello di costo uniforme. Il modello di costo uniforme considera costante la complessità asintotica delle operazioni aritmetiche di base (es.: moltiplicazione, divisione) e questo può andare bene per analisi della complessità i cui input numerici siano al più grandi quanto il massimo numero rappresentabile con una parola di memoria.
Nel caso degli algoritmi proposti, la dimensione degli input numerici è ritenuta tale da non poter ignorare il costo computazionale di una moltiplicazione. Di seguito viene mostrato il modello di costo adottato per le funzioni aritmetiche principali; con la notazione $M(n)$ si intende la complessità dell'algoritmo di moltiplicazione.

Operazione	**Input**	**Output**	**Algoritmo**	**Complessità**
Addizione	Due numeri di n cifre	Un numero di $n + 1$ cifre	Addizione con riporto	$\Theta(n)$
Sottrazione	Due numeri di n cifre	Un numero di $n + 1$ cifre	Addizione con resto	$\Theta(n)$
Moltiplicazione	Due numeri di n cifre	Un numero di $2n$ cifre	Algoritmo di Fürer	$O(n \log n \cdot 2^{O(\log^* n)})$
Divisione	Due numeri di n cifre	Un numero di n cifre	Divisione di Newton–Raphson	$O(M(n))$
Modulo	Due numeri di n cifre	Un numero di n cifre	Divisione di Newton–Raphson	$O(M(n))$
Radice quadrata	Un numero di n cifre	Un numero di n cifre	Metodo di Newton	$O(M(n))$
Esponenziazione modulare	Due numeri di n cifre ed un esponente di k bit	Un numero di n cifre	Esponenziazione modulare per quadrati	$O(M(n) \cdot k)$

Tabella C.1: Modello computazionale

Appendice D

Il metodo di fattorizzazione di Fermat

In teoria dei numeri, il metodo di fattorizzazione di Fermat si basa sulla rappresentazione di un intero composto e dispari n come la differenza di due quadrati: $n = a^2 - b^2$. Tale metodo viene presentato in quanto è ritenuto essere alla base del corretto funzionamento degli algoritmi descritti negli ultimi capitoli.
Equivalentemente, il metodo di Fermat è basato sul trovare una congruenza tra quadrati modulo n, dove n è il numero che si vuole fattorizzare. Tale congruenza viene trovata selezionando casualmente un intero x per cui, per qualche y sia vero che:

$$x^2 \equiv y^2 \mod n \tag{D.1}$$

con

$$x \not\equiv \pm y \mod n \tag{D.2}$$

In questo modo, se x e y fossero coprimi rispetto a n, allora per qualche intero $\alpha > 0$ si avrebbe che

$$\begin{aligned} \alpha n &= x^2 - y^2 \\ &= (x + y)(x - y) \end{aligned} \tag{D.3}$$

Se n è pari e composto da almeno due fattori primi, allora è stato dimostrato che almeno metà delle soluzioni x, y possibili comporta $1 < GCD(n, x - y) < n$.
Sul metodo di Fermat si basano altri algoritmi famosi, tra cui il Quadratic Sieve o Crivello Quadratico.

Appendice E

Tecniche note per fattorizzare i numeri di Carmichael

Grazie al test di Miller-Rabin, fattorizzare i numeri di Carmichael è relativamente semplice:

- Nel caso particolare in cui n sia un numero di Carmichael, allora per il criterio di Korselt si ha che, per qualche base casuale a coprima con n, il test di Miller-Rabin può identificare n come composto solo se $a^{2^r \cdot d} - 1 \equiv 0 \mod n$ per qualche r tale che $0 < r < s$

- Se n viene identificato dal test come composto, allora

$$1 < GCD(a^{2^{r-1} \cdot d} - 1, n) < n \tag{E.1}$$

  oppure

$$1 < GCD(a^{2^{r-1} \cdot d} + 1, n) < n \tag{E.2}$$

  Questa tecnica di fattorizzazione è un caso particolare del metodo di fattorizzazione di Fermat (vedi appendice D).

Per ulteriori dettagli matematici si legga [27, Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer].

Dato che il test di Miller-Rabin è probabilistico, la tecnica di fattorizzazione appena proposta è polinomiale in tempo $\tilde{O}(\log^2 n)$ ma è anch'essa *probabilistica*.

Appendice F

Esempi numerici

F.1 Algoritmo di fattorizzazione degli pseudoprimi di Carmichael

Di seguito vengono mostrati gli esperimenti più significativi per l'algoritmo di fattorizzazione degli pseudoprimi di Carmichael in sezione 7.5.2.
Significati delle colonne:

1. Con la dicitura *Numero di Carmichael* si intende il numero di Carmichael n in input all'algoritmo

2. δ è la distanza dalla base iniziale descritta nella sottosezione 7.5.2.1

Tabella F.1: Punti evidenziati sulla linea continua *rossa* in fig.7.1 fino a $10^{21} \approx 2^{70}$

Numero di Carmichael	δ
151821983553419521	1
6937889459346457561	2
27973595986255933441	3
76126530764135817601	4
133699550134722962881	5
595583342715844675921	7

Tabella F.2: Punti evidenziati sulla linea continua *arancione* in fig.7.1 fino a $10^{21} \approx 2^{70}$

Numero di Carmichael	δ
187231315941536641	1
4191584984046066961	2
48728998776481487857	3
94817254765460306689	4
244467658390409012329	5
509651902534766342401	6

F.2 Test degli Esploratori

Di seguito vengono mostrati i risultati più significativi per il test degli Esploratori. Significati delle colonne:

1. Con la dicitura *Numero* si intende il numero n in input al test

2. δ è la distanza dalla base iniziale descritta nella sottosezione 7.5.2.1

3. $\bar{\beta}$ è la variabile relativa a n definita in eq.(8.2)

Tabella F.3: Punti evidenziati sulla linea continua *viola* in fig.8.1 fino a $12994178157973 \approx 2^{44} \approx 1.3 \cdot 10^{13}$

Numero	δ	$\bar{\beta}$
91	1	1
6330721	2	2
14267548181	3	2

Tabella F.4: Punti evidenziati sulla linea continua *nera* in fig.8.1 fino a $12994178157973 \approx 2^{44} \approx 1.3 \cdot 10^{13}$

Numero	δ
1247	1
2560600351	2

Tabella F.5: Punti evidenziati sulla linea continua *blu* in fig.8.2 fino a circa $4 \cdot 10^{20} \approx 2^{68}$

Numero	δ	$\bar{\beta}$
91	1	1
6330721	2	2
14267548181	3	2
29616682315681	4	1
17965260416171503	5	1
568152899708218831	6	1

Tabella F.6: Punti evidenziati sulla linea continua *verde* in fig.8.2 fino a $10^{21} \approx 2^{70}$

Numero di Carmichael	δ
15841	1
2560600351	2
10888913045652841	3
217454857254780751	4

Bibliografia

[1] *Elliptic curve primality proving*, https://en.wikipedia.org/wiki/Elliptic_curve_primality_proving

[2] *Fermat's little theorem*, https://en.wikipedia.org/wiki/Fermat's_little_theorem.

[3] *Euclidean algorithm*, https://en.wikipedia.org/wiki/Euclidean_algorithm

[4] *Carmichael number*, https://en.wikipedia.org/wiki/Carmichael_number

[5] *Hardy–Ramanujan theorem*, https://en.wikipedia.org/wiki/Hardy%E2%80%93Ramanujan_theorem

[6] *Fermat factorization*, https://en.wikipedia.org/wiki/Fermat's_factorization_method

[7] *Dixon's factorization method*, https://en.wikipedia.org/wiki/Dixon's_factorization_method

[8] *Elliptic curve*, https://en.wikipedia.org/wiki/Elliptic_curve

[9] *Prime number*, https://en.wikipedia.org/wiki/Prime_number

[10] *Erdos–Kac theorem*, https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93Kac_theorem

[11] *ECPP*, https://primes.utm.edu/glossary/page.php?sort=ECPP

[12] *Quadratic sieve*, https://en.wikipedia.org/wiki/Quadratic_sieve

[13] *Number Theory Library*, https://en.wikipedia.org/wiki/Number_Theory_Library

[14] *Richard G.E. Pinch*, http://www.chalcedon.demon.co.uk/rgep/rcam.html

[15] *AKS primality test*, https://en.wikipedia.org/wiki/AKS_primality_test

[16] *Primality test*, https://en.wikipedia.org/wiki/Primality_test

[17] A. Bertossi, A. Montresor, *Algoritmi e strutture di dati*, Città Studi Edizioni, terza edizione, pg. 319

[18] *Vocabolario Treccani*, http://www.treccani.it/vocabolario/

[19] Martin E. Hellman, Bailey W. Diffie, Ralph C. Merkle, *Cryptographic apparatus and method*, https://www.google.com/patents/US4200770

[20] T. Elgamal, *A public key cryptosystem and a signature scheme based on discrete logarithms* https://dx.doi.org/10.1109%2FTIT.1985.1057074

[21] M. Campagna, *SEC 4: Elliptic Curve Qu-Vanstone Implicit Certificate Scheme (ECQV)* Standards for Efficient Cryptography, http://www.secg.org/sec4-1.0.pdf

[22] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, P. Zimmermann, *Factorization of a 768-bit RSA modulus*, Cryptology ePrint Archive: Report 2010/006, https://eprint.iacr.org/2010/006

[23] P. Gutmann, *Malware-signing certs with 512-bit keys*, https://groups.google.com/forum/?fromgroups#!topic/mozilla.dev.security.policy/cn1sBm2ibWo

[24] A. Joux, A. Odlyzko, C. Pierrot, *The Past, evolving Present and Future of Discrete Logarithm*, https://www-almasty.lip6.fr/~pierrot/papers/Dlog.pdf

[25] M. Lazzarini, *Crittografia basata su curve ellittiche e implementazione di funzioni di libreria per Cryptokit*, http://amslaurea.unibo.it/3898/1/lazzarini_margherita_tesi.pdf

[26] *Certicom announces elliptic curve cryptography challenge winner*, https://www.certicom.com/news-releases/300-solution-required-team-of-mathematicians-2600-computers-and-17-months-

[27] P. W. Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, https://arxiv.org/abs/quant-ph/9508027

[28] *D-wave Two, il computer quantistico poco quantistico*, Le Scienze, http://www.lescienze.it/news/2014/06/20/news/d-wave_quantum_speed-up_verifica-2186534/

[29] *Qubit connessi attraverso fononi*, Le Scienze, http://www.lescienze.it/news/2015/09/14/news/connettere_qubit_fononi-2762670/

[30] J. Proos, C. Zalka, *Shor's discrete logarithm quantum algorithm for elliptic curves*, https://arxiv.org/abs/quant-ph/0301141

[31] M. Mosca, *Cybersecurity in an era with quantum computers: will we be ready?*, https://eprint.iacr.org/2015/1075.pdf

[32] P. Erdós, C. Pomerance, *On the Number of False Witnesses for a Composite Number*, http://www.ams.org/mcom/1986-46-173/S0025-5718-1986-0815848-X/S0025-5718-1986-0815848-X.pdf

[33] M. O. Rabin, *Probabilistic algorithm for testing primality*, http://www.sciencedirect.com/science/article/pii/0022314X80900840

[34] I. Damgard, P. Landrock, C. Pomerance, *Average case error estimates for the strong probable prime test*, http://www.math.dartmouth.edu/~carlp/PDF/paper88.pdf

[35] *Euclid's lemma*, https://en.wikipedia.org/wiki/Euclid%27s_lemma

[36] M. Agrawal, N. Kayal, N. Saxena, *PRIMES is in P*, http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf

[37] H.W. Lenstra jr., C. Pomerance, *Primality testing with Gaussian periods*, http://www.math.dartmouth.edu/~carlp/aks041411.pdf

[38] F. Morain, *Implementing the asymptotically fast version of the elliptic curve primality proving algorithm*, http://arxiv.org/pdf/math/0502097

[39] G. L. Miller, *Riemann's Hypothesis and tests for primality*, https://dx.doi.org/10.1145%2F800116.803773

[40] E. Bach, *Explicit bounds for primality testing and related problems*, http://www.ams.org/journals/mcom/1990-55-191/S0025-5718-1990-1023756-8/home.html

[41] *Grover's algorithm*, https://en.wikipedia.org/wiki/Grover%27s_algorithm

[42] *Learning with errors*, https://en.wikipedia.org/wiki/Learning_with_errors

[43] D. Gligoroski, S. Samardjiska, *The Multivariate Probabilistic Encryption Scheme MQQ-ENC*, https://eprint.iacr.org/2012/328.pdf

[44] *Quantum key distribution*, https://en.wikipedia.org/wiki/Quantum_key_distribution

[45] *Meccanismi simmetrici*, http://lia.deis.unibo.it/Courses/SicurezzaM1213/CifrariSimmetrici.pdf

[46] *Coppersmith method*, https://en.wikipedia.org/wiki/Coppersmith_method

[47] *Order (group theory)*, https://en.wikipedia.org/wiki/Order_(group_theory)

[48] *Cyclic group*, https://en.wikipedia.org/wiki/Cyclic_group