UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di laurea triennale in

Sicurezza dei sistemi e delle reti informatiche

STUDIO DI TECNICHE DI CRITTOANALISI PER RSA E IMPLICAZIONI PRATICHE PER LA SICUREZZA

Relatore: Prof. Stelvio Cimato

Candidato: Giorgio Oppo

Matricola: 909776

Anno Accademico 2018-2019

dedicato a …

Ringraziamenti

Ci tengo a ringraziare:

I miei genitori e la mia famiglia che

nel corso degli anni mi hanno sempre sostenuto.

I miei amici che sono, negli ultimi due anni,

sempre stati al mio fianco.

I miei professori che mi hanno inspirato e sostenuto

nella mia carriera e nelle mie ricerche.

INDICE

Introduzione

L'algoritmo RSA è estremamente importante nella crittografia, infatti è il primo sistema crittografico moderno che utilizza due differenti chiavi criptografiche una pubblica, alla quale chiunque può esserne a conoscenza e viene di norma usata per cifrare il messaggio, e una privata che permette di decifrare il messaggio cifrato e dovrebbe essere a conoscenza solo da colui che ha generato la chiave pubblica. La forza di questo sistema crittografico si basa sul non riuscire ad ottenere la chiave privata a partire dalla chiave pubblica. Questo algoritmo risulta quindi molto adatto ad essere usato nelle comunicazioni via internet in quanto si può inviare in maniera 'sicura' la chiave criptografica di cifratura che, se venisse spiata non creerebbe nessun problema alla comunicazione poiché non porterebbe comunque a decifrare i messaggi cifrati. Dato questo elevato grado di sicurezza RSA viene estremamente utilizzato, dall'uso dei certificati, utili a stabilire se l'origine dei messaggi risulta essere quella dichiarata, alla crittografia end-to-end, la quale consente di scambiare messaggi tra due individui in maniera confidenziale pur utilizzando la rete internet che per definizione è un canale condiviso. Mediante l'uso dei certificati e di un canale riservato che implementa anche l'uso di tecniche di controllo dell'integrità è possibile eseguire pagamenti online in maniera sicura. Possiamo quindi affermare che RSA ha contribuito allo sviluppo e alla crescita di colossi commerciale come Amazon, Ebay, Alibaba… si noti che soltanto Amazon ha fatturato 3,56 miliardi di dollari nel 2019.

Nel corso degli anni, data l'importanza dell'algoritmo, RSA è stato soggetto ad innumerevoli analisi che sono sfociate poi in svariati attacchi, alcuni di tipo matematico (Cycle attack, Quadratic Sieve, General Number Field Sieve, Wiener Attack, Johannes Blömer Attack) altri legati all'implementazione dell'algoritmo (moduli comuni, low exponent).

In questo documento illustrerò gli attacchi sopra citati e mi concentrerò particolarmente sull'analisi del Cycle attack, poiché ritengo che sia un attacco che non sia stato sufficientemente esaminato. Analizzandone le caratteristiche e studiando il funzionamento del Toziente di Eulero sono riuscito a definire, tramite probabilità, la quantità di fattori primi differenti generati della funzione di Eulero applicata al modulo di RSA. Utilizzando quest'ultima ho potuto ridurre notevolmente la complessità dell'attacco, illustrerò quindi questa variante dell'attacco evidenziando i miglioramenti computazionali che ne derivano.

Per concludere riporterò le differenze computazionali tra i vari attacchi descritti e fornirò il codice che ho scritto e utilizzato per implementare il Cycle attack.

La tesi sarà organizzata come segue, nel **capitolo I** introdurrò i concetti base della crittografia e il funzionamento dell'algoritmo RSA così com'era stato ideato originariamente. Nel **capitolo II** descriverò i vari attacchi che ha subito RSA. Nel **capitolo III** descriverò dapprima il funzionamento del Cycle attack andandone poi ne descriverò la sua complessità, in seguito illustrerò il funzionamento delle stime probabilistiche e come migliorano il Cycle attack. Infine ne descriverò una sua variante che prende il nome di Recursive Cycle Attack.

Vi saranno anche due appendici, **appendici 1** dove vi sarà inserito il codice sorgente debitamente commentato, **appendice 2** dove vi saranno inserite tutti i teoremi e i concetti matematici utili per comprendere al meglio questa tesi.

CAPITOLO I: *LA CRITTOGRAFIA*

La crittografia, è una parola composta derivante dall'unione di due parole greche κρυπτός [*kryptós*] ovvero "nascosto", e γραφία [*graphía*] ovvero "scrittura" ed è la scienza che studia i meccanismi per nascondere il significato dei messaggi. Ad oggi la crittografia è prevalentemente impiegata mediante sistemi di cifratura, ma non è l'unica forma di crittografia infatti esistono altri metodi crittografici come la steganografia, la quale si prefigge il compito di nascondere il messaggio senza però garantire che se il messaggio venisse trovato non sia possibile comprenderlo. Fu ampiamente utilizzata durante la guerra fredda, tramite l'uso di microfilm e anche ad oggi è una tecnica usata, si basti pensare che dei ricercatori sono riusciti a creare un passaggio di informazioni da un computer isolato dalla rete[1] codificando le informazioni attraverso piccole variazioni della luminosità di un monitor che non risulta visibile ad occhio nudo, a dimostrazione del fatto che questa tecnica può risultare altrettanto efficacie rispetto alla cifratura.

1 I sistemi di cifratura

La cifratura è quindi un metodo di crittografia derivante dal concetto che la sicurezza non deve dipendere dall'offuscamento, ovvero il grado di sicurezza deve dipendere soltanto dalla chiave di cifratura utilizzata e dalla funzione matematica di cifratura/decifratura.

Nel corso degli anni si è cercato quindi di definire i sistemi di cifratura in maniera sempre più formale fino ad esprimere la cifratura come l'insieme di tutti i sistemi di cifratura *S* dove *S* è anch'esso un insieme definito come:

S= {M , C , K, E, D}

> *M* è l'insieme di tutti i testi possibili da cifrare *m,* ovvero il dominio della funzione crittografica
>
> *C* è l'insieme di tutti i testi cifrati *c,* ovvero il codominio della funzione crittografica
>
> *K* è l'insieme delle chiavi ovvero dei segreti che permettono di ottenere il testo cifrato *c:C* a partire da un messaggio *m:M* e ottenere il messaggio *m:M* a partire da *c:C*

E ovvero la funzione matematica che definisce come si ottenga un testo cifrato *c* a partire da un messaggio *m*

D ovvero la funzione matematica che definisce come si ottenga un messaggio *m* a partire da un testo cifrato *c*

Data questa definizione formale possiamo categorizzare ogni sistema crittografico in due categorie a seconda della loro sicurezza intrinseca:

- **incondizionatamente sicuro:** un sistema di cifratura si definisce tale se non è possibile risalire al messaggio senza avere la chiave di decifratura indipendentemente dal numero di testi cifrati di cui si è in possesso, dal tempo e dalle risorse di calcolo che ha a disposizione.
- **computazionalmente sicuro:** un sistema di cifratura si definisce tale se la sua sicurezza può essere garantita poiché non è possibile risalire al messaggio a partire da un testo in cifrato senza avere la chiave crittografica in un tempo non polinomiale.

Ad oggi esistono diversi sistema di cifratura incondizionatamente sicuri il più famoso è il one-time-pad, ma come si evince dal nome, una stessa chiave può essere utilizzata solo una volta per eseguire la cifratura altrimenti il messaggio può essere facilmente decodificato, inoltre la chiave di questo cifrario deve essere delle stesse dimensioni del testo e quindi grava non poco sullo spazio utilizzato (Es se volessi cifrare un hard disk da 1TB dovrei utilizzare una chiave anch'essa da 1TB e quindi mi servirebbero 2TB per salvare il messaggio cifrato e la chiave di decodifica).

2 Le proprietà di sicurezza

La crittografia è uno strumento estremamente potente ma non può garantire tutto, ogni sistema di crittografia mira a garantire una o più delle seguenti quattro principali proprietà di sicurezza nella comunicazione tra due o più individui:

1. Confidenzialità: ovvero la proprietà che se rispettata determina che nessun utente non autorizzato possa riuscire a comprendere il messaggio.

2. Integrità: ovvero la proprietà che se rispettata determina che nessun utente non autorizzato possa riuscire a modificare il messaggio.
3. Autenticazione: ovvero la proprietà che se rispettata determina il riconoscimento degli interlocutori.
4. Non Ripudiabilità: ovvero la proprietà che se rispettata determina che un utente che genera un messaggio non possa disconoscerlo.

Queste proprietà possono essere applicate anche quando la crittografia non è applicata nella comunicazione come ad esempio cifrare un computer affiche solo il proprietario possa leggere il contenuto del disco, poiché questa operazione può essere vista come una comunicazione tra il proprietario e sé stesso.

3 Crittografia moderna

La crittografia moderna è volta a risolvere un problema che si è fatto sempre più rilevante con l'avvento delle telecomunicazioni ovvero la trasmissione della chiave crittografica, poiché fino ad ora la chiave con la quale si eseguiva la cifratura e la chiave di decifratura erano uguali un attaccante poteva intercettare lo scambio della chiave criptografica e quindi un sistema criptografico non poteva più garantire le sue proprietà di sicurezza.

Viene così negli anni '70 introdotto il concetto di crittografia asimmetrica dove la chiave di cifratura e decifratura non corrispondono ma sono legate da un'espressione matematica, il più celebre algoritmo di crittografia asimmetrica è RSA e deriva da l'algoritmo di scambio delle chiavi Diffie-Hellman.

3.1 L'agorimo crittografico RSA

RSA fu inventato nel 1977 da Ronald Rivest, Adi Shamir e Leonard Adleman ed è metodo crittografico asimmetrico ad oggi più ampiamente utilizzato, il suo campo di applicazione spazia dai certificati, con i quali si possono garantire le proprietà di autenticazione e non ripudio, all'utilizzo di funzioni Mac, con le quali si possono garantire le proprietà di integrità autenticazione e integrità, all'utilizzo come meccanismo crittografico dei messaggi con la quale si possono garantire la proprietà di confidenzialità e autenticazione. Utilizzando quindi le funzioni Mac, i certificati e la

crittografia del messaggio si riesce a garantire tutte le proprietà di sicurezza nella comunicazione.

Il primo passo per utilizzare RSA consiste nella selezione casuale di due numeri *P,Q* molto grandi il cui prodotto *N* è detto modulo di RSA. Successivamente si genera una chiave pubblica $K_{pubblica}$ composta da *N* e da un numero *E* che rispetta le seguenti condizione $gcd(E, \varphi(N)) = 1$ e $E \neq 1\ mod[\varphi(N)]$ questa sarà la chiave di cifratura dove con $\varphi(N)$ si indica la funzione Toziente di Eulero applicata al numero *N*. Si genera quindi la chiave privata $K_{privata}$ composta da *N* e da un numero *D*, questa sarà la chiave di cifratura. Queste due chiavi sono legate da una correlazione del tipo $E * D\ mod(\varphi(N)) = 1$

Vengono così definite due chiavi:

- $K_{pubblica}$(N,E) chiave pubblica alla quale chiunque può avere accesso, che consente solo la cifratura.
- $K_{privata}$(N,D) chiave privata alla quale solo chi ha generato le chiavi ha accesso, che consente solo la decifratura.

Il processo di cifratura può essere quindi descritto mediante una funzione matematica dove, dato un testo in chiaro *m,* si genera un testo cifrato *c* applicando la potenza modulare come segue:

$$c = m^E mod(N)$$

Il processo di decifratura può essere anch'esso espresso mediante la stessa funzione matematica, la quale dato un testo in cifrato *c* genera un testo in chiaro *m* applicando la potenza modulare:

$$m = c^D mod(N)$$

3.1.1 Correttezza dell'algoritmo

Per dimostrare la correttezza Di RSA bisogna, date le due funzioni di cifratura e decifratura, bisogna verificare la correttezza della decifratura rispetto a ogni testo cifrato. Quindi dimostrando che per ogni testo cifrato

c, ottenuto a partire da un qualsiasi *m*, si riesce ad ottenere *m* utilizzando la chiave di decifratura *D* possiamo definire l'algoritmo corretto.

$c^D mod(N) = (m^E)^D mod(N) =$ { per sostituzione

$= m^{E*D}\ mod(N) = m^{(1+x*\varphi(N))}\ mod(N) =$ {per definizione delle chiavi

$= m^{1+x*\varphi(N)} mod(N) = m^1 * \left(m^{\varphi(N)}\right)^X mod(N) =$ { per scomposizione

$= m^1 * \left(m^{\varphi(N)}\right)^x mod(N) = m^1 * 1^x mod(N) =$

{ per Teorema di Eulero $m^{\phi(N)}$ mod(N) =1

$= m$ { dato che m < N & 1^X=0 per ogni x>0

3.1.2 Esempio applicativo di RSA

In questa sezione illustrerò i vari passaggi crittografici che portano allo scambio del messaggio "ciao Mario" secondo quanto era stato originariamente definito il sistema RSA, utilizzando RSA con 2048bit di modulo.

1) Vengono generati due numeri P e Q:

   P=166110523582992887160649054868816407731251165218887793661310886870981387792012053555740837434771626714062276175699004592914172089531219864191061927342547910905446660900734709205172513491543021763505280967668724868077363193762933125402682836423266866417248534733248292422181157386038169582705948644505964443187

   Q=154742875164331293915132630285507918877582420947760790616185640942021817038148582862935187528170855509629520087740080493822602532125705525962099578066402028653769974368567470237476349547483773940985290024334563407029537970266392737279054664302611919617991132311867207659428388534184522701445168162194543864667

2) i ricava il modulo di RSA: N=P*Q=

   166110523582992887160649054868816407731251165218887793661310886870981387792012053555740837434771626714062276175699004592914172089531219864191061927342547910905446660900734709205172513491543021763505280967668724868077363193762933125402682836423266866417248534733248292422181157386038169582705948644505964443187 * 154742875164331293915132630285507918877582420947760790616185640942021817038148582862935187528170855509629520087740080493822602532125705525962099578066402028653769974368567470237476349547483773

940985290024334563407029537970266392737279054664302611919617991132311867207659428388534184522701445168162194543864667=

257044200142847777371027959379531306757904552571855192781654448538573654708647760115359956174075851825808012055158027821273978438838546189797048038796783050918651494636228179073103475435430042632537959262625926924080652866955936329139290031167934859918628320995337405158137901994591462822460627611996148162459383938967738177060498658435913131154222943247243808455507642005972406022289745869310014990646003414735697579454940472859251374065061775511779600989911145171901342579463571265151453102231757336594621159583289475060180543345705866231689351798070856312416745957315676904018858706602798140505306520036857381737 29

3) Si calcola Phi(N):

φ(N)=(P-1)*(Q-1)=
(166110523582992887160649054868816407731251165218887793661310886870981387792012053555740837434771626714062276175699004592914172089531219864191061927342547910905446660900734709205172513491543021763505280967668724868077363193762933125402682836423266866417248534733248292422181157386038169582705948644505964443187-1) *
(154742875164331293915132630285507918877582420947760790616185640942021817038148582862935187528170855509629520087740080493822602532125705525962099578066402028653769974368567470237474634954748377394098529002433456340702953797026639273727905466430261191961799113231186720765942838853418452270144516816219454386466 7-1)=

257044200142847777371027959379531306757904552571855192781654448538573654708647760115359956174075851825808012055158027821273978438838546189797048038796783050918651494636228179073103475435430042632537959262625926924080652866955936329139290031167934859918628320995337405158137901994591462822460627611996148162456175404980264935249740841584369887888134607385577322612732676727842373973988139505123254741016578592498779616820549621991883627848492521610247985935821645776309176226770549470724964471841489379549715449663256592309111531705412607604871976790812068452064349286864521903202763247400571217663795351969852298658 76

4) Si sceglie un numero casuale *E* e si verifica che sia un esponente valido per RSA, scegliendo *E* come numero primo la probabilità che sia coprimo con, *Phi(N*) aumenta e quindi anche la probabilità che sia valido:

E=65537

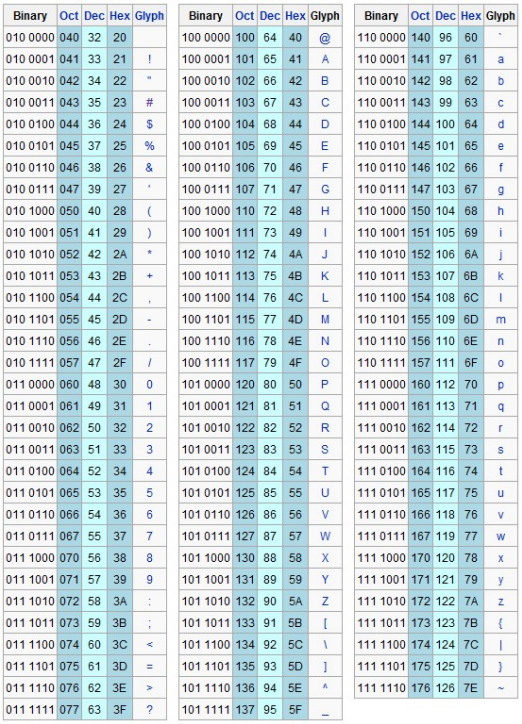
gcd(E, Phi(N))=
gcd(65537,25704420014284777737102795937953130675790455257185519278165444853857365470864776011535995617407585182580801205515802782127397843883854618979704803879678305091865149463622817907310347543543004263253795926262592692408065286695593632913929003116793485991862832099533740515813790199459146282246062761199614816245617540498026493524974084158436988788813460738557732261273267672784237397398813950512325474101657859249877961682054962199183627848492521610247985935821645776309176226770549470724964471841489379549715449663256592309111531705412607604871976790812068452064349286864521903202763247400571217663795351969852298658765876)=1

5) Si calcola l'elemento inverso di E ovvero D:

D=E^-1 mod[Phi(N)] = 65537^-1
mod(25704420014284777737102795937953130675790455257185519278165444853857365470864776011535995617407585182580801205515802782739784388385461897970480387967830509186514946362281790731034754354300426325379592626259269240806528669559363291392900311679348626832099533740515813790199459146282246062761199614816245617498026493524974084158436988788813460738557732261273267672784237397398813950512325474101657859249877961682054962199183627848492521610247985935821645776309176226770549470724964471841489379549715449663256592309111531705412607604871976790812068452064349286864521903202763247400571217663795351969852298658765876)=
183088641635538353837633034343895828780166049996288695763315551707114405228411219462976915546363654715359931195306954799958658277056047342507530089858745893158575620201012674173543850600444173827448685114342141000396859125110549213124665424797448268853647384719827660866183002014274746723458268726911396499254111785401891259019991031417366842188473863731420953001891665507002301897855262527101616713694476483153570216714223673714132774335039434232387601377391248401420398483938477804033020530540497211144243204689419426973795495850288629867144189519994784128214838779012202984015261472937517204201651446134316113854 61

6) Avendo quindi tutti i parametri di RSA, si passerà alla fase di cifratura dove dobbiamo trasformare il testo in chiaro in un numero tramite conversione Base64 o ASCII:

MessaggioInChiaro= conversione carattere per carattere del Testo_in_chiaro secondo la tabella allegata(ASCII)=



Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=
011 1110	076	62	3E	>
011 1111	077	63	3F	?

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	@
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g
110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}
111 1110	176	126	7E	~

‘c’ =099

‘i’ =105

‘a’ =097

‘o’ =111

‘ ’ =032

‘M’ =077

‘a’ =097

‘r’ =114

‘i’ =105

‘o’ =111

MessaggioInChiaro
=099105971103207709711410511 1

7) Si esegue la cifratura del messaggio precedentemente convertito in una sequenza numerica:

MessaggioCifrato= $MessaggioInChiaro^{E}$ mod(N) =
$099105971103207709711410511 1^{65537}$mod(257044200142847777371027959379531306757904552571855192781654448538573654708647760115359956174075851825808012055158027821273978438838546189797048038796783050918651494636228179073103475435430042632537959262625926924080652866955936329139290031167934859918628320995337405158137901994591462822460627611996148162459383938967738177060498658435913131154222943247243808455507642005972406022289745869310014990646003414735697579454940472859251374065061775511779600989911145171901342579463571265151453102231757336594621159583289475060180543345705866231689351798070856312416745957315676904018858706602798140505306520036857381737 29)=
105681076775388517738788098566822222304469653350953502054087754416287312424937096818796514886678582527169882920817731347850851614839310380180974834805823645016425640559344338457437773228387565701405740798857829496524293423718132807049148678162790486552475 7

6985407635373016456101202538558694665613624955101639194053608084
6587657052785544288345536294641950993251591966712385400672414651
5243479269817853519818154623654353621097255776717536395539544658
3291556355852860895390833971781220343721589115936715197584173266
6603159230822116180631517128016605092823033636828609122621833628
3348959499891187903735615631486643588883 9

8) Il MessaggioCifrato verrà inviato e si eseguirà la decifratura del messaggio:

MessaggioInChiaro= $MessaggioCifrato^{D}$ mod(N)=
1056810767753885177387880985668222223044696533509535020540877544
1628731242493709681879651488667858252716988292081773134785085161
4839310380180974834805823645016425640559344338457437773228387565
7014057407988578294965242934237181328070491486781627904865524757
6985407635373016456101202538558694665613624955101639194053608084
6587657052785544288345536294641950993251591966712385400672414651
5243479269817853519818154623654353621097255776717536395539544658
3291556355852860895390833971781220343721589115936715197584173266
6603159230822116180631517128016605092823033636828609122621833628
33489594998911879037356156314866435888839^{183088641635538353837633034343895828780166049996288695763315551707114405228411219462976915546363654715359931195306954799958658277056047342507530089858745893158575620201012674173543850600444173827448685114342141000396859125110549213124665424797448268853647384719827660866183002014274746723458268726911396499254111785401891259019991031417366842188473863731420953001891665507002301897855262527101616713694476483153570216714223673714132774335039434232387601377391248401420398483938477804033020530540497211144243204689419426973795495850288629867144189519994784128214838779012202984015261472937517204201651446134316113854 61

mod(2570442001428477773710279593795313067579045525718551927816544485385736547086477601153599561740758518258080120551580278212739784388385461897970480387967830509186514946362281790731034754354300426325379592626259269240806528669559363291392900311679348599186283209953374051581379019945914628224606276119961481624593839389677381770604986584359131311542229432472438084555076420059724060222897458693100149906460034147356975794549404728592513740650617755117796009899111451719013425794635712651514531022317573365946211595832894750601805433457058662316893517980708563124167459573156769040188587066027981405053065200368573817372 9)=
991059711032077097114105111

9) Il messaggio verrà ricostruito a partire dal messaggio decifrato tramite conversione Base64 o ASCII:

Testo_in_chiaro = conversione carattere per carattere del MessaggioInChiaro secondo la tabella allegata=

099=’c’

105=’i’

097=’a’

111=’o’

032=’ ’

077=’M’

097=’a’

114=’r’

105=’i’

111=’o’

Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=
011 1110	076	62	3E	>
011 1111	077	63	3F	?

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	@
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g
110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}
111 1110	176	126	7E	~

3.2 Costo computazionale

L’algoritmo RSA è estremamente costoso computazionalmente ed il suo costo è proporzionale alla scelta dei due numeri primi che compongono il modulo.

Dato che non esiste un algoritmo efficiente per generare numeri primi, bisogna generare casualmente un numero nell’intervallo interessato e poi controllare che esso sia primo.

Se vogliamo generare un numero primo x in un intervallo [2^x , 2^y] allora la probabilità che esso sia primo è calcolabile come casi favorevoli fratto casi contrari. Dato che tramite il teorema dei numeri primi possiamo calcolare quanti sono i numeri primi in un intervallo si ottiene la seguente formula

$$\left(\frac{2^x}{\ln(2^x)} - \frac{2^y}{\ln(2^y)}\right) * \frac{1}{2^y}$$

quindi la probabilità che x preso nell'intervallo [2^1024 2^1023] sia primo è circa $\frac{1}{709,78}$. Se scegliessimo solo x dispari nello stesso intervallo allora dimezzeremo i casi totali lasciando però inalterati i casi favorevoli, ottenendo quindi una probabilità di $\frac{1}{335,23}$. Quindi statisticamente dovremmo generare 356 numeri differenti prima di ottenerne uno primo.

Inoltre vi è un ulteriore costo, quello di trasporto: prendendo in considerazione l'esempio precedente possiamo notare che per inviare in rete un messaggio di piccole dimensioni come "Ciao Mario" se questo viene cifrato con RSA bisognerà inviare tutti i bit della chiave pubblica, 2048bit del modulo più 17bit dell'esponente, e anche tutti i bit del messaggio cifrato ovvero 2048bit. Quindi in totale per inviare il messaggio "Ciao Mario" che è di 80bit bisognerà utilizzarne ben 2048bit+17bit+2048bit = 4114bit

3.2.1 Test di primalità

Vi è in oltre un altro fattore importante che determina l'aumento del costo computazionale ovvero la difficoltà di determinare se un numero sia effettivamente primo. Ci sono due approcci per determinarlo ovvero

- I test di primalità deterministici, i quali come dice il nome individuano in maniera esatta se un numero è primo ma hanno un costo computazionale elevato. L'algoritmo di questo tipo più efficiente fu ideato nel 2002 e prende il nome di algoritmo AKS, con il quale si riduce il tempo computazionale, arrivando ad avere una complessità di O(log(x) ^{6})
- Test di primalità probabilistici, i quali non riescono a determinare con esattezza la primalità ma determinano con un'elevata probabilità che un numero sia primo, più passi ricorsivi esegue più diminuisce il suo fattore di errore.

3.2.2 Potenza modulare

Anche il processo di cifratura/decifratura ha un costo notevole, l'esecuzione della potenza modulare effettuata in maniera naif può aggravare non soltanto sui tempi computazionali ma anche sull'utilizzo delle risorse.

Se questa operazione venisse calcolata eseguendo prima la potenza e successivamente il modulo potrebbe anche portare al fallimento dell'operazione in quanto potrebbero essere necessari anche 10^301 GB per scrivere il risultato di una potenza con una base di 1024 bit.

Un'altra implementazione naif prevede di eseguire a seguito di ogni operazione moltiplicativa il modulo ed eseguire ciò ricorsivamente fino ad aver eseguito tante moltiplicazioni quanto espresso dal valore dell'esponente. Se l'esponente è di 512 bit allora dovranno essere eseguite 2^512 operazioni.

Furono così introdotti due metodi che determinano un notevolmente abbassamento del costo computazionale per le operazioni di potenza modulare:

- Metodo right-to-left
- Metodo left-to-right

Questi due metodi consistono nello scomporre bit a bit l'esponente della potenza ed eseguire dei prodotti nel primo caso e delle somme nel secondo, sfruttando poi le proprietà delle potenze per ridurre il numero di operazioni. Utilizzando tali metodi il costo della potenza modulare è pari al numero di bit di cui è composto l'esponente.

Il costo computazionale risulta essere proporzionato al numero di bit dell'esponente, dato che il prodotto tra l'esponente utilizzato per la cifratura e quello usato per la decifratura deve essere maggiore di $\phi(N)$ per definizione, si può scegliere di utilizzare un esponente piccolo per il processo di cifratura in modo da alleggerire il costo dell'operazione su un dispositivo dotato di poca potenza di calcolo e appesantire il processo di decifratura o viceversa.

3.3 La Sicurezza di RSA

L'algoritmo RSA è un sistema computazionalmente sicuro da qui nasce l'importante compito di analizzarne la sicurezza. Per essere definito computazionalmente sicuro abbiamo bisogno di affermare che non esista un metodo per cui un testo cifrato possa essere decifrato in un tempo polinomiale. Analizzando l'algoritmo possiamo affermare che la sicurezza di RSA si basa su due problemi matematici ancora irrisolti efficientemente:

- Logaritmo discreto
- Fattorizzare un numero

Detto ciò sappiamo che questi due problemi matematici ad oggi non sono stati ancora risolti in un tempo polinomiale e quindi RSA può essere definito sicuro.

3.3.1 Il problema del logaritmo discreto

Così come il logaritmo è l'operazione inversa dell'esponenziale, alla stessa maniera il logaritmo discreto è l'operazione inversa della potenza discreta.

Si immagini di avere un insieme A contenente i numeri interi compresi fra 0 e *p - 1*, dove *p* è un numero primo:

A={0,1,2,3,4,…,p-1}

Definiamo un'operazione che per convenienza qui denoteremo * su due numeri *a,b* $\in A$: $a * b = ab \, mod(p)$

dove l'operazione * è detta potenza discreta, per risolvere questo problema dovremmo determinare un numero naturale c >0 tale per cui

$(a^b)^c$= a mod A

Quindi dovremmo conoscere un c tale che

a^{b*c} = a mod A

Ciò vorrebbe dire conoscere un c tale per cui b*c sia uguale ad 1 modulo ϕ(A) e quindi dovremmo determinare ϕ(A), il che risulta impossibile senza riuscire a fattorizzare A.

3.3.2 Il problema della fattorizzazione

Fattorizzare un numero, ovvero scomporlo in numeri primi, di per sé non è impossibile, basta pensare alla scomposizione di 1000, ma quando il numero da fattorizzare è grande questo processo risulta molto arduo.

Questo problema è stato analizzando fin dai tempi di Euclide (IV sec A.C.) senza però trovare un metodo efficace i più di 2000 anni. In tempi recenti sono state proposte innumerevoli sfide di fattorizzazione con premi in denaro che hanno raggiunto anche la cifra di 200.000 dollari.

Il problema principale è dato dall'impossibilità di avere una lista di tutti i possibili fattori nel quale un numero possa essere scomposto, nell'esempio precedente conosciamo tutti i numeri primi minori di 1000 e quindi possiamo banalmente provare se ognuno di essi lo divide.

Il problema di fattorizzare un numero grande è subordinato al fatto che non esiste una formula efficiente per calcolare i numeri primi da cui ne deriva che non conosciamo tutti i numeri primi minori del numero che vogliamo fattorizzare se esso è molto grande.

Tuttavia, nel corso dell'ultimo secolo sono state adottate diverse strategie per ridurre i tempi di ricerca dei vari fattori primi che compongono un numero, ne illustrerò alcuni nel capitolo successivo.

CAPITOLO II: Attacchi ad RSA

Nel corso degli anni, data l'importanza dell'algoritmo, RSA è stato soggetto ad innumerevoli analisi che sono sfociate poi in svariati attacchi, alcuni di tipo matematico (cycle attack, Quadratic Sieve, General Number Field Sieve, Wiener Attack, Johannes Blömer Attack) altri legati all'implementazione dell'algoritmo (moduli comuni, low exponent).

Per cercare di prevenire quest'ultima tipologia di attacchi sono stati introdotti numerosi standard. Nel documento RFC8017 [2] sono stati definite varie modalità per implementare correttamente l'algoritmo. In questo documento si definiscono le primitive crittografiche, gli schemi di cifratura e di firma, e di verifica del messaggio.

Illustrerò da prima due attacchi storici legati all'errata implementazione dell'algoritmo per passare poi a descrivere una serie di attacchi legati alla fattorizzazione.

1 Moduli comuni

È una tipologia di attacco storica, si basa sulla proprietà delle potenze. Poniamo il caso che un utente malevolo (Trudy) riesca a intercettare la comunicazione verso un server, e che quest'ultimo fornisca due chiavi pubbliche a Bob K(n,e2) e ad Alice K(n,e1), Trudy ha lo scopo di scoprire cosa comunica Alice al server poiché potrebbe essere la password di accesso al server. Queste due chiavi hanno *n* in comune e *e1≠e2, s*e Alice e Bob inviano lo stesso messaggio *M* al server, ovvero entrambi inviano la password di accesso al server, Trudy potrà leggere il contenuto dei due pacchetti se gcd(e1,e2)=1 e quindi ottenere l'accesso al server poiché:

messaggio cifrato da Alice: $C1=M^{e1} \bmod n$,

messaggio cifrato da alice Bob: $C2=M^{e2} \bmod n$

$$c1^x * c2^y = (m^{e1})^x (m^{e2})^y = m^{e1*x} * m^{e2*y} = m^{e1*x+e2*y} = \mathrm{M}$$

Dove x e y sono due numeri interi positivi ricavabili grazie al teorema di Euclide esteso.

2 Low Exponent Attack

È un attacco che si basa su una caratteristica del modulo, ovvero se un numero in modulo è minore del modulo stesso allora l'operazione di modulo lascerà inalterato il numero.

Sia (e,N) una chiave pubblica di RSA e vi sia testo in chiaro *m*, se conosciamo l'ordine di grandezza di m da cui riusciamo ad affermare che $m^e < N$ allora possiamo affermare che il messaggio cifrato *c* non è in modulo *N* da cui utilizzando la potenza inversa di *e* ovvero $\frac{1}{e}$ possiamo decifrare il messaggio cifrato.

Se $C < N \rightarrow m = c1/e$

questo attacco ritrova un'applicazione pratica quando si utilizza RSA per trasferire in maniera sicura un chiave crittografica simmetrica senza usare dei bit di riempimento. Al fine di prevenire questo tipo di attacca bisogna controllare che:

$$e > \log2(N)$$

2.1 Applicazione reale dell'attacco Low Exponent

Poniamo il caso di voler scambiare la chiave criptografica di AES *m*, dove la dimensione massima della chiave è di 256 bit, utilizzando RSA a 2048bit.

m=231584178474632390047141970017375815706539969331281128078915168015826259279872

e=3

N=25082575430177714668185764839756381086500862523429533676316435357732812082572457255743570391499607434079346983223958080489373925863940774890257647439900722040806033771265811579021842956529750062559323295833301503517658358890059659995203004735376460861813429590364553454568147202841699608838423818144983536476053263775415701829003314700405761146144000671098371294162047705635923425611717371124931678581399931372538454402687755544363602265338509634993194539029303803980938310167225827372053476968451307453815190669091773002325338759593525045725987775823620807377715127223753018129930774885099395783077410652077117943313

$C=m^e \bmod N =$

=2315841784746323900471419700173758157065399693312811280789151680158262592798723=

=1242014473840567135247687978025108873078607082545163997982308685988299735190156634697785849344024343669232640052901567982506680176265242693551702320564237762787260934975947764440212228505278755860210399354973175000714277483052846284 8

La vittima invia il messaggio cifrato *C* convinta che questo non possa essere letta da nessuno se non il destinatario dato che ho utilizzato RSA a 2048bit; un attaccante intercetta la comunicazione e computare la radice e esima del testo cifrato per ottenere il la chiave AES scambiata.

$m= C^{1/3}$ =

=1242014473840567135247687978025108873078607082545163997982308685988299735190156634697785849344024343669232640052901567982506680176265242693551702320564237762787260934975947764440212228505278755860210399354973175000714277483052846284 8$^{1/3}$

=2315841784746323900471419700173758157065399693312811280789151680158262592798 72

3 Algoritmi di fattorizzazione

Gli algoritmi deterministici capaci di fattorizzare numeri molto grandi sono molto onerosi, essi si basano sul crivello dei campi di numeri generale :

- Quadratic Sieve
- General Number Field Sieve (NFS)

Questi due algoritmi tuttavia non sono i più efficienti in assoluto infatti, esiste un algoritmo non deterministico molto efficiente, l'algoritmo di Shor, che riesce a stimare con un'elevata probabilità i fattori di un numero. Poiché esso è un algoritmo non deterministico si presta bene ad essere utilizzato nei computer quantistici, la sua complessità è molto bassa ed è pari a

$$\log^2(N) * \log(\log(N)) * \log\left(\log(\log(N))\right)$$

Questo algoritmo permetterebbe di rompere RSA con un modulo a 2048 bit in sole 6.223.632 operazioni, ma data la difficoltà ancora attuale nel creare un computer quantistico non può ancora essere considerato una minaccia per il sistema crittografico. Tuttavia, sono stati presentati diversi attacchi basati su meccanismi di fattorizzazione differenti e specifici per RSA come Johannes Blömer attack e Fattorizzare ϕ(n) usando le approssimazioni Diophantine e lattice basis reduction.

3.1 Quadratic Sieve

Il Quadratic Sieve[3] è l'algoritmo deterministico più veloce conosciuto per fattorizzare numeri con meno di 100 cifre decimali ovvero circa 333 bit. Dato il numero n da fattorizzare bisogna trovare due numeri *x* e *y* tali che $x^2 = y^2 mod\ n$ per $x \neq |y|\ mod\ n$, e dato $x^2 = y^2 mod\ n = x^2 - y^2 = 0\ mod\ n$ e che $x^2 - y^2 = (x + y)(x - y)$ allora almeno un fattore di n sarà uguale a $x - y$ con una probabilità del 50% da cui quindi un fattore di n sarà gcd(x-y,n).

Dato che questo algoritmo ha successo al 50% per implementare efficacemente questo algoritmo dobbiamo disporre di un insieme di numeri primi sufficientemente grande (factor base):

- Si ricavano diversi interi *x* tali che i fattori primi di x^2 mod n siano nella factor base
- Si prendono i prodotti dei diversi *x* in maniera tale che tutti i fattori primi di x^2 vengano utilizzati un numero pari di volte
- Avremo una congruenza del tipo $x1^2 * x2^2$ mod n che ci porta alla fattorizzazione di n in modo efficiente

La complessità di questo algoritmo è mediamente:

$$O\left(e^{(1+O(1))\,(\ln(n)*\ln(\ln(n)))^{\frac{1}{2}}}\right)$$

3.2 General Number Field Sieve (GNFS)

L'algoritmo General Number Field Sieve[8] è un algoritmo che deriva dalla teoria dei numeri. Dato un numero *n* si creano taniti polinomi *f(x)* compresi in Z_n che per qualche valore di x generano il numero *n*. questi polinomi *f(x)* devono però rispettare tre regole:

- Il grado del polinomio *d* deve essere maggiore o uguale a 3
- Deve essere irriducibile in Z_n
- Deve esistere un valore *m* per cui il polinomio generi il numero *n*

Definito il grado del polinomio *d*, si determina il valore *m* chiuso rispetto a $n\text{^}(1/d)$, verificato che m è valido ovvero che è irriducibile in Z_n si procede nella prossima fase.

In questa fase bisogna cercare due polinomi *g(x)* e *h(x)* appartenenti a Z_n il cui prodotto sia uguale al polinomio *f(x):*

$$f(x) = g(x) * h(x)$$

Questa è la fase più importante dell'algoritmo, difatti è qui che vengono generati i due fattori di n, la sfida di trovare i due polinomi più adatti a questa fase è tutt'ora motivo di studio. In media questo algoritmo risulta avere complessità:

$$O\left(e^{((\frac{64}{9})^{\frac{1}{3}}+\mathrm{O}(1))(ln^{\frac{1}{3}}(n)*(\ln(\ln(n)))^{\frac{2}{3}}}\right)$$

3.3 Wiener Attack

Il Wiener Attack[4][7] è un attacco storico e molto importante per RSA, difatti il suo inventore Wiener dimostrò che poteva decifrare RSA in un tempo polinomiale se la chiave di decifratura *d* è abbastanza piccola, con $d < \frac{1}{3}N^{\frac{1}{4}}$.

L'attacco si effettua secondo le seguenti osservazioni:

$$\lambda(n) = lcm(p-1, q-1) = \frac{(p-1)*(q-1)}{G} = \frac{\varphi(n)}{G}$$

dove G=gcd(p-1,q-1)

Da cui si deriva che il prodotto delle chiavi di cifratura e decifratura sono congrue a 1modulo $\lambda(n)$ dato che sono congrue a $\varphi(\mathrm{n})$ che è multiplo di $\lambda(n)$

$e * d = 1\ mod(\ \lambda(n))$→ $e * d = \ K * \ \lambda(n) + 1$

Da cui

$e * d = K/G \ * \ \varphi(n) \ + 1$→ $k = \frac{K}{\gcd(K,G)}$

$e * d = K/G \ * \ \varphi(n) \ + 1$→ g= $\frac{G}{\gcd(K,G)}$

Supponendo quindi che G non sia troppo grande vale l'equazione $e * d \ > \ p * q$ e quindi $e * d * g = k * \ \varphi(n) + g$

Il Wiener attack è estremamente importante infatti moltissimi studi e attacchi attuali si basano ancora su questo.

3.4 Johannes Blömer attack

Questo attacco è una generalizzazione del Wiener Attack[5], può essere utilizzato se N e $\varphi(N)$ hanno lo stesso ordine di grandezza e esiste un'equazione $ex \ + \ y \ = \ 0\ mod\ \varphi(N)$

per $0 < x < \frac{1}{3} * N^{\frac{1}{4}}$ e $|y| = \ O\left(N^{-\frac{3}{4}}ex\right)$.

Date queste premesse si calcola tramite la formula di espansione della frazione $\frac{e}{d}$ e per ogni convergenza a $\frac{k}{x}$ si calcola

- $s = \ N - 1 - \frac{ex}{k}$
- $t = (s^2 - 4N)^{\frac{1}{2}}$

Un probabile fattore di N risulterà essere p1= $\frac{1}{2}(s + t)$, applicando l'algoritmo di Coppersmith

$$P1 \ + \ (2k \ + \ 1)N\text{^}(1/4)$$

determiniamo se p1 sia un fattore di N per k:K{-3, -2 ,…,2}

3.5 Fattorizzare φ(n) usando le approssimazioni Diophantine e lattice basis reduction

Questo attacco è particolarmente recente, infatti è stato pubblicato nell'agosto del 2019 [6].

Consiste nel fattorizzare simultaneamente più moduli di RSA, consideriamo quindi $n_s=p_sq_s$ ove s indica i differenti moduli; si creano quindi quattro equazioni differenti del tipo:

$e_s*d - k_s*\phi(n_s) = 1$

$e_s*d_s - k*\phi(n_s) = 1$

$e_s*d - k*\phi(n_s) = z_s$

$e_s*d_s - k*\phi(n_s) = z_s$

si dimostra che la seguente equazione sia una buona approssimazione per $\varphi(n_s)$

$$n - \left\lceil \left(\frac{\left(a^{\frac{i+1}{i}}\right) + \left(b^{\frac{i+1}{i}}\right)}{\left(2 * (a*b)^{\frac{i+1}{2i}}\right)} \right) + \left(\frac{a^{\frac{1}{j}} + b^{\frac{1}{j}}}{2\,(ab)^{\frac{1}{2j}}} \right) * n^{\frac{1}{2}} \right\rceil + 1$$

Con valori di d, d_s, k_s, z_s positivi interi non noti.

Si noti che questo algoritmo permette la fattorizzazione di molteplici moduli di RSA in tempi polinomiali, risultando ad oggi l'attacco più efficace all'algoritmo.

CAPITOLO III: Cycle attack

Il cycle attack si basa su un concetto semplice, partendo da un testo cifrato si esegue iterativamente il processo di cifratura con la stessa chiave fino ad riottenere il testo cifrato. Il risultato dell'iterazione precedente all'ottenimento del testo cifrato di partenza è uguale al testo in chiaro oggetto della prima cifratura. Questo processo risulta corretto poiché il processo di cifratura con una stessa chiave è univoco, un testo in chiaro genererà sempre lo stesso testo cifrato e quindi un testo cifrato sarà sempre dato dalla cifratura dello stesso testo in chiaro.

Tale attacco può essere espresso sotto forma di equazione come:

$$m^{e\ mod\ \varphi(\mathrm{n})} mod\ n = m^{e^{y}\ \mathrm{mod}\ \varphi(\mathrm{n})} mod\ n \text{ per y>1}$$

Dove *m* è il testo in chiaro, *n* è il modulo di RSA, *e* è la chiave di cifratura ed *y* è il numero di ripetizioni maggiori di 1 che porta a soddisfare l'eguaglianza.

Dato che l'esponente *e* è coprimo con ϕ(n) per definizione allora possiamo dividere l'esponente per *e* ottenendo cosi:

$$m^{\frac{e}{e}\ \mathrm{mod}\ \varphi(\mathrm{N})} mod\ n = m^{\frac{e^{y}}{\mathrm{e}}\mathrm{mod}\ \varphi(\mathrm{N})} mod\ n$$

$$m^{1\ \mathrm{mod}\ \varphi(\mathrm{N})} mod\ n = m^{\mathrm{e}^{\mathrm{y}-1}\mathrm{mod}\ \varphi(\mathrm{N})} mod\ n = m$$

Con questa equazione riesco a dimostrare che *y-1* è una chiave di decifratura valida.

La particolarità di questo attacco è il riuscire a spostare il problema dal dominio *n* a ϕ(n) in quanto si richiede di trovare un *y* tale per cui

$$e^{y-1} mod\ \varphi(\mathrm{N}) = 1\ mod\ \varphi(\mathrm{n})$$

Da cui per teorema di Eulero:

$$\mathrm{y} - 1 = \mathrm{k} * \varphi\big(\varphi(\mathrm{n})\big) \text{ per } k > 0$$

Per dimostrare che sia un attacco possibile per RSA dobbiamo dimostrare che esiste sempre un *y* tale da soddisfare l'equazione descritta sopra. Poiché vale il teorema di Eulero, è esso stesso a sancire l'esistenza di un *y* valido da qui si definisce l'esistenza di almeno un $y = \varphi(\varphi(n))$ per cui

$$e\hat{}\varphi(\varphi(n)) = 1\ mod\ \varphi(n)$$

1 Complessità

Per determinare se il Cycle attack sia una minaccia concreta al sistema crittografico bisogna analizzarne la complessità. Nella sua forma base questo attacco ha una complessità molto elevata in quanto bisognerebbe provare tutti i numeri y>1 fino ad y= $\varphi(\varphi(n))$, quindi possiamo affermare che la sua complessità sia:

$$O(\varphi(\varphi(n)))$$

Per ridurre la complessità dell'attacco possiamo stimare alcuni fattori da cui è composto ϕ(n). Possiamo determinare che y è un multiplo di 2 dato che

$$m^{e^{y \bmod \varphi(\varphi(n))} \bmod \varphi(n)} \bmod n$$

Sapendo che $n = p * q$ e che *p,q* sono primi per definizione, possiamo affermare che le seguenti congruenze risultino vere

$$p = 1 \bmod 2\ ,\ q = 1 \bmod 2$$

ricordando che $\varphi(n) = (p-1) * (q-1)$ dato dal fatto che *p* e *q* sono coprimi, possiamo dedurre che p-1 e q-1 sono pari ovvero:

$$\varphi(p) = p - 1 = 0 \bmod 2\ ,\ \varphi(q) = q - 1 = 0 \bmod 2$$

Utilizzando la proprietà associativa applicata alla funzione di Eulero $\varphi(pq) = \varphi(p) * \varphi(q) = \varphi(n)$ possiamo affermare che $\varphi(n)$ è multiplo di 4

$$\phi(n) = 0 \bmod 4$$

e quindi ϕ(ϕ(n)) risulterà essere multiplo di 2

$$\varphi(\varphi(n)) = 2 * \frac{2}{2} * j = 2 * j$$

dove *j* è il risultato della funzione $\frac{\varphi(\varphi(n))}{4}$.

Abbiamo quindi determinato che $\varphi(\varphi(n))$ è multiplo di 2, per cui bisognerebbe provare la metà dei casi originali poiché possiamo già escludere tutti gli *y* che non siano multipli di 2, dovendo così provare al massimo $\frac{\varphi(\varphi(n))}{2}$ casi differenti, ottenendo quindi una complessità massima pari a

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

1.1 Determinazione del valore massimo di φ(φ(n))

Ricordando che la funzione ϕ(n) determina la quantità di numeri coprimi con un numero possiamo affermare che il valore massimo di ϕ(ϕ(n)) si ha quando ϕ(n) è composto da un solo fattore primo

$\varphi(n) = x\hat{}y$ per ogni *x* primo e y>0

Poiché se ϕ(ϕ(n)) fosse composto da un solo numero primo bisognerebbe sottrarre da ϕ(n) soltanto i multipli di $x < \varphi(n)$ ottenendo quindi $\varphi\big(\varphi(n)\big) \leq \varphi(n) - \frac{\varphi(n)}{x}$

Se invece ϕ(n) fosse composto da piu numeri primi bisognerebbe sottrarre non solo i multipli di $x < \varphi(n)$ ma anche quelli di $z < \varphi(n)$ non divisibili per *x*. Ma poiché $gcd(x, z) = 1$ dato che *x,z* sono due numeri primi differenti esisterebbe almeno un valore di $z < \varphi(n)$ tale per cui $z \neq 0 \ mod \ x$ e quindi $\varphi\big(\varphi(n)\big) \leq \varphi(n) - \frac{\varphi(n)}{x} - 1$;

dato che $\varphi(n) - \frac{\varphi(n)}{x} - 1 < \varphi(n) - \frac{\varphi(n)}{x}$ possiamo affermare che otteniamo il valore massimo da $\varphi\big(\varphi(n)\big)$ quando $\varphi(n)$ è composto da un solo numero primo.

Se ciò è vero e sapendo che ϕ(n) è sempre composto da 2 allora possiamo affermare che:

$$\varphi(\varphi(n)) <= \varphi(n) - [\varphi(n)/2]$$

$$\varphi(\varphi(n)) <= \frac{\varphi(n)}{2}$$

1.2 Determinazione del valore minimo di φ(φ(n))

Con il ragionamento opposto a quello utilizzato nel capitolo precedente potremmo asserire che il valore minimo di ϕ(ϕ(n)) si ottiene quando ϕ(n) è composto da tutti numeri primi differenti ripetuti soltanto 1 volta. Questo ragionamento però non è sufficiente poiche non sappiamo quali sono i più fattori di ϕ(n) ma se calcolassimo un $k < \varphi(n)$ ove *k* è composto da tutti numeri primi in ordine crescente partendo da 2 si potremmo affermare che $\varphi(k) < \varphi(\varphi(n))$ poiché:

$$\varphi(k) = \varphi(k) * [(p1-1)/p1]\,[(p2-1)/p2]\,[(p3-1)/p3] \dots [(pn-1)/pn]$$

$$\varphi(\varphi(n)) = \varphi(\varphi(n))[(p1-1)/p1][(p2-1)/p2][(p3-1)/p3] \dots [(pn-1)/pn]$$

dato che $\varphi(pn) = [(pn-1)/pn] = 1 - [1/pn]$ e questa funzione è una funzione crescente che parte da 0 con pn=2 e tende a 1 al crescere di pn allora

$$\varphi(p1) < \varphi(p2) \text{ se } p1 < p2$$

Dato che k è composto dai minori numeri primi i più piccoli possibili non esiste un divisore di ϕ(n) che sia minore di un divisore di k e $\varphi(n) > K$ quindi:

$$\varphi\big(\varphi(n)\big) \geq \varphi(k)$$

Possiamo dunque definire che la funzione ϕ(ϕ(n)) sia compresa tra:

$$\frac{\varphi(n)}{2} \geq \varphi\big(\varphi(n)\big) \geq \varphi(k)$$

2 Ottimizzazione

Se riuscissimo a scoprire quanti sono i divisori di ϕ(n) allora potremmo abbassare il costo computazionale dell'attacco di $2\text{^}x$ ove x sono i fattori primi differenti che compongono ϕ(n) poiché

$$\varphi\big(\varphi(n)\big) = \varphi(n) * \left[\frac{p1-1}{p1}\right]\left[\frac{p2-1}{p2}\right]\left[\frac{p3-1}{p3}\right] \dots \left[\frac{pm-1}{pm}\right] =$$

$$= (p1-1)(p2-1)(p3-1) \dots (pm-1) * \left[\frac{\varphi(n)}{p1 * p2 * p3 \dots * pm}\right]$$

sapendo che *p1,p2,p3…pn* sono numeri primi, allora *(p1-1) (p2-1) (p3-1) … (pm-1)* saranno numeri pari e quindi divisibili per 2 per ogni pm>2, dato che al più può esservi un solo pm=2 allora ϕ(n) sarà sicuramente multiplo di 2^{m-1}

$$\varphi\big(\varphi(n)\big) = 0 \bmod 2^{m-1},$$

2.1 stime probabilistiche

Per determinare quanti sono i divisori di ϕ(n) ho ideato delle funzioni matematiche, basate sul teorema dei numeri primi, volte a determinare con che probabilità un numero primo *p* meno 1 sia composto da *x* differenti fattori primi.

Premesse:

- p è primo
- $2^{k-1}+1 < p < 2^{k+1}-1$
- Teorema dei numeri primi
  - La quantità di numeri primi in un intervallo k è definita della seguente formula

    k /ln(k) < quantità di numeri primi minori di k <1.25506* k /ln(k)

  - $f(x) = 1.25506 * \frac{2^x}{\ln(x)}$
  - $g(x) = \frac{2^x}{\ln(x)}$
  - $h(x) = f(x) - g(x-1)$ ovvero la quantità massima di numeri primi tra 2^{x-1} e 2^x
- Probabilità di un evento x = P(x) = eventi favorevoli/eventi possibili

Cerco di stabilire una funzione che determini quante sono le possibili combinazioni di *x* numeri primi che rientrano nello stesso intervallo di *p*. Per determinare questa funzione mi baso su sul fatto che un numero di un ordine 2^t può essere scomposto in due numeri di ordine $2^{t-r} * 2^r$ con t>r e r>0 e utilizzando questa scomposizione calcolo quanti sono i numeri primi minori del primo intervallo $1.25506 * \frac{2^{t-r}}{ln(2^{t-r})} - \frac{2^{t-r-1}}{ln(2^{t-r-1})}$

per quelli che sono i numeri primi nel secondo intervallo

$$1.25506 * \frac{2^r}{ln(2^r)} - \frac{2^{r-1}}{ln(2^{r-1})}$$

I cui valori vengono sommati per tutti i possibili valori di r e t, ottenendo così:

$$\sum_{r=1}^{r=t}[(1.25506 * \frac{2^{t-r}}{ln(2^{t-r})} - \frac{2^{t-r-1}}{ln(2^{t-r-1})}) * (1.25506 * \frac{2^r}{ln(2^r)} - \frac{2^{r-1}}{ln(2^{r-1})})]$$

Chiamiamo quindi questa formula $f_{s2}(t)$ ed è in grado di stimare quanti sono i numeri composti da 2 numeri primi differenti nell'intervallo $[2^{t-1}+1\,,2^{t+1}-1]$

Utilizzando la funzione precedente possiamo impostare una funzione che è in grado di stimare quanti sono i numeri composti da tre numeri primi differenti nell'intervallo $[2^{t-1}+1\,,2^{t+1}-1]$ come:

$$fs3(t)=\sum_{r=t-2}^{r=t}(1.25506*\frac{2^r}{ln(2^r)}-\frac{2^{r-1}}{ln(2^{r-1})})*fs2(r-1)$$

Possiamo quindi definire una funzione che stimi quanti sono i numeri composti da m numeri primi differenti nell'intervallo $[2^{t-1}+1\,,2^{t+1}-1]$ come:

$$fsm(t)=\sum_{r=t-(m-1)}^{r=t}\left(1.25506*\frac{2^r}{ln(2^r)}-\frac{2^{r-1}}{ln(2^{r-1})}\right)*fs(m-1)(r-1)$$

Quindi possiamo stabilire la probabilità che un numero primo in un certo intervallo sia composto da m differenti fattori primi come casi favorevoli su casi totali:

$$P(p)=\frac{fm(t)}{\frac{t}{2}}$$

Dove i casi totali posso essere al più *t/2 -1* poiché sappiamo che *p* è necessariamente divisore di 2.

Questa funzione determina un limite superiore della probabilità P(p) poiché viene utilizzato il teorema dei numeri primi superiore.

Per determinare il limite inferiore della probabilità P(p) può essere utilizzata una forma analoga inserendo il coefficiente moltiplicativo del teorema dei numeri primi come coefficiente moltiplicativo nell'elemento sottrattivo della formula, definendo la funzione come:

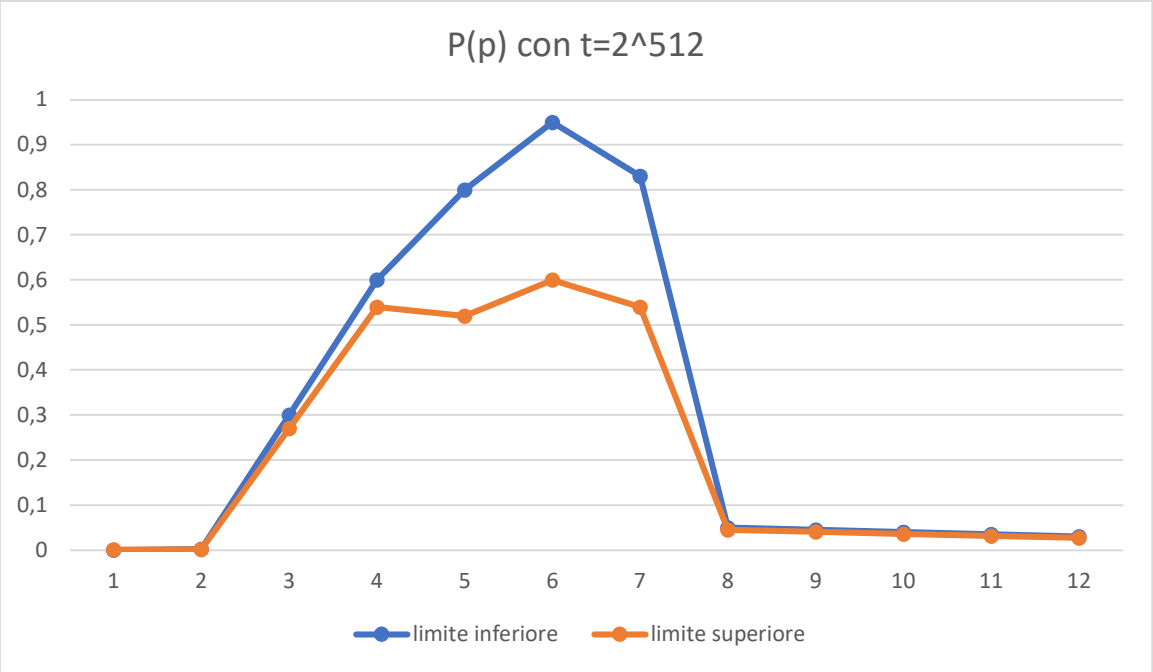
$$fi2(t)=\sum_{r=1}^{r=t}(\frac{2^{t-r}}{ln(2^{t-r})}-1.25506*\frac{2^{t-r-1}}{ln(2^{t-r-1})})*(\frac{2^r}{ln(2^r)}-1.25506*\frac{2^{r-1}}{ln(2^{r-1})})$$

Da cui poi:

$$fim(t) = \sum_{r=t-(m-1)}^{r=t} \left(\frac{2^r}{ln(2^r)} - 1.25506 * \frac{2^{r-1}}{ln(2^{r-1})} \right) * fi(m-1)(r-1)$$

Ho così determinato due funzioni che determinano il limite superiore ed inferiore della probabilità P(p) che sarà:

$$fim(t) < P(p) < fsm(t)$$



Date queste formule si riesce a ricostruire un addensamento il che se utilizzato per implementare il Cycle attack determina un diminuzione della complessità dei calcoli poiché riusciamo a stimare da quanti fattori primi differenti è composto ϕ(n).

Per esempio, se si utilizza RSA con modulo a 128 bit la probabilità ricavata da queste stime fa presumere che probabilmente ϕ(p) sia composto da almeno 4 differenti valori e quindi la complessità dell'algoritmo risulta essere:

$$O\left(\frac{\varphi(\varphi(n))}{2^4}\right)$$

Al contempo possiamo anche affermare che

$$\varphi(\varphi(n)) \leq \varphi(n) - \frac{\varphi(n)}{2} - \frac{\left(\left(\frac{\varphi(n)}{3}\right) - 1\right)\varphi(n)}{\frac{\varphi(n)}{3}} - \frac{\left(\left(\frac{\varphi(n)}{5}\right) - 1\right)\varphi(n)}{\frac{\varphi(n)}{5}} - \frac{\left(\left(\frac{\varphi(n)}{7}\right) - 1\right)\varphi(n)}{\frac{\varphi(n)}{7}}$$

3.2.2 Addensamento verso il basso

Nel corso del mio studio sui numeri primi ho anche scoperto che dato un numero primo *p* è molto probabile che ϕ(p) sia composto prevalentemente da numeri primi piccoli.

Dato che per definizione p è coprimo con tutti i numeri C minori di p, ovvero

$$p \neq 0 \; mod \; C1$$

Da cui p può essere congruo a *C1* numeri tranne uno ovvero lo '0'

Applicando lo stesso ragionamento per p-1

$$p - 1 \neq C1 - 1 \; mod \; C1$$

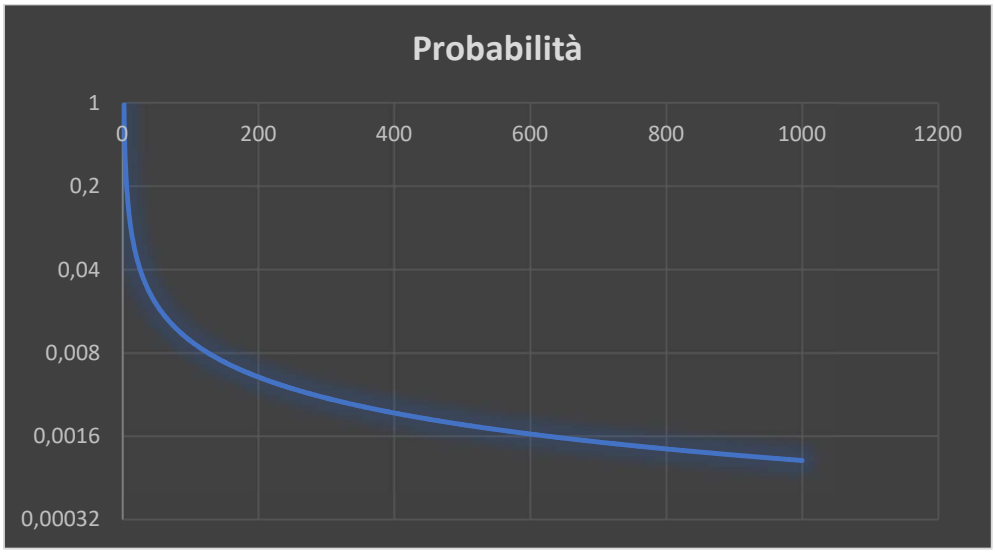
Per cui *p-1* può essere congruo a *C1* numeri tranne 1 ovvero *C1-1*

Quindi la probabilità che *p-1* sia divisibile per *C1* è

$$P = \frac{1}{C1-1}$$

Quindi al crescere di C1 la probabilità che un *φ(p)* sia multiplo di *C1* decresce

$$\lim_{p\to\infty} \frac{1}{P-1} = 0$$



Deduco quindi che ϕ(P) tende a esser composto da numeri primi piccoli

3 Recursive Cycle attack

Una variante del Cycle attack prevede di spostare il problema del logaritmo discreto in un dominio sempre più piccolo fino a far collassare il problema su un dominio di un solo possibile valore.

$$m^{e^{d1^{d2^{\cdots^{dl\ mod\ \varphi(2)}\bmod \varphi\left(\varphi(\varphi(n))\right)}_{mod\ \varphi(\varphi(n))}\bmod \varphi(n)}mod\ n}$$

Per riuscire a fare ciò è essenziale che i vari *d* che vengono scelti siano coprimi con i vari moduli non noti, si ottiene quindi che la probabilità di scegliere i diversi esponenti *d* casualmente e che questi siano coprimi con i rispettivi *φ(n)* è pari alla somma delle probabilità nei singoli casi, portando quindi la probabilità generale di un fallimento molto vicino ad 1.

Per azzerare queste probabilità di fallimento si potrebbe scegliere un numero primo *p* maggiore di *φ(n)*, il quale in modulo *φ(n)* genererebbe un numero *j* minore di *φ(n)*,e coprimo con *φ(n).*

$$gcd(j, \varphi(n)) = 1$$

Da cui o $P\ mod\ \varphi(n)\ = 1$ o *P* è coprimo con *φ(n)* per cui in entrambi i casi si potrebbe usare P come potenza per la ricorsione che se eseguita il numero *t* volte dove *t*=all'ordine di grandezza di *n* si riuscirebbe a decifrare RSA. Poiché se è uguale a 1 allora abbiamo trovato due potenze modulari che sono uguali sennò è come eseguire ricorsivamente RSA finché ϕ(ϕ(ϕ(…ϕ(2)))) =1.

Questo metodo è puramente teorico poiché si dovrebbero eseguire troppe potenze di numeri con troppe cifre circa !t

Esempio:	101 mod 2= 1	gcd(1,2)=1
	101 mod 3= 2	gcd(2,3)=1
	101 mod 4= 1	gcd(1,4)=1
	101 mod 5= 1	gcd(1,5)=1
	101 mod 6= 5	gcd(5,6)=1
	101 mod 7= 3	gcd(3,7)=1
	101 mod 8= 5	gcd(5,8)=1
	101 mod 9= 2	gcd(2,9)=1
	101 mod 10=1	gcd(1,10)=1

CAPITOLO IV: Implementazione

Per dimostrare il funzionamento del Cycle attack in maniera empirica ne ho creato un'implementazione in c#, che pur essendo un linguaggio "lento" rispetto al linguaggio c comporta dei notevoli vantaggi. Infatti dato che c# ha delle classi per la gestioni dei numeri molto grandi, complete e ben ottimizzate può arrivare ad avere prestazioni superiori al c implementato in maniera naif. Le classi che ho usato sono principalmente due:

- BigInteger: è una classe che permette di gestire numeri interi grandi di dimensioni illimitate, purché il dispositivo abbia sufficiente memoria nella quale inserire i dati. Ho utilizzato questa classe per gestire i processi crittografici. I metodi principali di questa classe che ho utilizzato sono:
    - ModPow: ovvero il metodo che esegue potenze modulari, sfruttando anche i concetti di ottimizzazione accennati a pagina nel paragrafo 3.2.2 del capitolo I.
    - Pow: metodo che consente di eseguire la potenza di un numero di tipo BigInteger.
    - Log: restituisce il valore del logaritmo di un numero BigInteger in una qualsia base intera.
    - Parse: converte una stringa in un numero BigInteger.
- BigFloat: è una classe che permette di gestire numeri decimali grandi di dimensioni illimitate, purché il dispositivo abbia sufficiente memoria nella quale inserire i dati. Ho utilizzato questa classe per gestire le stime probabilistiche, dato che i valori in gioco sono molto grandi e quindi non posso usare i normali tipi e che la probabilità è definita tra 0 e 1 è quindi impossibile utilizzare la classe BigInteger. I metodi principali di questa classe che ho utilizzato sono:
    - Pow: metodo che consente di eseguire la potenza di un numero di tipo BigFLoat.
    - Log: restituisce il valore del logaritmo di un numero BigFLoat in una qualsia base di tipo BigFLoat.
    - Ceiling: approssima un numero BigFloat nel numero BigFLoat intero più vicino, con un'approssimazione per eccesso
    - Parse: converte una stringa in un numero BigFloat.

Nell'appendice 2 è allegato il codice sorgente dove si può notare che il programma è strutturato in tre classi principali:

- **Program**: dove è contenuto il Main del programma; il Main avrà il compito di chiamare il processo di decifratura.
- **funzioniMatematiche**: è la classe che avrà il compito di implementare tutte le funzioni matematiche e le stime probabilistiche illustrate in questo studio.
- **cycleAttack**: contiene i metodi che permettono di eseguire l'attacca con le dovute implementazioni

la classe funzioniMatematiche contiene tutte le implementazioni delle varie formule utilizza in questo studio:

- TeoremaNumeriPrimiSuperiore: implementazione della funzione matematica

$$1.25506 * \frac{2^r}{ln(2^r)}$$

- TeoremaNumeriPrimiInferiore: implementazione della funzione matematica

$$\frac{2^r}{ln(2^r)}$$

- stimeProbabilisticheSuperiori: implemntazione della funzione matematica

$$fim(t) = \sum_{r=t-(m-1)}^{r=t} \left(1.25506 * \frac{2^r}{ln(2^r)} - \frac{2^{r-1}}{ln(2^{r-1})}\right) * f(m-1)(r-1)$$

Viene eseguita in maniera ricorsiva:

$$\sum_{r=t-(m-1)}^{r=t} (\text{TeoremaNumeriPrimiSuperiore} - \text{TeoremaNumeriPrimiInferiore}) * stimeProbabilisticheSuperiori(m-1)(r-1)$$

utilizzando come caso base, ossia quando m=2, la funzione:

$$\left(1.25506 * \frac{2^r}{ln(2^r)} - \frac{2^{r-1}}{ln(2^{r-1})}\right)$$

- stimeProbabilisticheInferiori: implemntazione della funzione matematica

$$fim(t) = \sum_{r=t-(m-1)}^{r=t} \left(\frac{2^r}{ln(2^r)} - 1.25506 * \frac{2^{r-1}}{ln(2^{r-1})} \right) * (m - 1)(r-1)$$

Viene eseguita in maniera ricorsiva, utilizzando come caso base come

$$\sum_{r=t-(m-1)}^{r=t} (\text{TeoremaNumeriPrimiInferiore} - \text{TeoremaNumeriPrimiSuperiore}) * stimeProbabilisticheInferiori(m-1)(r-1)$$

utilizzando come caso base, ossia quando m=2, la funzione:

$$\left(\frac{2^r}{ln(2^r)} - 1.25506 * \frac{2^{r-1}}{ln(2^{r-1})} \right)$$

- DeterminareK: viene utilizzata per stimare il valore minimo di ϕ(ϕ(n)), sapendo che ϕ(n) è dello stesso ordine di n, si scorre un file che contiene una lista di numeri primi minori di 38792 così da determinare un $k < \varphi(n)$ composto da prodotto dei numeri primi dal più piccolo al più grande contenuto nel file e nel contempo si genera un ϕ(k) tramite prodotti tra i numeri letti dal file a cui prima si sottrae 1.

cycleAttack

- Decript: è il metodo che avvia la fase di decifratura, richiede all'utente di inserire la chiave di RSA (n,exponent) e un testo cifrato in modo da decriptarlo, salva i seguenti parametri e chiama il metodo Decifratura
- Decifratura: chiamando il metodo analisiStime così da determinare un divisore di ϕ(n), definiamo così una variabile chiamata Coefficiente che sarà uguale ad $exponent^{analisiStime}$. Utilizzando la funzione DeterminareK definiamo una variabile *e* come $exponent^{DeterminareK}$, questo sarà il minimo valore della chiave di decifrtura. Eseguiamo ciclicamente il prodotto tra *e* $exponent^{DeterminareK}$ una volta salvato in *e* utilizziamo quest'ultima come esponente delle potenze modulare. Il processo si ripete fino a che il risultato ottenuto dalla potenza modulare sia uguale al testo cifrato di partenza.
- analisiStime: è il metodo che analizza le stime probabilistiche al fine di determinare qual è il risultato ottimale da utilizzare.

Conclusioni

Per concludere vorrei quindi analizzare le differenze computazionali tra il Cycle attack nella sua versione base e il Cycle attack implementato usando le stime probabilistiche con gli algoritmi di fattorizzazione.

Il cycle attack implementato usando le stime probabilistiche dimostra di riuscire a ridurre i tempi computazionali di circa 10 volte rispetto all'attacco base su RSA con chiave a 1024 bit, ciò nonostante risulta essere molto più oneroso degli algoritmi di fattorizzazione.

Bit della chiave	cycle attack	cycle attack con stime probabilistiche	Quadratic Seive	General Number Field Sieve
32	2^30	2^27	2^12	2^17
64	2^62	2^58	2^19	2^24
512	2^510	2^505	2^66	2^64
1024	2^1022	2^1012	2^99	2^87
2048	2^2046	2^2039	2^147	2^117

Tabella riassuntiva dei vari costi computazionali massimi in relazione alla chiave di RSA

Tuttavia, dalle prove sperimentali sono emersi dei valori per cui la computazione del Cycle attack in ogni sua forma risulta essere notevolmente più veloce di tutti gli altri attacchi ad RSA, ma purtroppo non sono riuscito a trovare una correlazione di questi valori per stimarne una formula.

Esempio.

Data la chiavepubblica(n,e) e il testo cifrato c con

n=2031649783

e= 65537

c= 191935117

il cycle attack trova un y che soddisfa la seguente equazione in circa 5 minuti: $n^{e^y} = n^e mod\ n$ con un y molto minore di ϕ(ϕ(n))

ϕ(ϕ(n))= 641433600 ≈ 2^30 y=104400 ≈ 2^17

tutto ciò se implementato con le stime probabilistiche impiega circa 1 minuto a computare la stessa equazione e eseguirebbero solo 2^13 differenti dentativi.

Da questo studio si può cogliere un'altra particolarità, osservando il comportamento del Recursive Cycle attack si può affermare che esiste una super chiave di decifratura di RSA anche se ad oggi impossibile da calcolare.

Una continuazione di questo studio consisterà nell'analizzare le anomalie sopra citate e studiare la probabilità che due numeri aventi entrambi lo stesso numero di fattori primi abbia qualche fattore in comune. Da questi due sviluppi si potrebbe trovare il più veloce metodo di attacco ad RSA.

Appendice 1: Codice Sorgente

```
using System;
using System.Globalization;
using System.Numerics;
using Extreme.DataAnalysis;
using Extreme.Mathematics;
using BigInteger = System.Numerics.BigInteger;

namespace tesi
{
    class Program
    {
        static void Main(string[] args)
        {
            CycleAttac.decript();
        }
    }
    public class funzioniMatematiche
    {
      // metodo che implementa la funzione 2^x * 1.25506 /ln(x)
        public static BigFloat TeoremaNumeriPrimiSuperiore(BigFloat t)
        {
            // utilizzo 2^t per ricostruire in range da analizzare
            BigFloat y = BigFloat.Pow(2, t);

//per migliorare la precisione posso utilizzare una base di numeri
primi
            if (t < 5)
            {
                int[] prime_list = { 1, 2, 3, 5, 7, 11, 13, 17, 19,
23, 29, 31 };
                int k = 0;
                while (y >= prime_list[k])
                {
                    k++;
                }
                return k;
            }
            //numeratore della frazione
            BigFloat num = 1.25506 * y;

            //denominatore della frazione
            BigFloat den = BigFloat.Log(y);

            //risultato
            BigFloat res = num / den;
            return res;
        }
```

```
        // metodo che implementa la funzione 2^x /ln(x)
        public static BigFloat TeoremaNumeriPrimiInferiore(BigFloat t)
        {
            // utilizzo 2^t per ricostruire in range da analizzare
            BigFloat y = BigFloat.Pow(2, t);

//per migliorare la precisione posso utilizzare una base di numeri
primi
            if (t < 5)
            {
                int[] prime_list = { 1, 2, 3, 5, 7, 11, 13, 17, 19,
23, 29, 31 };
                int k = 0;
                while (y >= prime_list[k])
                {
                    k++;
                }
                return k;
            }
            //numeratore della frazione
            BigFloat num = 1.25506 * y;

            //denominatore della frazione
            BigFloat den = BigFloat.Log(y);

            //risultato
            BigFloat res = num / den;
            return res;
        }
// metodo che implementa la funzione ricorsiva che stima per eccesso
quanti sono i numeri composti da x fattori nell'intervallo 2^(t-1),
2^t
        public static BigFloat stimeProbabilisticheSuperiori(int
x,BigFloat t)
        {
            //caso base x=2
            if(x==2)
            {
                BigFloat r;
                r = 1;
                t = t - 1;
                BigFloat result;
                result = 0;
// vado a definire il ciclo che eseguirà la sommatoria per tutti i
possibili valori di r
                while (r < t)
                {
// implementazione del caso base dove il numero di fattori deve essere
uguale a 2: 2^t * 1.25506 /ln(2^t)- 2^(t-1) /ln(2^(t-1))
                result = result + (TeoremaNumeriPrimiSuperiore(t-r) -
TeoremaNumeriPrimiInferiore(t-r - 1)) *
(TeoremaNumeriPrimiSuperiore(r)-TeoremaNumeriPrimiInferiore(r - 1));
                    r = r + 1;
                }
                return result;
```

```
            }
// se non siamo nel caso base ovvero t>2
            else
            {
                BigFloat r;
                r = (x-1);
                t = t -(x-1);
                BigFloat result;
                result = 0;
// vado a definire il ciclo che eseguirà la sommatoria per tutti i
possibili valori di r
                while (r < t)
                {
// implementazione ricorsiva dove il numero di fattori deve essere
uguale a n
                    result = result + (TeoremaNumeriPrimiSuperiore(t-
r) - TeoremaNumeriPrimiInferiore(t-r - 1)) *
stimeProbabilisticheSuperiori(x-1,r);
                    r = r + 1;
                }
                return result;
            }
        }

// metodo che implementa la funzione ricorsiva che stima per difetto
quanti sono i numeri composti da x fattori nell'intervallo 2^(t-1),
2^t
        public static BigFloat stimeProbabilisticheInferiori(int x,
BigFloat t)
        {
            //caso base t=2
            if (x == 2)
            {
                BigFloat r;
                r = 1;
                t = t - 1;
                BigFloat result;
                result = 0;
// vado a definire il ciclo che eseguirà la sommatoria per tutti i
possibili valori di r
                while (r < t)
                {
// implementazione del caso base dove il numero di fattori deve essere
uguale a 2: 2^t /ln(2^t)- 1.25506 * 2^(t-1) /ln(2^(t-1))
                    result = result + (TeoremaNumeriPrimiInferiore(t-
r) - TeoremaNumeriPrimiSuperiore(t-r - 1)) *
(TeoremaNumeriPrimiInferiore(r) - TeoremaNumeriPrimiSuperiore(r - 1));
                    r = r + 1;
                }
                return result;
            }
            else
            {
                BigFloat r;
                r = (x-1);
```

```
                t = t -(x-1);
                BigFloat result;
                result = 0;
// vado a definire il ciclo che eseguirà la sommatoria per tutti i
possibili valori di r
                while (r < t)
                {
// implementazione ricorsiva dove il numero di fattori deve essere
uguale a n
                    result = result + (TeoremaNumeriPrimiInferiore(t-
r) - TeoremaNumeriPrimiInferiore(t-r - 1)) *
stimeProbabilisticheInferiori(x - 1, r);
                    r = r + 1;
                }
                return result;
            }
        }


// implementazione della funzione volta a stabilire un numero ϕ(K)
minore di ϕ(ϕ(n))
        public static BigInteger DeterminareK(BigInteger mod)
        {
            //determino l'ordine di grandezza del modulo
            mod = BigInteger.Pow(2, (int)BigInteger.Log(mod, 2));
            System.IO.StreamReader file = openfile();
            BigInteger k, fiK, appoggio;
            k = 2;
            fiK = 1;
            appoggio = 2;
//eseguo il ciclo che scansionando il file nel quale è contenuta la
lista di numeri primi e determinerà il massimo valore di k minore di
ϕ(n) e composto da soli numeri primi crescenti
            while (k < mod)
            {
//calcolo ϕ(K) mentre calcolo K in modo da risparmiare tempo
                fiK = fiK * (appoggio - 1);
                appoggio = BigInteger.Parse(readline(file));
                k = k * appoggio;
            }
            return fiK;
        }

        //utilizzo questo metodo per tenere più pulito il codice
        private static System.IO.StreamReader openfile()
        {
            var _assembly = System.Reflection.Assembly
               .GetExecutingAssembly().GetName().CodeBase;
            var _path = System.IO.Path.GetDirectoryName(_assembly);
            System.IO.StreamReader file =
            new System.IO.StreamReader(@"..\..\..\NumeriPrimi.txt");
            return file;
        }
```

```
        //questo metodo legge i vari numeri primi presenti nel file
        private static string readline(System.IO.StreamReader file)
        {
            string j;
            j = file.ReadLine();
            if (j != null)
                return j;
            else
            {
                file.Close();
                return "";
            }
        }

    }

    public class CycleAttac
    {
// è un metodo che avrà il compito di acquisire i valori necessari
alla decifratura e fornirà i risultati della decifratura
        public static void decript()
        {
            BigInteger cyfer, modulo;
            BigInteger exponet, decript;
            Console.WriteLine("\n\ninserire il testo ciffrato");
            cyfer = BigInteger.Parse(Console.ReadLine());
            Console.WriteLine("inserire la chiave di cifratura");
            exponet = BigInteger.Parse(Console.ReadLine());
            Console.WriteLine("inserire il modulo");
            modulo = BigInteger.Parse(Console.ReadLine());
            int primiDifferenti;
            primiDifferenti= analisiStime(modulo);
            decript=decifratura(cyfer, exponet, modulo,
primiDifferenti);
            Console.WriteLine("il testo in chiaro è:\n" +
decript.ToString());
        }

// è il metodo dove avviene il processo di decifratura
        public static BigInteger decifratura(BigInteger cyfer,
BigInteger exponet, BigInteger modulo, int primiDifferenti)
        {
            BigInteger e,cyfer1,appoggio;
            cyfer1 = cyfer;
            int r = 2 * (primiDifferenti + 1);
            BigInteger Coefficente = BigInteger.Pow(exponet, r);
            BigInteger fiK = funzioniMatematiche.DeterminareK(modulo);
//dato che ϕ(K) potrebbe non essere congruo al minimo valore di
ϕ(ϕ(n)) che ho calcolato determino un valore appena più piccolo di
ϕ(K) che lo sia
            fiK = fiK-(fiK % r);
            int ripetizioni = 0;
            e = Coefficente;
            e = BigInteger.Pow(e, fiK);
```

```
// è il processo per il quale si cercano iterativamente di determinare
quale sia il numero che ha generato il testo cifrato
            do
            {
                ripetizioni++;
//valore candidato ad essere il testo decifrato
                appoggio = cyfer1;
                cyfer1 = BigInteger.ModPow(cyfer, e, modulo) ;
//sfruttando le proprietà delle potenze evito di eseguire tante
potenze con la stessa base e * e^b = e^b+1
                e = e * Coefficente;
            }
//il ciclo termina quando trovo l'uguaglianza
            while (cyfer != cyfer1);
            appoggio = BigInteger.ModPow(appoggio,
BigInteger.Pow(exponet, r-1), modulo);
            return appoggio;
        }


//è il metodo che richiama le stime probabilistiche e le valuta
secondo un grado di tolleranza che vogliamo
        public static int analisiStime(BigInteger modulo)
        {
            BigFloat t;
            t =
BigFloat.Ceiling(BigFloat.Log(BigFloat.Parse(modulo.ToString()), 2));
            t = t / 2;
            int x;
            x = 2;
            BigFloat precisione;
//grado di tolleranza scelto, più questo è basso e meno migliora la
complessità dell'algoritmo rispetto al Cycle attack base. Più è alto
questo valore più è probabile che non sia affidabile la stima
            precisione = 0.3;
            while
(funzioniMatematiche.stimeProbabilisticheSuperiori(x,t) /
BigFloat.Pow(2, t - 1) < precisione)
            {
                x++;
            }
            x--;
            return x;
        }

    }
}
```

Appendice 2: Nozioni matematiche

Gruppo

È una struttura algebrica matematica definita per una certa operazione * che soddisfa le seguenti 4 proprietà [10]:

- Esistenza dell'elemento neutro
- Esistenza dell'elemento inverso
- Chiuso rispetto all'operazione
- Associatività

Esistenza dell'elemento neutro

Dato un insieme Z_n e l'operazione * definita in Z_n, si dice che l'elemento $u \in Z_n$ è elemento neutro dell'operazione * se per ogni $a \in Z_n$ risulta:

$$a*u = a.$$

Esistenza dell'elemento inverso

L'operazione * in Z_n gode della proprietà di invertibilità se per ogni coppia a, $a' \in Z_n$ risulta:

$$a*a' = a'*a=u$$

dove u è l'elemento neutro dell'insieme Z_n

Chiuso rispetto all'operazione

L'operazione * in Z_n gode della proprietà di chiusura se per ogni coppia $a,b \in Z_n$ risulta:

$$a*b \in Z_n$$

Associatività

L'operazione * in Z_n gode della proprietà associativa se per ogni terna a, b, $c \in Z_n$ risulta:

$$a*(b*c) = (a*b)*c$$

Loop

È una struttura matematica simile ad un gruppo e definita come quasi-group che necessita di tutte le proprietà di un gruppo tranne della proprietà associativa

La potenza discreta non può essere considerata un gruppo ma è a tutti gli effetti un loop:

- Esistenza dell'elemento neutro:

  a*u = a → $a^1 mod(N)=a$

- Esistenza dell'elemento inverso
  a*a' =u → $a^{phi(n)} mod(n)=1$ se gcd(a,n)=1
- Chiusura
  a*b ∈ Z_n → $a^b mod(n) \in Z_n$ per definizione dell'operazione modulo
- Associatività

  a*(b*c)=(a*b)*c

  $a^{b^c mod\ N} mod(N) = (a^b mod\ N)^c mod\ N$

  $a^{b^c mod\ N} mod(N) = a^{b*c}\ mod\ N$

  $b\hat{}c\ mod\ N = b * c$ ←questa proprietà non è soddisfatta

Logaritmo discreto

Sia G un gruppo ciclico finito di n elementi. Definiamo un numero b come generatore di G, allora ogni elemento g di G può essere scritto nella forma $g=b^k$ per ogni k intero positivo. Due numeri h,k si dicono congrui in modulo se vale la seguente uguaglianza

$b^k = g = b^h$

Si definisce la funzione logaritmo discreto in Z_n con la seguente equazione:

$log_b : G \rightarrow Z_n$

Toziente phi di Eulero

È una funzione definita per ogni numero $n \in N_+$ e rappresenta il numero degli interi positivi coprimi con n nell'intervallo 1,n.

La funzione Phi(n) si esprime con ϕ(n)

Gode della proprietà associativa:

$$\varphi(n * n1) = gcd(n, n1) * \varphi\left(\frac{n}{gcd(n, n1)}\right) * \varphi(n1) =$$

$$= gcd(n, n1) * \varphi(n1/gcd(n, n1)) * \varphi(n)$$

Da cui se n è coprimo con n1 vale la seguente eguivalenza:

$$\varphi(n * n1) = \varphi(n) * \varphi(n1) \quad \text{se gcd(n,n1)=1}$$

Calcolo della funzione

per ogni numero n>1 e se n è un numero primo allora Toziente phi di Eulero sarà:

$$\varphi(n) = n - 1$$

Per convenzione ϕ(1)=1.

Se N non è primo allora la funzione ϕ(n) viene calcolata come:

$$\varphi(n) = n \cdot \left[\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_r}\right)\right] = n \prod_{p|n}\left(1 - \frac{1}{p}\right)$$

$$\varphi(n) = gcd(p1 \dots p2) * \varphi\left(\frac{n}{gcd(n,n1)}\right) * \varphi(n1) =$$

Per ogni p1,p2,…,pn che fattorizzano n.

Teorema di Eulero

Per ogni a $\in Z_n^*$ e gcd(a,n)=1

$a^{\phi(n)} = 1 \bmod n$

Teorema dei numeri primi

Numeri primi minori di x = π(x) [6]

$$\frac{x}{\ln(x)} < \pi(x) < 1.25506\frac{x}{\ln(x)}$$

Bibliografia

[1] Guri, M., Bykhovsky, D., & Elovici, Y. (2019a). BRIGHTNESS: Leaking Sensitive Data from Air-Gapped Workstations via Screen Brightness. In proceedings of 2019 12th CMI Conference on Cybersecurity and Privacy. Disponibile da https://arxiv.org/abs/2002.01078

[2] RFC8017, https://tools.ietf.org/html/rfc8017

[3] Pomerance, C. (1984). The Quadratic Sieve Factoring Algorithm. In proceedings of Eurocrypt 1984. Disponibile da http://websites.math.leidenuniv.nl/algebra/sieving.pdf

[4] Smart, N. P. (2003). Cryptography: An Introduction. McGraw-Hill.

[5] Blömer J., May A. (2004) A Generalized Wiener Attack on RSA. In: Bao F., Deng R., Zhou J. (eds) Public Key Cryptography – PKC 2004. PKC 2004. Lecture Notes in Computer Science, vol 2947. Springer, Berlin, Heidelberg, 2004.https://www.researchgate.net/publication/221010611_A_generalized_Wiener_attack_on_RSA

[6] Abubakar, S. I. (2019). Successful Cryptanalytic Attacks Upon RSA. MALAYSIAN JOURNAL OF MATHEMATICAL SCIENCES. Disponibile da https://www.researchgate.net/publication/337294136_Successful_Cryptanalytic_Attacks_Upon_RSA

[7] Susilo W., Tonien J., Yang G. (2019) The Wiener Attack on RSA Revisited: A Quest for the Exact Bound. In: Jang-Jaccard J., Guo F. (eds) Information Security and Privacy. ACISP 2019. Lecture Notes in Computer Science, vol 11547. Springer, Cham.

[8] STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of ComputingApril 1990 Pages 564–572 https://doi.org/10.1145/100216.100295.

Testi e Pubblicazioni utilizzate nelle mie ricerche

[9] Gysin M., Seberry J. (1999) Generalised Cycling Attacks on RSA and Strong RSA Primes. In: Pieprzyk J., Safavi-Naini R., Seberry J. (eds) Information Security and Privacy. ACISP 1999. Lecture Notes in Computer Science, vol 1587. Springer, Berlin, Heidelberg.

[10]Gysin, Marc & Seberry, Jennifer. (1999). Generalised cycling attacks on RSA and strong RSA primes. 1587. 149-163. 10.1007/3-540-48970-3_13.

[11] Cycle attack on RSA. (2012, 4 Gennaio). Disponibile da https://crypto.stackexchange.com/questions/1572/cycle-attack-on-rsa

[12] How does the cyclic attack on RSA work? (2012, 26 Giugno). Disponibile da https://crypto.stackexchange.com/questions/3055/how-does-the-cyclic-attack-on-rsa-work

[13] Stelvio Cimato, Dispense del corso di Crittografia (2018-2019)

[14] Laura Citrini, Dispense del corso di “Matematica del discreto”