Capitolo 1

Introduzione

Da quando la crittografia ha cessato di essere un'arte per assurgere allo stato di scienza[DH76], sono nati almeno due modi profondamente diversi fra loro di vedere la crittografia.
In uno di questi, il modello *formale*, le operazioni crittografiche sono rappresentate da espressioni simboliche, formali. Nell'altro, il modello *computazionale* le operazioni crittografiche sono viste come funzioni su stringhe di bit; le operazioni hanno una semantica probabilistica. Il primo modello è stato teorizzato in []... Il secondo modello trova le basi in lavori di altrettanto illustri studiosi []

In [AR07] gli autori cercano per la primavolta di porre le basi per iniziare a collegare questi due modelli.

Capitolo 2

Il Modello Computazionale

2.1 L'Avversario

Il tipico avversario con cui si ha a che fare quando si studiano cifrari o protocolli crittografici nel modello computazionale, è un avversario con risorse di calcolo *limitate*. Limitate nel senso che si sceglie di porre un limite alla potenza di calcolo dell'avversario. Questo significa che non avremo a che fare con un avversario che ha una potenza computazionale infinita o un tempo illimitato a disposizione.
Sebbene siano stati ideati cifrari sicuri anche rispetto ad avversari non limitati[1], questi hanno alcuni difetti come per esempio il fatto che la chiave debba essere lunga quanto il messaggio o che sia utilizzabile una sola volta. Per rappresentare in modo formale un avversario con risorse di calcolo limitate, lo si può pensare come un algoritmo appartenente ad una particolare classe di complessità computazionale[2].
Una linea di pensiero che accomuna ogni campo dell'informatica, considera efficienti gli algoritmi che terminano in un numero di passi polinomiale nella lunghezza dell'input, e inefficienti quelli che hanno una complessità computazionale maggiore (e.g. esponenziale). La scelta di porre un limite alle risorse di calcolo dell'avversario è dettata dal buon senso. È ragionevole infatti pensare che l'attaccante non sia infinitamente potente; è altrettanto ragionevole pensare che un attaccante non sia disposto ad impiegare un tempo *eccessivo* per violare un schema crittografico.
Se un attaccante, infatti, per violare un cifrario, utilizzasse un algoritmo che impega l'età dell'universo (stimata dai fisici intorno ai 13 miliardi di anni), sicuramente non riuscirebbe a sfruttare questa *vittoria*. È logico, quindi,

[1] one-time pad ne è un esempio lampante.
[2] un avversario è, alla fine dei conti, una macchina di Turing che esegue un algoritmo.

pensare che gli avversari vogliano essere *efficienti*. Può sembrare, quindi, naturale immaginare gli avversari come degli algoritmi che terminano in un numero polinomiale di passi rispetto alla lunghezza dell'input.
Come si può notare, non si fa alcuna assunzione particolare sul comportamento dell'avversario. Le uniche cose che sappiamo sono che:

- l'avversario non conosce la chiave, ma conosce l'algoritmo di cifratura utilizzato e i parametri di sicurezza, come per esempio la lunghezza della chiave[3].

- l'avversario vuole essere efficiente, ovvero polinomiale.

Non si fanno ipotesi sull'algoritmo che questo andrà ad eseguire. Per esempio dato un messaggio cifrato $c = E_k(m)$, non ci aspettiamo che l'avversario non decida di utilizzare la stringa c' tale che: $c' = D_{k'}(E_k(m))$ con $k \neq k'$. Ovvero, sarebbe sbagliato supporre che l'avversario non cerchi di decifrare un messaggio mediante una chiave diversa da quella utilizzata per cifrarlo. Nel modello computazionale i messaggi sono stringhe di bit e l'avversario può effettuare qualsiasi operazione su queste. Questa visione è, a differenza di quella che si ha nel modello formale, sicuramente molto più realistica[Dwo06].

Non bisogna però dimenticare che un avversario può sempre *indovinare* il segreto che cerchiamo di nascondere, o che cifriamo. Per esempio: se il segreto che si cerca di nascondere ha una lunghezza di n bit, l'avversario può sempre lanciare una moneta n volte e associare, via via, la testa della moneta al valore 1 e la croce al valore 0. La probabilità che l'avversario ottenga una stringa uguale al segreto è ovviamente di $\frac{1}{2^n}$. Questa probabilità tende a 0 in modo esponenziale al crescere della lunghezza del segreto, ma per valori finiti di n questa probabilità non sarà mai 0. È quindi più realistico cercare di rappresentare l'avversario come un algoritmo che, oltre a terminare in tempo polinomiale, ha anche la possibilità di effettuare scelte random. La classe dei problemi risolti da questo tipo di algoritmi è indicata con la sigla *BPP* (i.e. *Bounded-Probability Polynomial Time*).
Un modo più formale di vedere questo tipo di algoritmi è il seguente: si suppone che la macchina di Turing che esegue l'algoritmo, oltre a ricevere l'input, diciamo x, riceva anche un input ausiliario r. Questa stringa di bit r, rappresenta una possibile sequenza di lanci di moneta. Quando la macchina dovrà effettuare una scelta random, non dovrà far altro che prendere

[3]Un famoso principio della crittografia afferma che l'algoritmo di cifratura non deve essere segreto e deve poter cadere nelle mani del nemico senza inconvenienti.

il successivo bit dalla stringa r, e prendere una decisione in base ad esso (è, in effetti, come se avesse preso una decisione lanciando una moneta). Ecco quindi che il nostro tipico avversario si configura come un algoritmo polinomiale probabilistico. È inoltre giustificato cercare di rendere sicuri[4] gli schemi crittografici rispetto, principalmente, a questo tipo di avversario. Con questa scelta si cerca di rispettare il più possibile un famoso principio di Kerckhoffs[5] che afferma:

> *Un cifrario deve essere, se non matematicamente, almeno praticamente indecifrabile.*

Non è quindi necessario dimostrare che un particolare schema crittografico sia inviolabile, ma basta dimostrare che:

- in tempi ragionevoli lo si può violare solo con scarsissima probabilità.
- lo si può violare con alta probabilità, ma solo in tempi non ragionevoli.

Sappiamo che il concetto di *tempo ragionevole* è catturato dalla classe degli algoritmi polinomiali probabilistici. Vediamo ora di catturare il concetto di *scarsa probabilità*.

2.2 Funzioni Trascurabili e non ...

In crittografia i concetti di *scarsa probabilità* e di evento *raro* vengono formalizzati attraverso la nozione di funzione trascurabile.

Definizione 2.1 *Funzione Trascurabile (negligible). Sia $\mu : \mathbb{N} \to \mathbb{R}^+$ una funzione. Si dice che μ è trascurabile se e solo se per ogni polinomio p, esiste $C \in \mathbb{N}$ tale che $\forall n > C$: $\mu(n) < \frac{1}{p(n)}$.*

Una funzione trascurabile, quindi, è una funzione che tende a 0 più velocemente dell'inverso di qualsiasi polinomio. Un'altra definizione utile è la seguente:

Definizione 2.2 *Funzione Distinguibile (noticeable). Sia $\mu : \mathbb{N} \to \mathbb{R}^+$ una funzione. Si dice che μ è distinguibile se e solo se esiste un polinomio p, tale per cui esiste $C \in \mathbb{N}$ tale che $\forall n > C$: $\mu(n) > \frac{1}{p(n)}$.*

[4] In qualsiasi modo si possa intendere il concetto di sicurezza. Vedremo che in seguito si daranno delle definizioni rigorose di questo concetto.

[5] Auguste Kerckhoffs (19 Gennaio 1835 – 9 Agosto 1903) fu un linguista olandese e un famoso crittografo.

Per esempio la funzione $n \to 2^{-\sqrt{n}}$ è una funzione trascurabile, mentre la funzione $n \to \frac{1}{n^2}$ non lo è. Ovviamente esistono anche funzioni che non sono né trascurabili né distinguibili. Per esempio, la seguente funzione definita per casi: $f(n) = \begin{cases} 1, & \text{se } n \text{ è pari} \\ 0, & \text{se } n \text{ è dispari} \end{cases}$ non è né trascurabile né distinguibile. Questo perché le definizioni precedenti, pur essendo molto legate, non sono l'una la negazione dell'altra.

Se sappiamo che, in un esperimento, un evento avviene con una probabiltà trascurabile, quest'evento si verificherà con una probabilità trascurabile anche se l'esperimento viene ripetuto molte volte (ma sempre un numero polinomiale di volte), e quindi per la legge dei grandi numeri, con una frequenza anch'essa trascurabile[6]. Le funzioni trascurabili, infatti, godono di due particolari proprietà di chiusura, enunciate nella seguente:

Proposizione 2.1 *Siano μ_1, μ_2 due funzioni trascurabili e sia p un polinomio. Se $\mu_3 = \mu_1 + \mu_2$, e $\mu_4 = p \cdot \mu_1$, allora μ_3, μ_4 sono funzioni trascurabili.*

Se quindi, in un esperimento, un evento avviene solo con probabilità trascurabile, ci aspettiamo che, anche se ripetiamo l'esperimento un numero polinomiale di volte, questa probabilità rimanga comunque trascurabile.
Per esempio: supponiamo di avere un dado truccato in modo che la probabilità di ottenere 1 sia trascurabile. Allora se lanciamo il dado un numero polinomiale di volte, la probabilità che esca 1 rimane comunque trascurabile.
È ora importantissimo notare che: **gli eventi che avvengono con una probabilità trascurabile possono essere ignorati per fini pratici**. In [KL07], infatti leggiamo:

> *Events that occur with negligible probability are so unlikely to occur that can be ignored for all practical purposes. Therefore, a break of a cryptographic scheme that occurs with negligible probability is not significant.*

Potrebbe sembrare pericoloso utilizzare degli schemi crittografici che ammettono di essere violati con probabilità trascurabile, ma questa possibilità è cosi remota, che se ci preoccupassimo, per amor di coerenza, dovremmo anche essere ragionevolmente sicuri di fare sei all'enalotto giocando una schedina semplice. Finora abbiamo parlato sempre di funzioni che prendono in input un argomento non meglio specificato. Al crescere di questo parametro, le funzioni si comportano in modo diverso, a seconda che siano trascurabili, oppure no. Ma cosa rappresenta nella realtà questo input? Di solito, questo

[6] In modo informale, la legge debole dei grandi numeri afferma che: per un numero grande di prove, la frequenza approssima la probabilità di un evento.

valore rappresenta un generico parametro di sicurezza, indipendente dal segreto. È comune immaginarlo come la lunghezza in bit delle chiavi.
D'ora in poi con affermazioni del tipo «ł'algoritmo è polinomiale, o esponenziale», si intenderanno algoritmi polinomiali o esponenziali nella lunghezza (in bit) del parametro di sicurezza (indicato con n). Si utilizzerà questa assunzione anche quando si faranno affermazioni su funzioni trascurabili o meno. Quelle funzioni saranno trascurabili o meno nel parametro n. Tutte le definizioni di sicurezza che vengono date nel modello computazionale e che utilizzano le probabilità trascurabili, sono di tipo *asintotico*. Un template di definizione di sicurezza è il seguente [KL07]:

> *A scheme is secure if for every probablistic polynomial-time adversary* ***A*** *[...], the probability that* ***A*** *succeds in this attack [...] is negligible*

Essendo questo schema di definizione asintotico (nel parametro di sicurezza n), è ovvio che non considera valori piccoli di n. Quindi se si dimostra che un particolare schema crittografico è sicuro secondo una definizione di questo tipo, può benissimo capitare che per valori piccoli di n lo schema sia violabile con alta probabilità e in tempi ragionevoli.

2.3 Indistinguibilità Computazionale

Se due oggetti, sebbene profondamente diversi fra loro, non possono essere distinti, allora sono da un certo punto di vista equivalenti. Nel caso della crittografia computazionale, due oggetti sono computazionalmente equivalenti se nessun algoritmo efficiente li può distinguere. Possiamo immaginare che un algoritmo riesca a distinguere due oggetti, se quando gli si da in input il primo, lui da in output una costante c, mentre se gli si fornisce come input il secondo da in output una costante c' e ovviamente $c \neq c'$. La definizione tipica di indistinguibilità computazionale è data prendendo come oggetti da distinguere alcune particolari distribuzioni statistiche detti *ensembles*.

Definizione 2.3 *Ensemble. Sia I un insieme numerabile infinito.*
$X = \{X_i\}_{i \in I}$ è un ensemble su I se e solo se è una sequenza di variabili statistiche.

Un *ensemble* è quindi una sequenza infinita di distribuzioni di probabilità[7]. Tipicamente le variabili dell'ensemble sono stringhe di lunghezza i. X_i è quindi una distribuzione di probabilità su stringhe di lunghezza i.

[7] siccome si parla di distribuzioni su stringhe di bit con lunghezza finita, in crittografia computazionale si considerano ensemble che sono una sequenza inifinta di distribuzioni finite di stringhe di bit.

Ora supponiamo di avere due ensemble X e Y. Intuitivamente queste distribuzioni sono indistinguibili se nessun algoritmo (efficiente) può accettare infiniti elementi di X_n (per esempio stampando 1 su input preso da X_n) e scartare infiniti elementi di Y_n (per esempio stampare 0 su input preso da Y_n). È importante notare che sarebbe facile distinguere due *singole* distribuzioni usando un approccio esaustivo, ecco perché si considerano sequenze infinite di ditribuzioni finite. In poche parole questi ensemble sono indistinguibili se ogni algoritmo (efficiente) accetta $x \in X_n$ se e solo se accetta $y \in Y_n$. Ovviamente il *se e solo se* non può e non deve essere inteso in senso *classico*, ma deve essere inteso in senso statistico. Poiché in crittografia si è soliti indicare con U_m una variabile uniformemente distribuita sull'insieme delle stringhe di lunghezza m, chiameremo $U = \{U_n\}_{n\in\mathbb{N}}$ l'ensemble uniforme. Dopo questa breve introduzione all'indistinguibilità siamo pronti per dare una definizione rigorosa:

Definizione 2.4 *Indistinguibilità computazionale. Due ensemble* $X = \{X_n\}$, $Y = \{Y_n\}$ *sono computazionalmente indistinguibili se e solo se per ogni algoritmo* $D \in BPP$ *(detto distinguitore) esiste* μ *trascurabile tale che:* $|Pr[D(1^n, X_n) = 1] - Pr[D(1^n, Y_n) = 1]| \leq \mu(n)$.

Nella definizione precedente: $Pr[D(1^n, X_n) = 1]$ è la probabilità che, scegliendo x secondo la distribuzione X_n e fornendo questo valore al distinguitore insieme al valore 1^n, il distinguitore stampi 1. Il fatto che al distinguitore si fornisca anche il valore del parametro di sicurezza in base unaria, serve ad esser sicuri che in ogni caso il distinguitore impieghi un tempo polinomiale nel parametro di sicurezza. Infatti, il distinguitore quando si troverà a dover leggere il primo parametro, necessariamente impiegherà un tempo polinomiale nel parametro di sicurezza, visto che questo è stato fornito in base unaria[8].

La definizione di indistinguibilità computazionale cattura quindi il seguente concetto: se due ensemble sono computazionalmente indistinguibili, allora la probababilità che un distinguitore riesca a discernere i valori provenienti da un insieme rispetto all'altro è trascurabile; di conseguenza agli occhi del distinguitore gli ensemble non sono differenti e quindi sono per lui equivalenti (o meglio computazionalmente equivalenti o ancora, indistinguibili in tempo polinomiale). Non è raro, nell'ambito scientifico in particolare, basarsi sul concetto generale di indistinguibilità al fine di creare nuove classi di equivalenza di oggetti.

[8] ignoreremo, d'ora in poi, questo cavillo formale.

> *The concept of efficient computation leads naturally to a new kind of equivalence between objects: Objects are considered to be computationally equivalent if they cannot be differentiated by any efficient procedure.* We note that considering indistinguishable objects as equivalent is one of the basics paradigms of both science and real-life situations. Hence, we believe that the notion of computational indistinguishability is a very natural one.[Gol00]

2.4 Pseudocasualità e Generatori Pseudocasuali

Argomento centrale di questa sezione è il concetto di *pseudocasualità* (pseudorandomness), applicato a stringhe di bit di lunghezza finita. Parlare di pseudocasualità applicata ad una *singola* stringa, ha poco senso quanto ne ha poco parlare di singola stringa casuale (random). Il concetto di casualità (come quello di pseudocasualità) si applica, infatti, a distribuzioni di oggetti (stringhe di bit nel nostro caso) e non a singoli oggetti.
La nozione di casualità è fortemente legata a quella di distribuzione uniforme. Un insieme di oggetti è caratterizzato da una distribuzione uniforme se la probabilità è equamente distribuita su tutti gli oggetti. Quindi *l'estrazione* di un elemento è del tutto casuale, perché non ci sono elementi più probabili di altri.

Il concetto di pseudorandomness è un caso particolare di indistinuguibilità, infatti una distribuzione è *pseudorandom* se nessuna procedura efficiente, può distinguerla dalla distribuzione uniforme.

Definizione 2.5 *Pseudorandomness. L'ensemble* $X = \{X_n\}_{n\in\mathbb{N}}$ *è detto pseudorandom se e solo se* $\exists l : \mathbb{N} \to \mathbb{N}$ *tale che:* X *è computazionalmente indistinguibile da* $U = \{U_{l(n)}\}_{n\in\mathbb{N}}$.

Data questa definizione, possiamo finalmente definire formalmente cosa sia un generatore pseudorandom.

Definizione 2.6 *Generatore Pseudorandom. Sia* $l : \mathbb{N} \to \mathbb{N}$ *un polinomio detto fattore d'espansione. Sia* G *un algoritmo polinomiale deterministico tale che:* $\forall s \in \{0,1\}^n\ G(s) \in \{0,1\}^{l(n)}$. *Allora* G *è un generatore pseudorandom se e solo se valgono le seguenti condizioni:*

- *Espansione:* $\forall n : l(n) > n$
- *Pseudocasualità:* $\forall D \in BPP, \exists \mu$ *trascurabile tale che*
  $|Pr[D(r) = 1] - Pr[D(G(s)) = 1]| \leq \mu(n)$
  con $r \in U_{l(n)}$ *e* $s \in U_n$

Quindi: se data una stringa di bit $s \in U_n$, nessun distinguitore efficiente riesce a distinguere (con una probabilità non trascurabile) $G(s)$ da una stringa $r \in U_{l(n)}$, allora G è un generatore pseudorandom. Il suo output, infatti, non è distinguibile dalla distribuzione effettivamente uniforme.

È importante però notare, che la stringa in output di un generatore pseudorandom è fortemente differente da una stringa effettivamente random. Per rendere più chiara questa distinzione procederemo con un importante esempio. Supponiamo di avere un generatore pseudorandom G con fattore d'espansione $l(n) = 2n$. L'insieme $A = \{0,1\}^{2n}$ ha, ovviamente, una cardinalità pari a 2^{2n}. Fissando quindi una certa stringa $x \in A$, questa ha una probabilità di esser scelta in maniera random pari a: $\frac{1}{|A|} = \frac{1}{2^{2n}}$.

Ragioniamo adesso sull'output del generatore G. Questo prende un input appartenente al dominio: $B = \{0,1\}^n$. Anche considerando il caso *migliore* di un generatore iniettivo[9], il codominio di G avrà una cardinalità pari a quella del dominio ovvero 2^n. La maggior parte degli elementi dell'insieme A non ricadrà nell'output di G; questo a causa dell'abissale differenza di cardinalità fra gli insiemi $G(B)$ e A. Quindi la probabilità che una stringa scelta in maniera uniforme dall'insieme A ricada nel codominio di G è di $\frac{2^n}{2^{2n}}$, cioè 2^{-n}. In teoria, quindi, è facile immaginare un distinguitore D che riesca a discernere l'output di G dalla distribuzione uniforme con probabilità non trascurabile. Supponiamo che D prenda in input $y \in A$. Tutto ciò che D deve fare è ricercare in modo esaustivo un $w \in B$ tale che $G(w) = y$. Se $y \in G(B)$ allora D se ne accorgerà con probabilità 1, mentre se y è stato scelto in maniera uniforme dall'insieme A, D stamperà 1 con probabilità 2^{-n}. Quindi abbiamo che:

$$|Pr[D(r) = 1] - Pr[D(G(s)) = 1]| \geq 1 - 2^{-n}$$

con $r \xleftarrow{R} A$ e $s \xleftarrow{R} B$[10].
Il membro a destra della disequazione è una funzione distinguibile. Sembrerebbe quindi che G non sia un generatore pseudorandom. C'è un' importante constatazione da fare però. Il distinguitore D non è efficiente! Infatti impiega un tempo esponenziale nel parametro n, e non polinomiale. La distribuzione generata da G dunque, è si ben lontana dall'essere uniforme, ma questo non è importante dal momento che nessun distinguitore che viaggia in tempo polinomiale può accorgersene.

Nella pratica lo scopo di G è prendere in input un *seed* random, e da quello generare una variabile pseudocasuale molto più lunga. Si inutisce da

[9] una generica funzione f è iniettiva se e solo se $\forall x_1, x_2 : x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$.

[10] con la notazione $s \xleftarrow{R} O$, si intende la scelta dell'elemento $s \in O$ in maniera random.

questo la grandissima importanza che hanno i generatori pseudorandom in crittografia. Per esempio il seed potrebbe corrispondere alla chiave di un cifrario, mentre l'output di G di lunghezza k potrebbe essere il valore con cui viene fatto lo XOR del messaggio (anch'esso di lunghezza k); otteniamo così una versione del one-time pad basato su una chiave più corta del messaggio. Siccome una stringa pseudorandom appare, ad un distinguitore efficiente D, come una stringa random, D non ottiene un vantaggio sensibile nel passaggio dal vero one-time pad al one-time pad che usa una chiave pseudorandom. In generale i generatori pseudorandom sono molto utili in crittografia per creare schemi crittografici simmetrici.

2.5 Dimostrazioni Basate su Games

In questo paragrafo si cercherà di spiegare cosa siano nell'ambito della crittografia i *games* (tradotti letteralmente con la parola giochi) e come sono strutturate la maggior parte delle dimostrazioni che utilizzano sequenze di games.
Si possono trovare approfondimenti nel tutorial di Shoup[Sho08]. Quella basata sul concetto di game è una tecnica molto utilizzata per provare la sicurezza di primitive crittografiche o di protocolli crittografici. Questi games sono giocati da un' ipotetica entità maligna, l'attacante, e da un'ipotetica parte benigna di solito chiamato sfidante[11]. È difficile dare una definizione formale di *game* perchè in letteratura non ve ne sono. Però possono intuitivamente essere immaginati come un insieme di azioni che servono a specificare il comportamento dei partecipanti al gioco, ovvero il protocollo crittografico. Di solito una definizione di sicurezza è associata ad un evento che dovrebbe avvenire con probabilità trascurabilmente vicina ad una qualche costante come 0 o $\frac{1}{2}$.
Per esempio nel caso della proprietà di sicurezza detta *indistinguishable encryptions in the presence of an eavesdropper*, supponiamo di fare l'esperimento mentale seguente:

- L'avversario **A** sceglie due messaggi m_0, m_1.
- Lo sfidante **S** genera una chiave k e un bit $b \xleftarrow{R} \{0,1\}$.
- **S** crea il messaggio c, tale che $c = Enc_k(m_b)$ e lo da in input ad **A**.
- **A** stampa un bit b'

[11] perchè *sfida* l'attaccante a vincere questo gioco.

Questo game descrive il comportamento che tengono lo sfidante e il challenger. Sia E l'evento: "**A** stampa b' e $b = b'$".
Definiamo sicuro il sistema se $\exists\mu$ trascurabile tale che $Pr[E] \leq \frac{1}{2} + \mu(n)$. Il game di cui sopra, e il la $Pr[E]$ descrivono il fatto che un attaccante non riesce a distinguere fra loro due messaggi cifrati e quindi la probabilità che riesca ad indovinare il bit b è trascurabilmente superiore alla costante $\frac{1}{2}$.

L'utilizzo che si fa dei games non è limitato a definizioni di sicurezza, ma si estende a cneh a vere e proprie dimostrazioni. Il principio che seguono questo tipo di dimostrazioni è quello di partire da un game iniziale G_0 in cui un evento *negativo* può accadere con probabilità non nulla, e poi effettuando delle modifiche al game G_i si passa al G_{i+1} tale che G_i e G_{i+1} sono computazionalmente indistinguibili. Alla fine si arriva ad un game G_f in cui l'evento non può accadere. Si può cosi dare un limite superiore alla probabilità che l'evento avvenga nel game iniziale. Questo limite è la somma di tutte le probabilità con cui un attacante riesce a distinguere un game dal successivo.

Capitolo 3

CryptoVerif

3.1 Introduzione al Tool

CryptoVerif è un *dimostratore* automatico che lavora direttamente nel modello computazionale. Questo tool è utilizzato per dimostrare le proprietà di segretezza e autenticazione. Questo dimostratore dimostra i suoi risultati basandosi sulla tecnica dei game e le sue prove di sicurezza sono valide per un numero polinomiale di game nel parametro di sicurezza. Il tool può inoltre calcolare la probabilità di successo di un attaccante come funzione di alcuni parametri come il tempo impiegato la cardinalità degli insiemi in gioco ecc, ecc... Il tool è liberamente scaricabile sotto la licenza CeCill[1] all'url [xxx]. L'autore principale di CryptoVerif è Bruno Blanchet [xxx]. Il tool è stato scritto in ML con 18000 righe di codice. È un tool molto recente ed è in continuo sviluppo. Un'altra strada è quella di dimostrare la correttezza del protocollo nel modello formale e poi di sfruttare il teorema [???]...

3.2 Un Esempio: FDH

Full Domain Hash è un protocollo...blahblah..

```
Initial state
Game 1 is
(
   foreach iH_13 ≤ qH do
   OH(x : bitstring) :=
   return(hash(x ))
|
```

[1] http://www.cecill.info/licences/

```
   Ogen() :=
      R
   r ← seed ;
   pk : pkey ← pkgen(r );
   sk : skey ← skgen(r );
   return(pk );
   (
      foreach iS 14 ≤ qS do
      OS(m : bitstring) :=
      return(invf(sk , hash(m)))
   |
      OT (m : bitstring, s : D) :=
      if (f(pk , s) = hash(m )) then
      find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
         end
      else
         event forge
   )
)
Applying equivalence
foreach iH 11 ≤ nH do OH(x : bitstring) := return(hash(x ))[all]
≈0
foreach iH 12 ≤ nH do OH(x16 : bitstring) := x : bitstring ← x16 ;
  find u ≤ nH suchthat defined(x [u], r [u]) ∧ (x = x [u]) then return(r [u]
          R
  else r ← D; return(r )
yields
Game 2 is
(
   foreach iH 13 ≤ qH do
   OH(x : bitstring) :=
   x23 : bitstring ← x ;
   find suchthat defined(x19 , r18 ) ∧ (x23 = x19 ) then
      return(r18 )
   ⊕ @i 29 ≤ qS suchthat defined(x21 [@i 29 ], r20 [@i 29 ]) ∧ (x23 = x21 [@
      return(r20 [@i 29 ])
   ⊕ @i 28 ≤ qH suchthat defined(x23 [@i 28 ], r22 [@i 28 ]) ∧ (x23 = x23 [@
      return(r22 [@i 28 ])
   else
          R
      r22 ← D;
```

```
      return(r22 )
|
   Ogen() :=
      R
   r ← seed ;
   pk : pkey ← pkgen(r );
                                                                    1
  sk : skey ← skgen(r );
  return(pk );
  (
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    x21 : bitstring ← m;
    find suchthat defined(x19 , r18 ) ∧ (x21 = x19 ) then
       return(invf(sk , r18 ))
    ⊕ @i 27 ≤ qS suchthat defined(x21 [@i 27 ], r20 [@i 27 ]) ∧ (x21 = x21 [@i 27
       return(invf(sk , r20 [@i 27 ]))
    ⊕ @i 26 ≤ qH suchthat defined(x23 [@i 26 ], r22 [@i 26 ]) ∧ (x21 = x23 [@i 26
       return(invf(sk , r22 [@i 26 ]))
    else
            R
       r20 ← D;
       return(invf(sk , r20 ))
  |
    OT (m : bitstring, s : D) :=
    x19 : bitstring ← m ;
    find suchthat defined(x19 , r18 ) ∧ (x19 = x19 ) then
       if (f(pk , s) = r18 ) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
    ⊕ @i 25 ≤ qS suchthat defined(x21 [@i 25 ], r20 [@i 25 ]) ∧ (x19 = x21 [@i 25
       if (f(pk , s) = r20 [@i 25 ]) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
    ⊕ @i 24 ≤ qH suchthat defined(x23 [@i 24 ], r22 [@i 24 ]) ∧ (x19 = x23 [@i 24
       if (f(pk , s) = r22 [@i 24 ]) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
```

```
          end
       else
          event forge
    else
            R
       r18 ← D;
       if (f(pk , s) = r18 ) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
  )
)
Applying simplify yields
Game 3 is
(
  foreach iH 13 ≤ qH do
  OH(x : bitstring) :=
                                                          2
  x23 : bitstring ← x ;
  find suchthat defined(x19 , r18 ) ∧ (x23 = x19 ) then
    return(r18 )
  ⊕ @i 29 ≤ qS suchthat defined(x21 [@i 29 ], r20 [@i 29 ]) ∧ (x23 = x21 [@i
    return(r20 [@i 29 ])
  ⊕ @i 28 ≤ qH suchthat defined(x23 [@i 28 ], r22 [@i 28 ]) ∧ (x23 = x23 [@i
    return(r22 [@i 28 ])
  else
        R
    r22 ← D;
    return(r22 )
|
  Ogen() :=
    R
  r ← seed ;
  pk : pkey ← pkgen(r );
  sk : skey ← skgen(r );
  return(pk );
  (
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    x21 : bitstring ← m;
```

```
    find suchthat defined(x19 , r18 ) ∧ (x21 = x19 ) then
       return(invf(sk , r18 ))
    ⊕ @i 27 ≤ qS suchthat defined(x21 [@i 27 ], r20 [@i 27 ]) ∧ (x21 = x21 [@i 27
       return(invf(sk , r20 [@i 27 ]))
    ⊕ @i 26 ≤ qH suchthat defined(x23 [@i 26 ], r22 [@i 26 ]) ∧ (x21 = x23 [@i 26
       return(invf(sk , r22 [@i 26 ]))
    else
            R
       r20 ← D;
       return(invf(sk , r20 ))
  |
    OT (m : bitstring, s : D) :=
    x19 : bitstring ← m ;
    find @i 25 ≤ qS suchthat defined(r20 [@i 25 ], x21 [@i 25 ]) ∧ (x19 = x21 [@i 2
       end
    ⊕ @i 24 ≤ qH suchthat defined(x23 [@i 24 ], r22 [@i 24 ]) ∧ (x19 = x23 [@i 24
       if (f(pk , s) = r22 [@i 24 ]) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
    else
            R
       r18 ← D;
       if (f(pk , s) = r18 ) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
  )
)
Applying remove assignments of useless yields
                                                                    3
Game 4 is
(
  foreach iH 13 ≤ qH do
  OH(x : bitstring) :=
  x23 : bitstring ← cst bitstring;
  find suchthat defined(m , x19 , r18 ) ∧ (x = m ) then
    return(r18 )
  ⊕ @i 29 ≤ qS suchthat defined(m[@i 29 ], x21 [@i 29 ], r20 [@i 29 ]) ∧ (x = m[@i
```

```
    return(r20 [@i 29 ])
  ⊕ @i 28 ≤ qH suchthat defined(x [@i 28 ], x23 [@i 28 ], r22 [@i 28 ]) ∧ (x
    return(r22 [@i 28 ])
  else
        R
    r22 ← D;
    return(r22 )
|
  Ogen() :=
    R
  r ← seed ;
  pk : pkey ← pkgen(r );
  sk : skey ← skgen(r );
  return(pk );
  (
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    x21 : bitstring ← cst bitstring;
    find suchthat defined(m , x19 , r18 ) ∧ (m = m ) then
       return(invf(sk , r18 ))
    ⊕ @i 27 ≤ qS suchthat defined(m[@i 27 ], x21 [@i 27 ], r20 [@i 27 ]) ∧ (
       return(invf(sk , r20 [@i 27 ]))
    ⊕ @i 26 ≤ qH suchthat defined(x [@i 26 ], x23 [@i 26 ], r22 [@i 26 ]) ∧
       return(invf(sk , r22 [@i 26 ]))
    else
          R
       r20 ← D;
       return(invf(sk , r20 ))
  |
    OT (m : bitstring, s : D) :=
    x19 : bitstring ← cst bitstring;
    find @i 25 ≤ qS suchthat defined(m[@i 25 ], x21 [@i 25 ], r20 [@i 25 ]) ∧
       end
    ⊕ @i 24 ≤ qH suchthat defined(x [@i 24 ], x23 [@i 24 ], r22 [@i 24 ]) ∧
       if (f(pk , s) = r22 [@i 24 ]) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
    else
          R
```

```
       r18 ← D;
       if (f(pk , s) = r18 ) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
  )
)
                                                                    4
Applying remove assignments of binder sk yields
Game 5 is
(
  foreach iH 13 ≤ qH do
  OH(x : bitstring) :=
  x23 : bitstring ← cst bitstring;
  find suchthat defined(m , x19 , r18 ) ∧ (x = m ) then
    return(r18 )
  ⊕ @i 29 ≤ qS suchthat defined(m[@i 29 ], x21 [@i 29 ], r20 [@i 29 ]) ∧ (x = m[@i
    return(r20 [@i 29 ])
  ⊕ @i 28 ≤ qH suchthat defined(x [@i 28 ], x23 [@i 28 ], r22 [@i 28 ]) ∧ (x = x [
    return(r22 [@i 28 ])
  else
        R
    r22 ← D;
    return(r22 )
|
  Ogen() :=
    R
  r ← seed ;
  pk : pkey ← pkgen(r );
  return(pk );
  (
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    x21 : bitstring ← cst bitstring;
    find suchthat defined(m , x19 , r18 ) ∧ (m = m ) then
       return(invf(skgen(r ), r18 ))
    ⊕ @i 27 ≤ qS suchthat defined(m[@i 27 ], x21 [@i 27 ], r20 [@i 27 ]) ∧ (m = m[
       return(invf(skgen(r ), r20 [@i 27 ]))
    ⊕ @i 26 ≤ qH suchthat defined(x [@i 26 ], x23 [@i 26 ], r22 [@i 26 ]) ∧ (m = x
       return(invf(skgen(r ), r22 [@i 26 ]))
```

```
    else
            R
       r20 ← D;
       return(invf(skgen(r ), r20 ))
  |
    OT (m : bitstring, s : D) :=
    x19 : bitstring ← cst bitstring;
    find @i 25 ≤ qS suchthat defined(m[@i 25 ], x21 [@i 25 ], r20 [@i 25 ])
       end
    ⊕ @i 24 ≤ qH suchthat defined(x [@i 24 ], x23 [@i 24 ], r22 [@i 24 ]) ∧
       if (f(pk , s) = r22 [@i 24 ]) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
    else
            R
       r18 ← D;
       if (f(pk , s) = r18 ) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
                                                                    5
          event forge
  )
)
Applying equivalence
                            R
foreach iK 1 ≤ nK do r ← seed; (
  Opk() := return(pkgen(r )) |
                              R
  foreach iF 2 ≤ nF do x ← D; (
    Oant() := return(invf(skgen(r ), x )) |
    Oim() := return(x )))
≈0
                            R
foreach iK 3 ≤ nK do r ← seed; (
  Opk() := return(pkgen(r )) |
                              R
  foreach iF 4 ≤ nF do x ← D; (
    Oant() := return(x ) |
```

```
    Oim() := return(f(pkgen(r ), x ))))
with r yields
Game 6 is
(
  foreach iH 13 ≤ qH do
  OH(x : bitstring) :=
  x23 : bitstring ← cst bitstring;
  find suchthat defined(m , x19 , r18 ) ∧ (x = m ) then
    return(f(pkgen(r ), r18 ))
  ⊕ @i 29 ≤ qS suchthat defined(m[@i 29 ], x21 [@i 29 ], r20 [@i 29 ]) ∧ (x = m[@i
    return(f(pkgen(r ), r20 [@i 29 ]))
  ⊕ @i 28 ≤ qH suchthat defined(x [@i 28 ], x23 [@i 28 ], r22 [@i 28 ]) ∧ (x = x [
    return(f(pkgen(r ), r22 [@i 28 ]))
  else
        R
    r22 ← D;
    return(f(pkgen(r ), r22 ))
|
  Ogen() :=
    R
  r ← seed ;
  pk : pkey ← pkgen(r );
  return(pk );
  (
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    x21 : bitstring ← cst bitstring;
    find suchthat defined(m , x19 , r18 ) ∧ (m = m ) then
       return(r18 )
    ⊕ @i 27 ≤ qS suchthat defined(m[@i 27 ], x21 [@i 27 ], r20 [@i 27 ]) ∧ (m = m[
       return(r20 [@i 27 ])
    ⊕ @i 26 ≤ qH suchthat defined(x [@i 26 ], x23 [@i 26 ], r22 [@i 26 ]) ∧ (m = x
       return(r22 [@i 26 ])
    else
          R
       r20 ← D;
       return(r20 )
  |
                                                                    6
    OT (m : bitstring, s : D) :=
    x19 : bitstring ← cst bitstring;
```

```
    find @i 25 ≤ qS suchthat defined(m[@i 25 ], x21 [@i 25 ], r20 [@i 25 ])
       end
    ⊕ @i 24 ≤ qH suchthat defined(x [@i 24 ], x23 [@i 24 ], r22 [@i 24 ]) ∧
       if (f(pk , s) = f(pkgen(r ), r22 [@i 24 ])) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
    else
            R
       r18 ← D;
       if (f(pk , s) = f(pkgen(r ), r18 )) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
  )
)
Applying simplify yields
Game 7 is
(
  foreach iH 13 ≤ qH do
  OH(x : bitstring) :=
  x23 : bitstring ← cst bitstring;
  find suchthat defined(m , r , r18 ) ∧ (x = m ) then
    return(f(pkgen(r ), r18 ))
  ⊕ @i 29 ≤ qS suchthat defined(m[@i 29 ], r , r20 [@i 29 ]) ∧ (x = m[@i 29
    return(f(pkgen(r ), r20 [@i 29 ]))
  ⊕ @i 28 ≤ qH suchthat defined(x [@i 28 ], r22 [@i 28 ]) ∧ (x = x [@i 28 ]
    return(f(pkgen(r ), r22 [@i 28 ]))
  else
        R
    r22 ← D;
    return(f(pkgen(r ), r22 ))
|
  Ogen() :=
    R
  r ← seed ;
  pk : pkey ← pkgen(r );
  return(pk );
  (
```

```
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    x21 : bitstring ← cst bitstring;
    find suchthat defined(m , r18 ) ∧ (m = m ) then
       return(r18 )
    ⊕ @i 27 ≤ qS suchthat defined(m[@i 27 ], r20 [@i 27 ]) ∧ (m = m[@i 27 ]) then
       return(r20 [@i 27 ])
    ⊕ @i 26 ≤ qH suchthat defined(x [@i 26 ], r22 [@i 26 ]) ∧ (m = x [@i 26 ]) the
       return(r22 [@i 26 ])
    else
                                                                  7
            R
       r20 ← D;
       return(r20 )
  |
    OT (m : bitstring, s : D) :=
    x19 : bitstring ← cst bitstring;
    find @i 25 ≤ qS suchthat defined(r20 [@i 25 ], m[@i 25 ]) ∧ (m = m[@i 25 ]) the
       end
    ⊕ @i 24 ≤ qH suchthat defined(x [@i 24 ], r22 [@i 24 ]) ∧ (m = x [@i 24 ]) the
       if (s = r22 [@i 24 ]) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
    else
            R
       r18 ← D;
       if (s = r18 ) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
  )
)
Applying remove assignments of useless yields
Game 8 is
(
  foreach iH 13 ≤ qH do
  OH(x : bitstring) :=
  find suchthat defined(m , r , r18 ) ∧ (x = m ) then
```

```
    return(f(pkgen(r ), r18 ))
  ⊕ @i 29 ≤ qS suchthat defined(m[@i 29 ], r , r20 [@i 29 ]) ∧ (x = m[@i 29
    return(f(pkgen(r ), r20 [@i 29 ]))
  ⊕ @i 28 ≤ qH suchthat defined(x [@i 28 ], r22 [@i 28 ]) ∧ (x = x [@i 28 ]
    return(f(pkgen(r ), r22 [@i 28 ]))
  else
        R
    r22 ← D;
    return(f(pkgen(r ), r22 ))
|
  Ogen() :=
    R
  r ← seed ;
  pk : pkey ← pkgen(r );
  return(pk );
  (
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    find suchthat defined(m , r18 ) ∧ (m = m ) then
       return(r18 )
    ⊕ @i 27 ≤ qS suchthat defined(m[@i 27 ], r20 [@i 27 ]) ∧ (m = m[@i 27 ]
       return(r20 [@i 27 ])
    ⊕ @i 26 ≤ qH suchthat defined(x [@i 26 ], r22 [@i 26 ]) ∧ (m = x [@i 26
       return(r22 [@i 26 ])
                                                                  8
    else
          R
       r20 ← D;
       return(r20 )
  |
    OT (m : bitstring, s : D) :=
    find @i 25 ≤ qS suchthat defined(r20 [@i 25 ], m[@i 25 ]) ∧ (m = m[@i 25
       end
    ⊕ @i 24 ≤ qH suchthat defined(x [@i 24 ], r22 [@i 24 ]) ∧ (m = x [@i 24
       if (s = r22 [@i 24 ]) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
    else
          R
```

```
       r18 ← D;
       if (s = r18 ) then
       find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
          end
       else
          event forge
  )
)
Applying equivalence
                                R
foreach iK 5 ≤ nK do r ← seed; (
  Opk() := return(pkgen(r )) |
                                  R
  foreach iF 6 ≤ nF do x ← D; (
    Oy() := return(f(pkgen(r ), x )) |
    foreach i1 7 ≤ n1 do Oeq(x : D) := return((x = x )) |
    Ox() := return(x )))
≈nK ×nF ×POW (time+(nK −1.)×time(pkgen)+(nF ×nK −1.)×time(f))
                                R
foreach iK 8 ≤ nK do r ← seed; (
  Opk() := return(pkgen (r )) |
                                  R
  foreach iF 9 ≤ nF do x ← D; (
    Oy() := return(f (pkgen (r ), x )) |
    foreach i1 10 ≤ n1 do Oeq(x : D) := if defined(k ) then return((x = x )) else r
    Ox() := k : bitstring ← mark; return(x )))
[Difference of probability POW (qS × time(f) + qH × time(f) + time + time(context f
qS × POW (qS × time(f) + qH × time(f) + time + time(context for game 8))+
qH × POW (qS × time(f) + qH × time(f) + time + time(context for game 8))] yields
Game 9 is
(
  foreach iH 13 ≤ qH do
  OH(x : bitstring) :=
  find suchthat defined(m , r , r18 ) ∧ (x = m ) then
    return(f (pkgen (r ), r18 ))
  ⊕ @i 29 ≤ qS suchthat defined(m[@i 29 ], r , r20 [@i 29 ]) ∧ (x = m[@i 29 ]) then
    return(f (pkgen (r ), r20 [@i 29 ]))
  ⊕ @i 28 ≤ qH suchthat defined(x [@i 28 ], r22 [@i 28 ]) ∧ (x = x [@i 28 ]) then
    return(f (pkgen (r ), r22 [@i 28 ]))
                                                                    9
  else
```

```
        R
    r22 ← D;
    return(f (pkgen (r ), r22 ))
|
  Ogen() :=
    R
  r ← seed ;
  pk : pkey ← pkgen (r );
  return(pk );
  (
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    find suchthat defined(m , r18 ) ∧ (m = m ) then
       k47 : bitstring ← mark;
       return(r18 )
    ⊕ @i 27 ≤ qS suchthat defined(m[@i 27 ], r20 [@i 27 ]) ∧ (m = m[@i 27 ]
       k48 : bitstring ← mark;
       return(r20 [@i 27 ])
    ⊕ @i 26 ≤ qH suchthat defined(x [@i 26 ], r22 [@i 26 ]) ∧ (m = x [@i 26
       k50 : bitstring ← mark;
       return(r22 [@i 26 ])
    else
            R
       r20 ← D;
       k45 : bitstring ← mark;
       return(r20 )
  |
    OT (m : bitstring, s : D) :=
    find @i 25 ≤ qS suchthat defined(r20 [@i 25 ], m[@i 25 ]) ∧ (m = m[@i 25
       end
    ⊕ @i 24 ≤ qH suchthat defined(x [@i 24 ], r22 [@i 24 ]) ∧ (m = x [@i 24
       find @i 56 ≤ qS suchthat defined(k50 [@i 56 ]) ∧ (@i 24 = @i 26 [@i 5
         if (s = r22 [@i 24 ]) then
         find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
            end
         else
            event forge
       else
         if false then
         find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
            end
```

```
          else
             event forge
    else
             R
       r18 ← D;
       find @i 53 ≤ qS suchthat defined(k47 [@i 53 ]) then
         if (s = r18 ) then
         find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
             end
         else
             event forge
       else
         if false then
         find u ≤ qS suchthat defined(m[u]) ∧ (m = m[u]) then
             end
                                                              10
         else
             event forge
  )
)
Applying simplify yields
Game 10 is
(
  foreach iH 13 ≤ qH do
  OH(x : bitstring) :=
  find suchthat defined(m , r , r18 ) ∧ (x = m ) then
    return(f (pkgen (r ), r18 ))
  ⊕ @i 29 ≤ qS suchthat defined(m[@i 29 ], r , r20 [@i 29 ]) ∧ (x = m[@i 29 ]) the
    return(f (pkgen (r ), r20 [@i 29 ]))
  ⊕ @i 28 ≤ qH suchthat defined(x [@i 28 ], r22 [@i 28 ]) ∧ (x = x [@i 28 ]) then
    return(f (pkgen (r ), r22 [@i 28 ]))
  else
         R
    r22 ← D;
    return(f (pkgen (r ), r22 ))
|
  Ogen() :=
    R
  r ← seed ;
  pk : pkey ← pkgen (r );
  return(pk );
```

```
  (
    foreach iS 14 ≤ qS do
    OS(m : bitstring) :=
    find suchthat defined(m , r18 ) ∧ (m = m ) then
       k47 : bitstring ← mark;
       return(r18 )
    ⊕ @i 27 ≤ qS suchthat defined(m[@i 27 ], r20 [@i 27 ]) ∧ (m = m[@i 27 ]
       k48 : bitstring ← mark;
       return(r20 [@i 27 ])
    ⊕ @i 26 ≤ qH suchthat defined(x [@i 26 ], r22 [@i 26 ]) ∧ (m = x [@i 26
       k50 : bitstring ← mark;
       return(r22 [@i 26 ])
    else
            R
       r20 ← D;
       k45 : bitstring ← mark;
       return(r20 )
  |
    OT (m : bitstring, s : D) :=
    find @i 25 ≤ qS suchthat defined(r20 [@i 25 ], m[@i 25 ]) ∧ (m = m[@i 25
       end
    ⊕ @i 24 ≤ qH suchthat defined(r22 [@i 24 ], x [@i 24 ]) ∧ (m = x [@i 24
       end
    else
            R
       r18 ← D
  )
)
                                                              11
Applying remove assignments of useless yields
Game 11 is
(
   foreach iH 13 ≤ qH do
   OH(x : bitstring) :=
   find suchthat defined(m , r , r18 ) ∧ (x = m ) then
     return(f (pkgen (r ), r18 ))
   ⊕ @i 29 ≤ qS suchthat defined(m[@i 29 ], r , r20 [@i 29 ]) ∧ (x = m[@i 29
     return(f (pkgen (r ), r20 [@i 29 ]))
   ⊕ @i 28 ≤ qH suchthat defined(x [@i 28 ], r22 [@i 28 ]) ∧ (x = x [@i 28 ]
     return(f (pkgen (r ), r22 [@i 28 ]))
   else
```

```
          R
     r22 ← D;
     return(f (pkgen (r ), r22 ))
|
   Ogen() :=
     R
   r ← seed ;
   pk : pkey ← pkgen (r );
   return(pk );
   (
     foreach iS 14 ≤ qS do
     OS(m : bitstring) :=
     find suchthat defined(m , r18 ) ∧ (m = m ) then
        return(r18 )
     ⊕ @i 27 ≤ qS suchthat defined(m[@i 27 ], r20 [@i 27 ]) ∧ (m = m[@i 27 ]) then
        return(r20 [@i 27 ])
     ⊕ @i 26 ≤ qH suchthat defined(x [@i 26 ], r22 [@i 26 ]) ∧ (m = x [@i 26 ]) th
        return(r22 [@i 26 ])
     else
            R
        r20 ← D;
        return(r20 )
   |
     OT (m : bitstring, s : D) :=
     find @i 25 ≤ qS suchthat defined(r20 [@i 25 ], m[@i 25 ]) ∧ (m = m[@i 25 ]) th
        end
     ⊕ @i 24 ≤ qH suchthat defined(r22 [@i 24 ], x [@i 24 ]) ∧ (m = x [@i 24 ]) th
        end
     else
            R
        r18 ← D
   )
)
Proved event forge ==⇒ false with probability
qH × POW (qS × time(f) + qH × time(f) + time + time(context for game 8))+
qS × POW (qS × time(f) + qH × time(f) + time + time(context for game 8))+
POW (qS × time(f) + qH × time(f) + time + time(context for game 8))
RESULT time(context for game 8) =
2. × qS × time(=bitstring, maxlength(game 8 : m ), maxlength(game 8 : m[iS 14 ]))+
qH × time(=bitstring, maxlength(game 8 : m ), maxlength(game 8 : x [iH 13 ]))+
qH × time(=bitstring, maxlength(game 8 : m[iS 14 ]), maxlength(game 8 : x [iH 13 ])
```

```
qS × qS × time(=bitstring, maxlength(game 8 : m[iS 14 ]), maxlength(game 8 :
time(=bitstring, maxlength(game 8 : m[iS 14 ]), maxlength(game 8 : m )) × qS
                                                                    12
qH × qH × time(=bitstring, maxlength(game 8 : x [iH 13 ]), maxlength(game 8
qS × time(=bitstring, maxlength(game 8 : x [iH 13 ]), maxlength(game 8 : m[i
time(=bitstring, maxlength(game 8 : x [iH 13 ]), maxlength(game 8 : m )) × q
All queries proved.
                                                                  13
```

Capitolo 4

Risultati Raggiunti

Capitolo 5

Conclusioni

Bibliografia

[AR07] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 20(3):395, 2007.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[Dwo06] Cynthia Dwork, editor. *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*. Springer, 2006.

[Gol00] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[Sho08] Victor Shoup. Sequences of games: A tool for taming complexity in security proofs. 2008.