Introduzione

Negli anni anni '80 la crittografia ha cessato di essere un' arte per assurgere allo stato di scienza.
Come scienza, la crittografia moderna si propone innanzi tutto di fornire definizioni rigorose dei concetti con cui ha a che fare (e.g. *segretezza*) per poi apportare prove dei propri asserti basandosi su argomentazioni di carattere logico-matematico, ovvero di dimostrazioni.
Sebbene in alcuni casi sia possibile (ed è stato fatto, i.e [Sha49]) provare dei risultati *incondizionatamente* la maggior parte delle prove di teoremi nel campo della crittografia moderna sono basate su assunzioni che non sono ancora certe, ma che sono comunque ben formalizzate e inequivocabilmente descritte.
La più importante è sicuramente l'ipotesi dell'esistenza di funzioni *one-way*. Tutte le prove che sono basate su assunzioni non verificate sono sempre e comunque valide, sia che le assunzioni fatte vengano dimostrate, sia che vengano confutate; questo perchè ogni dimostrazione è del tipo: *Se X allora Y* (e.g. se esiste una funzione one-way allora esiste un generatore pseudocasuale con fattore d'espansione polinomiale). Semplicemente, nel caso in cui le assunzioni vengano confutate si renderebbero poco interessanti le dimostrazioni che si basano su queste[1]. È però giusto dire che le assunzioni che vengono fatte sono largamente ritenute vere; questo anche grazie al fatto che alcune di esse si possono dedurre da altre ipotesi largamente ritenute vere sebbene non dimostrate.
Lo studio di una congettura infatti, può fornire evidenti prove della sua validità mostrando che essa è la tesi di un teorema che assume come ipotesi un'altra congettura largamente ritenuta valida. Da quando la crittografia si è guadagnata una validità scientifica, sono nati almeno due modi profondamente diversi fra loro di studiarla.
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 che hanno una semantica probabilistica. Il primo modello è stato ampiamente trattato in [AG99, BAN90, Kem87, Pau98]. Il secondo modello trova le basi in lavori di

[1] È ovvio che partendo da ipotesi false si possa dimostrare qualsiasi cosa.

altrettanto illustri studiosi [**?**].
Il modello formale può contare su un vastissimo insieme di conoscenze teoriche derivanti da altri rami dell'informatica, in particolare la teoria dei linguaggi formali e della logica; anche per questo il modello formale è stato sicuramente trattato in modo più approfondito, almeno fino ad ora. Questo ha fatto sì che lo stato dell'arte veda, per esempio, molti più tool di verifica automatica che lavorano nel modello formale rispetto a tool che lavorano nel modello computazionale. Uno tra i più famosi tool che lavora nel modello formale è sicuramente il *ProVerif*[2]. I sostenitori del modello formale affermano che è molto conveniente ignorare i dettagli di una funzione crittografica e lavorare con una descrizione più di alto livello di questa, che non includa dettagli riguardanti la funzione crittografica. I sostenitori del modello crittografico computazionale invece, affermano che la visione del modello formale non è molto realistica e che le funzioni crittografiche non devono essere viste come espressioni formali ma come algoritmi deterministici o probabilistici. Inoltre i sostenitori del modello formale, trattando le primitive crittografiche come dei simboli impenetrabili, assumono implicitamente che siano corrette e inviolabili, cosa non esatta. D'altra parte il modello formale ha molti vantaggi come quello precedentemente accennato che riguarda la semplicità con cui vengono costruiti strumenti automatici per la verifica di protocolli in questo modello. Sembrerebbe quindi che esista un divario molto ampio fra i due modelli. In effetti così è, ma non sono mancati i tentativi di unificare i due modelli, o comunque di cercare una linea di collegamento che li congiunga. In [AR07] gli autori cercano per la prima volta di porre le basi per iniziare a collegare questi due modelli. In particolare il principale risultato di questo lavoro afferma che: se due espressioni nel modello formale sono equivalenti, una volta dotate di un'opportuna semantica probabilistica, vengono mappati in *ensemble* computazionalmente indistinguibili; quindi, sotto forti ma accettabili ipotesi[3], l'equivalenza formale implica l'indistinguibilità computazionale. È quindi lecito affermare che un attaccante per il modello computazionale non è più potente di un attaccante nel modello formale. Questo risultato ha dato un'ulteriore spinta ai sostenitori del modello formale che potevano così dimostrare risultati nel modello formale, con tutti i benefici che questo comportava, e estendere questo risultato al modello computazionale senza troppi problemi. Un altra strada, invece, è quella che prevede di lavorare direttamente nel modello computazionale senza preoccuparsi di rispettare le forti ipotesi che erano state usate per dimostrare il risultato raggiunto in [AR07]. Rispetto al passato oggi giorno la crittografia non è utilizzata solo in ambiente militare, i suoi

[2]Per informazioni più dettagliate su questo tool si visiti il seguente sito web: `http://www.proverif.ens.fr/`

[3]Un'importante ipotesi usata nella dimostrazione del risultato in questione è che non devono esserci cicli crittografici, ovvero si considerano solo schemi crittografici in cui una chiave k, non è mai *cifrata* attraverso k stessa.

campi di utilizzo infatti, si estendono a molti aspetti della vita quotidiana. Questo fatto ha comportato anche un'estensione dei possibili utilizzi della crittografia. Se infatti, un tempo l'unico scopo che si proponeva la crittografia era quello di garantire la segretezza oggi giorno deve poter fornire molte altre garanzie fra le quali: autenticazione e integrità dei messaggi scambiati fra due parti. Ecco quindi, che la nascita di queste esigenze ha portato allo studio di schemi crittografici come per esempio *message authentication code* oppure schemi crittografici asimmetrici. Quando si parla di segretezza è importante come gia accennato dare prima una definizione di cosa si intende con questo termine. Se per esempio si intende che nessun attaccante possa mai venire a conoscenza della chiave allora si ottiene qualcosa che non è quello che si vorrebbe. La segretezza riguarda un messaggio, la chiave è solo un mezzo che si utilizza per ottenere questo fine. Se invece si intende che un attaccante non riesca mai a scoprire il messaggio si è alla ricerca di una chimera. Innanzi tutto infatti, come si vedrà nel proseguo, dato abbastanza tempo a disposizione si può sempre riuscire ad scoprire con *certezza* il mesasggio; inoltre esiste sempre la possiblità che un attaccante riesca ad indovinare il messaggio semplicemente *tirando ad indovinare*. Se infine si intende che ogni attaccante con determinate caratteristiche riesca difficilmente ad indovinare il messaggio cifrato, allora esiste la possibilità di ottenere schemi che rispettano questo tipo di definizione. Una tenica molto utilizzata per provare dei risultati nell'ambito dell crittografia computazionale è la cosidetta tecnica per riduzione. Le dimostrazioni di sicurezza basate su questa tecnica consistono nel mostrare che se esiste un aversario chè può vincere con una probabilità significativa e in un tempo ragionevole allora anche un problema ben definito può essere risolto con una probabilità significativa e in un tempo ragionevole. Ovviamente si cerca di ridurre lo schema crittografico ad un problema che si sa bene essere difficile da risolvere.

Capitolo 1

Il Modello Computazionale

In questo capitolo si cercherà di descrivere le principali caratteristiche del modello computazionale. Saranno resi evidenti alcuni legami che esistono fra la crittografia, la teoria della calcolabilità e alcune nozioni di statistica e probabilità. Sono questi infatti i cardini su cui poggia la crittografia computazionale.
Si cercherà sempre di dare delle definizioni rigorose e il più possibile non ambigue. Si tenterà sempre, inoltre, di fornire delle dimostrazioni delle affermazioni che si fanno; è questo infatti il giusto modo di procedere. Non è raro infatti, trovare esempi di schemi crittografici che sono stati ritenuti validi solo sulla base di argomentazioni approssimative e non formali e che non essendo stati dimostrati *matematicamente* validi si sono poi rivelati tutt'altro che affidabili[1].

1.1 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 capacità di calcolo potenzialmente infinita o un tempo illimitato a disposizione.
Sebbene siano stati ideati cifrari sicuri anche rispetto ad avversari non limitati[2], 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, si può rappresen-

[1] Il cifrario di Vigenerè ritenuto indecifrabile per moltissimi anni si può infatti violare facilmente con tecniche di tipo statistico.

[2] Il cifrario *one-time pad* è il tipico esempio di cifrario perfettamente sicuro o teoricamente sicuro.

tare questo come un generico algoritmo appartenente ad una particolare classe di complessità computazionale[3].
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 uno schema crittografico.
È logico quindi 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[4].

- 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 trattati come sequenze di bit e non come espressioni *formali*; l'avversario, nel modello computazionale, può effettuare qualsiasi operazione su un messaggio. 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 effettuare una scelta casuale fra il valore 0 e il valore 1 per n volte. 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,

[3]Un avversario infatti è una macchina di Turing che esegue un algoritmo.

[4]Il principio di Kerchoffs (famoso crittografo olandese, 19 Gennaio 1835 - 9 Agosto 1903) afferma che l'algoritmo di cifratura non deve essere segreto e deve poter cadere nelle mani del nemico senza inconvenienti.

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 casuali e di commettere errori (anche se con probabilità limitata). 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 dove è stato associato al valore 0 la croce e al valore 1 la testa (o anche viceversa ovviamente). Qualora la macchina dovesse effettuare una scelta casuale, non dovrà far altro che prendere 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[5] gli schemi crittografici rispetto, principalmente, a questo tipo di avversario. 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*.

1.2 Funzioni Trascurabili e non . . .

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

Definizione 1.1 *Funzione Trascurabile. 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:

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

Definizione 1.2 *Funzione Distinguibile. 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 $\forall n > C$: $\mu(n) > \frac{1}{p(n)}$.*

Per esempio la funzione $n \mapsto 2^{-\sqrt{n}}$ è una funzione trascurabile, mentre la funzione $n \mapsto \frac{1}{n^2}$ 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 1.1 *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 così remota, che una tale preoccupazione è da ritenersi ingiustificata. Finora abbiamo parlato

[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.

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 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.

1.3 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 1.3 *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, tutte con lo stesso tipo di distribuzione.*

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.

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 distribuzioni 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 1.4 *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 probabilità 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).

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

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

Non è raro, nell'ambito scientifico in particolare, basarsi sul concetto generale di indistinguibilità al fine di creare nuove classi di equivalenza di oggetti.

> *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].*

1.4 Pseudocasualità e Generatori Pseudocasuali

Argomento centrale di questa sezione è il concetto di *pseudocasualità* 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. 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 pseudocasualità è un caso particolare di indistinguibilità, infatti una distribuzione è *pseudocasuale* se nessuna procedura efficiente, può distinguerla dalla distribuzione uniforme.

Definizione 1.5 *Pseudocasualità. L'ensemble* $X = \{X_n\}_{n\in\mathbb{N}}$ *è detto pseudocasuale 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 pseudocasuale.

Definizione 1.6 *Generatore Pseudocasuale. 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 pseudocasuale se e solo se valgono le seguenti condizioni:*

- *Espansione:* $\forall n : l(n) > n$
- *Pseudocasualità:* $\forall D \in BPP, \exists \mu$ *trascurabile tale che*

$$|Pr[D(r) = 1] - Pr[D(G(s)) = 1]|$$

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 pseudocasuale. Il suo output, infatti, non è distinguibile dalla distribuzione effettivamente uniforme.

È importante però notare, che la distribuzione di stringhe in output di un generatore pseudocasuale è fortemente differente dalla distribuzione effettivamente casuale. Per rendere più chiara questa distinzione procederemo con un importante esempio. Supponiamo di avere un generatore pseudocasuale 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 casuale 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}.

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

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 pseudocasuale. 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* casuale, e da quello generare una variabile pseudocasuale molto più lunga. Si intuisce da questo la grandissima importanza che hanno i generatori pseudocasuali 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 pseudocasuale appare, ad un distinguitore efficiente D, come una stringa casuale, D non ottiene un vantaggio sensibile nel passaggio dal vero one-time pad al one-time pad che usa una chiave pseudocasuale. In generale i generatori pseudocasuali sono molto utili in crittografia per creare schemi crittografici simmetrici.

1.5 Dimostrazioni Basate su Games

In questo paragrafo si cercherà di spiegare cosa siano nell'ambito della crittografia i *games*[11] e come siano strutturate la maggior parte delle dimostrazioni che utilizzano sequenze di games.
Si possono trovare approfondimenti riguardo a questi concetti nel lavoro di Shoup [Sho]. Quella basata sul concetto di game è una tecnica[12] molto utilizzata per provare la sicurezza di primitive crittografiche o di protocolli crittografici. Questi games sono giocati da un' ipotetica entità maligna, l'attaccante, e da un'ipotetica parte benigna di solito chiamato sfidante[13]. È difficile dare una definizione formale di game; infatti il concetto di game cambia, sebbene in maniera non considerevole, da situazione a situazione, da ambiente ad ambiente e da dimostrazione a dimostrazione (sia che queste siano fatte manualmente sia che queste siano automatiche

[10] Con la notazione $s \xleftarrow{R} O$, si intende la scelta dell'elemento $s \in O$ in maniera casuale.
[11] Che intenderemo letteralmente come giochi.
[12] Game hopping technique.
[13] Perchè *sfida* l'attaccante a vincere questo gioco.

e quindi dipendenti dal framework in cui vengono costruite). Intuitivamente però, i game possono essere immaginati come un insieme di azioni, modellate in una particolare algebra di processi, che servono a specificare il comportamento dei partecipanti al gioco, ovvero le entità che partecipano come *principals* al protocollo crittografico.
Il lettore troverà utile pensarli, almeno nell'ambito di questa tesi, come insiemi di processi che, fra le altre cose, forniscono un'interfaccia all'attaccante attraverso degli *oracoli* che possono resitituire dei valori all'attaccante. Questi oracoli, prima di ritornare il valore, possono effettuare calcoli, dichiarare ed utilizzare variabili che non saranno visibili all'esterno.
Si deve pensare agli oracoli come delle scatole nere inaccessibili dall'esterno. Questi oracoli una volta interrogati forniscono una risposta, e questo è il massimo livello di interazione che dall'esterno si può avere con queste entità.

Il primo passo che le dimostrazioni di sicurezza basate su sequenze di game fanno, è quello di modellare il protocollo crittografico reale in un game iniziale G_0. In G_0 esiste la probabilità non nulla che un evento *negativo* possa accadere (immaginiamolo come una sorta di vittoria da parte dell'attaccante). Ora, in generale, si procede effettuando delle modifiche al game G_i ottenendo un game G_{i+1} tale che G_i e G_{i+1} siano computazionalmente indistinguibili. Le modifiche che si effettuano fra un game e un altro devono introdurre delle differenze computazionalmente irrilevanti. Queste modifiche possono essere viste come regole di riscrittura delle distribuzioni di probabilità delle variabili in gioco nei game. Se per esempio in un game G_i un processo ha a che fare con un variabile casuale, e nel game G_{i+1} questa viene sostituita con una variabile che invece è pseudocasuale, non vengono introdotte modifiche computazionalmente rilevanti e quindi il passaggio è lecito, visto che i due game non sono distinguibili se non con probabilità trascurabile. Alla fine si arriva ad un game G_f in cui *l'evento* non può accadere. Se l'evento non può accadere, l'attaccante non ha possibilità di vincere. Se, quindi, nel game finale l'attaccante non ha possibilità di vincere e il game finale è computazionalmente indistinguibile dal penultimo, questo lo è dal terzultimo e così via fino a G_0, allora il game finale è computazionalmente indistinguibile dal primo[14]. Adesso, quindi, se nel game iniziale esiste la possibilità che un evento avvenga, e nel game finale no, si può 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 attaccante riesce a distinguere un game dal successivo all'interno della sequenza.

È importante dire che, anche nel modello formale si possono utilizzare le sequenze di game, la differenza è che due game successivi non sono computazionalmente indistinguibili ma sono perfettamente indistinguibili. In particolare quello

[14]Ricordiamo infatti che la somma di due probabilità trascurabili rimane trascurabile.

che si vuole sottolineare è che: se in una sequenza di game G_b e G_a sono l'uno il successore dell'altro, allora i due game devono appartenere ad una stessa classe di equivalenza indotta dalla particolare relazione di equivalenza che il modello in cui si sta costruendo la catena di games sfrutta. Nel modello formale questa relazione sarà l'indistinguibilità perfetta (meglio nota come equivalenza osservazionale) mentre in quello computazionale sarà l'indistinguibilità computazionale.

Capitolo 2

CryptoVerif

CryptoVerif è un *dimostratore* automatico di sviluppo molto recente che lavora direttamente nel modello computazionale.
CryptoVerif è stato scritto da Bruno Blanchet[1] in Ocaml. Si possono trovare più informazioni nella home page di Blanchet[2]. Questo tool è utilizzato per dimostrare proprietà di segretezza e autenticazione. Queste prove si basano sulla tecnica delle sequenze di game. Il tool è liberamente scaricabile[3] sotto la licenza CeCill[4].
CryptVerif è stato già utilizzato per dimostrare la correttezza di alcuni protocolli crittografici, come per esempio: FDH [BP06], Kerberos [BJST08].
Scopo di questo capitolo è quello di descrivere in modo non troppo formale[5] un sottinsieme del linguaggio che CryptoVerif implementa, in modo da poter leggere il capitolo successivo con le conoscenze necessarie. Alla fine del capitolo verrà descritta brevemente l'implementazione dello schema di firma FDH nel linguaggiodi CryptoVerif.

[1] Ricercatore al LIENS (Computer Science Laboratory of Ecole Normale Supérieure)

[2] `http://www.di.ens.fr/~blanchet/index-eng.html`

[3] `http://www.cryptoverif.ens.fr/cryptoverif.html`

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

[5] Nel senso che la semantica formale non sarà trattata rigorosamente. Il lettore interessato all'argomento può comunque trovare più informazioni in [BJST08]

2.1 La sintassi

Un tipico file di input per CryptoVerif ha la seguente forma:

$$< decalaration >^{*} \textbf{ process } < odef >.$$

Di solito quindi, la parte iniziale dell'input è costituita da una serie di dichiarazioni, una serie di definizioni di primitive crittografiche e, successivamente alla parola chiave **process**, si trova la definizione di un oracolo che descrive il protocollo di cui si vogliono dimostrare alcune proprietà. Gli oracoli sono definiti in un calcolo di processi. In questo calcolo i termini rappresentano computazioni su stringhe di bit. I termini del calcolo sono distinti in due categorie: semplici e non. La sintassi dei termini semplici è descritta dalla seguente grammatica:

```
<simpleterm> ::= <ident>
                 | <ident> (seq <simpleterm> )
                 | (seq <simpleterm> )
                 | <ident> [seq <simpleterm> ]
                 | <simpleterm> = <simpleterm>
                 | <simpleterm> <> <simpleterm>
                 | <simpleterm> || <simpleterm>
                 | <simpleterm> && <simpleterm>
```

Dove

- $[M]$ significa che M è opzionale.
- seq<X> è una sequenza (eventualmente vuota) di X.

È interessante la terza regola che definisce una tupla di termini semplici. Ovvero partendo dai termini $M_1, ..., M_n$ si costruisce la tupla $(M_1, ...M_n)$ che corrisponde alla concatenazione di $M_1, ..., M_n$. La tupla conterrà anche informazioni sulle singole componenti (tipo, lunghezza) in modo che possano essere successivamente recuperate senza ambiguità. Non si deve confondere la tupla con un semplice termine fra parentesi, che in quel caso, non definiscono un nuovo oggetto ma semplicemente permettono di non avere ambiguità all'interno di espressioni. Abbiamo poi, delle produzioni che descrivono il fatto che un termine semplice può essere un identificatore o un identificatore *applicato* ad altri termini (una funzione). Il calcolo dispone infine di simboli di funzione di default: ovvero uguaglianza, diseguaglianza, disgiunzione e congiunzione. I termini non semplici (che chiameremo termini), contengono dei costrutti più complessi. La sintassi dei termini, è descritta dalla seguente grammatica:

```
<term> ::=
         | <ident> <-R <ident> ; <term>
         | <ident> [: <ident> ] <- <term>
         | let <pattern> = <term> in <term> [else <term> ]
         | if <cond> then <term> else <term>
         | find [[unique]] <tfindbranch> (orfind <tfindbranch>
                          else <term>
```

Il costrutto x<-R D assegna in maniera casuale un valore del dominio D alla variabile x, abbiamo poi il costrutto $x : T$<- M che dichiara una variabile x di tipo T, e assegna a questa il valore ottenuto dalla valutazione del termine M. È interessante analizzare il costrutto *let* $p = M$ *in* M' *else* M''; questo cerca di decomporre il termine M secondo il pattern p. In caso di successo ritorna il valore di M' altrimenti ritorna il valore di M''. Il pattern p può essere una variabile, un simbolo di funzione (eventualmente applicato ai suoi argomenti), una tupla di valori, oppure un'ugualiglianza con un altro termine. Possiamo vedere ciò dalla grammatica che descrive la sintassi di un pattern:

```
<pattern> ::= <ident>
              | <vartype>
              | <ident> (seq <pattern> )
              | (seq <pattern> )
              | = <term>
```

Tralasciando l'ovvio costrutto condizionale analizziamo il costrutto definito dall'ultima produzione che ha come testa il non terminale < *term* >. Questo costrutto, che è probabilmente il maggior apporto fornito da CryptoVerif rispetto ad altri calcoli che lavorano invece su liste, permette di effettuare un lookup su un array. In particolare un < $tfindbranch$ > è descritto dalla seguente produzione:

```
<tfindbranch> ::= seq(<identbound> suchthat <cond> then <term>)
```

In particolare un < *identbound* > è una diseguaglianza fra una variabile e un parametro, per esempio: i <= N dove i è una variabile e N è un parametro; questa diseguaglianza permette di ricercare in un array un indice i minore uguale di N tale per cui valga una certa condizione specificata da <cond>, se questo indice viene trovato allora viene ritornato il valore di < *term* >. Il costrutto find però, può in generale avere vari indici che scorrono vari array in cui si testano diverse condizioni; in generale quindi avremo: $find\ u_{j1}$ <= n_{j1} *suchthat* $cond_j$ *then* M_j,...,$orfind\ u_{ji}$ <= n_{ji} *suchthat* $cond_j$ *then* M_j, ..., $orfind\ u_{jm}$ <= n_{jm} *suchthat* $cond_j$ *then* M_j *else* M. Se nessuna condizione si verifica viene ritornato il valore M, altrimenti si possono avere due casi: o si verifica una sola delle condizioni e allora viene ritornato il valore del termine relativo a quella condizione, oppure

più di una condizione si verifica e in questo caso viene scelto in maniera casuale uno dei termini relativi alle condizioni vere e viene ritornato quello. Il costrutto *find* esiste anche in relazione agli oracoli e quindi verrà ulteriormente spiegato in seguito.

Il calcolo di processi distingue fra:

- definizione di un oracolo.
- corpo di un oracolo.

La grammatica che descrive la sintassi della definizione degli oracoli è la seguente:

```
<odef> ::= <ident>
         | (<odef>)
         | 0
         | <odef> | <odef>
         | foreach <ident> <= <ident> do <odef>
         | <ident> (seq <pattern> ) := <obody>
```

È interessante analizzare il costrutto *foreach* che permette di creare un numero limitato di oracoli identici.
Per esempio:

```
  foreach iS <= qS do OS():=P
```

crea *qS* copie dell'oracolo *OS* il cui corpo è definito da *P*. *qS* è un valore che deve essere dichiarato mediante la parola chiave *param*, inserendo nella parte iniziale del file di input la seguente dichiarazione:

```
param qS.
```

qS non è meglio specificato e non assume un valore definito, l'unica cosa che è data sapere è che è un valore polinomiale nel parametro di sicurezza. Se cosi non fosse sarebbe come mettere a disposizione dell'attaccante più mezzi di quanti ne possa avere nel modello computazionale, mentre sappiamo bene che le risorse dell'attaccante sono, nel modello computazionale, limitate da una funzione polinomiale nel parametro di sicurezza. La grammatica che descrive la sintassi del corpo degli oracoli è la seguente:

```
obody ::= <ident>
          | ( <obody> )
          | end
          | event <ident> [(seq <term> )] [; <obody> ]
```

```
| <ident> <-R <ident> [; <obody> ]
| <ident> [: <ident> ] <- <term> [; <obody> ]
| let <pattern> = <term> [in <obody> [else <obody> ]]
| if <cond> then <obody> [else <obody> ]
| find [[unique]] <findbranch> (orfind <findbranch> )*
          [else <obody> ]
| return(seq <term> )[; <odef> ]
```

Come si può vedere nel corpo di un oracolo si possono assegnare valori a variabili in maniera casuale con il costrutto fornito dalla quinta produzione, o si può anche assegnare un valore che è risultato delle computazioni di un altro oracolo come nella sesta produzione. Un oracolo ritorna il valore con un *return* e successivamente può chiamare un altro oracolo come si dall'ultima produzione. La parola chiave *end* serve per far terminare la definizione di un oracolo. Attraverso l'istruzione condizionale, all'interno del corpo di un oracolo, possiamo chiamare altri oracoli a seconda del valore di verità della condizione verificata. Attraverso il costrutto *let* è possibile cercare di decomporre il valore del termine in un pattern specificato e, a seconda che questa decomposizione abbia o meno successo, è possibile chiamare un oracolo o un altro. In particolare se il pattern è una semplice variabile l'effetto, da un punto di vista pratico, è un assegnamento semplice che ha sempre successo, e sfocia in una dicharazione e inizializzazione di una variabile il cui scope è l'oraclo che si trova dopo la parola chiave *in*; se il pattern è una variabile quindi il ramo *else* non viene mai eseguito. La grammatica che descrive la sintassi dei pattern è stata già trattata precedentemente e non verrà ripetuta.
È fondamentale specificare a CryptoVerif, nella parte delle dichiarazioni (quindi all'inizio), le proprietà che deve cercare di dimostrare. Questo è fatto attraverso la parola chiave *query*, la cui sintassi è descritta dalla seguente grammatica:

```
<query> ::= secret <ident>
     | secret1 <ident>
     | [seq <vartype> ;] <event> <event> (&& <event> )* ==> <queryterm>
<queryterm> ::= <queryterm> && <queryterm>
     | <queryterm> || <queryterm>
     | <event>
     | <simpleterm>
<event> ::= [inj:] <ident> [(seq <simpleterm> )]
```

Per esempio attraverso la seguente dichiarazione:

```
query secret1 w.
```

si chiede a CryptoVerif di tentare di dimostrare la proprietà di *one-session secrecy* per l'array *w*. Ovvero si chiede al tool di provare l'indistinguibilità di ogni elemento dell'array *w* da un numero casuale per un singolo test. Mediante la seguente dichiarazione invece:

```
query secret w.
```

si chiede a CryptoVerif di tentare di dimostrare la proprietà di *secrecy* per l'array *w*. Ovvero si chiede al tool di provare che l'indistinguibilità degli elementi di w da numeri casuali valga per un numero di test qualsiasi.

2.2 Un Esempio: FDH

Full Domain Hash è uno schema di firma che segue il paradigma *hash-and-sign*[6]. Quella che a breve seguirà è la descrizione dello schema di firma FDH nel linguaggio di CryptoVerif.
L'input che si fornisce al CryptoVerif è costituito, fondamentalmente, da due parti:

- Un game iniziale G_0, in cui si modella la sicurezza dello schema.
- Alcune equivalenze necessarie a CryptoVerif per effettuare le modifiche ai game.

La descrizione dello schema di firma FDH viene data al CryptoVerif attraverso il seguente game:

$G_0 \equiv$ **foreach iH** $\leq$ **qH do** OH(x: bitstring):= **return**(hash(x)); **|**

Ogen():=r $\stackrel{R}{\leftarrow}$ seed;pk $\leftarrow$ pkgen(r);sk $\leftarrow$ skgen(r);**return**(pk);
(**foreach iS** $\leq$ **qS do** OS(m: bitstring):= **return**(invf(sk, hash(m)));) **|**

OT(m': bitstring, s:D):=
if f(pk, s)=hash(m') **then find u** $\leq$ **qS suchthat** (**defined**(m[u]) $\wedge$ $m' = m[u]$))
then end
else event forge

[6]Questo paradigma vuole che: dato un messaggio *m* se ne ritorni la firma di *hash*(*m*) e non la firma di *m*, dove la funzione hash può essere istanziata con qualsiasi funzione hash collision resistant. Si ottiene così una firma di lunghezza fissa e non dipendente dalla lunghezza del messaggio

Procediamo adesso con una breve spiegazione di questo game. Possiamo vedere come in questo game si forniscano qH copie dell' oracolo OH i quali ritornano l'hash della stringa che gli si fornisce in input ovvero x. Abbiamo poi il processo $Ogen()$ che ritorna al contesto una chiave pubblica dopo aver creato, partendo da un seme casuale, una coppia costituita da una chiave privata e una chiave pubblica. Abbiamo poi qS copie dell'oracolo OS che si occupano di fornire una firma del messaggio che gli viene dato in input. Infine abbiamo un singolo oracolo OT che si occupa di verificare se la firma s è valida per il messaggio m'. In particolare se $hash(m') \neq f(pk, s)$ allora la firma non è valida per il messaggio. Se invece $hash(m') = f(pk, s)$ l'oracolo si occupa di verificare se, per il messaggio m', è stata mai rilasciata una firma dall'oracolo OS, in caso affermativo il processo termina, altrimenti significa che il contesto, ovvero l'attaccante, è stato in grado di forgiare una firma valida per il messaggio, e quindi è avvenuto l'evento *forge*. Notiamo come l'attaccante non sia modellato esplicitamente, infatti non possiamo fare nessuna assunzione sul comprtamento di questo. Possiamo immaginare un attaccante come un altro processo non meglio specificato messo in parallelo con questo game. In gergo l'attaccante viene anche detto anche *contesto*. Per poter trasformare un game in un altro CryptoVerif ha bisogno di alcune equivalenze da poter utilizzare. Le definizioni in CryptoVerif vengono fornite attraverso delle equivalenze che possono essere viste come regole di riscrittura delle distribuzioni di probabilità delle variabili in gioco. Queste regole di riscrittura di un generico elemento L in un generico elemento R possono valere incondizionatamente, oppure possono valere a meno di una certa probabilità. Nel secondo caso la riscrittura di un termine L nell'equivalente R comporta l'introduzione di una differenza non nulla fra i due game.

Prima di descrivere il secondo input di CryptoVerif è necessario fare una piccola digressione riguardo al concetto di funzione *one-way*. Intuitivamente una funzione f è one-way se non può essere invertita facilmente. Cioè se dato $y = f(x)$ è arduo riuscire a trovare un x' tale che $f(x') = y$. Si noti come non sia necessario, per invertire la funzione, trovare x ma è sufficiente trovare un x' qualsiasi tale che $f(x') = y$.
Possiamo ora dare la seguente:

Definizione 2.1 *Funzione One-Way.* $f : \{0,1\}^* \rightarrow \{0,1\}^*$ *è one-way se e solo se* $\forall x \in \{0,1\}^*, \forall A \in PPT$: $|Pr[A(f(x)) \in f^{-1}(f(x))]|$ *è una funzione trascurabile.*

È importante notare come la definizione valga per ogni x del dominio; con questo si vuole sottolineare che invertire f deve essere *sempre* difficile e non per particolari x. È importante affermare ciò perchè tutta la crittografia moderna è fondata sull'ipotesi che esistano le funzioni one-way. Si parla di ipotesi perchè, sebbene la maggior parte degli informatici ne sia convinta, non è stato ancora di-

mostrata l'esistenza di funzioni di questo genere[7]. La dimostrazione dell'esistenza di funzioni one-way comporterebbe, tra l'altro, anche la risoluzione della famosa questione riguardo agli insiemi P e NP[8]. Il fatto che però qualcuno un giorno possa dimostrare che $P \neq NP$ non dimostrerebbe affatto l'esistenza delle funzioni one-way. Infatti una funzione one-way deve, come prima sottolineato, essere non invertibile in modo efficiente, sempre e non solo nel *caso pessimo*.
Siamo ora pronti per vedere come il concetto di funzione one-way viene modellato in CryptoVerif.

Si tratta di dare una definzione di one-way per mezzo di un'equivalenza fra games. La definizione è la seguente:
equiv
foreach ik ≤ **nk do** r $\stackrel{R}{\leftarrow}$ seed; (Opk():= **return**(r);) **|**

foreach iF ≤ **nF do** x $\stackrel{R}{\leftarrow}$ D; (Oy():= **return**(f(pkgen(r), x)); **|**

foreach i1 ≤ **n1 do** Oeq(x': D):= **return**(x'=x); **|**

Ox():=**return**(x);
$\approx_{pow}$
foreach ik ≤ **nk do** r $\stackrel{R}{\leftarrow}$ seed; (Opk():= **return**(r);) **|**

foreach iF ≤ **nF do** x $\stackrel{R}{\leftarrow}$ D; (Oy():= **return**(f'(pkgen'(r), x)); **|**

foreach i1 ≤ **n1 do** Oeq(x': D):= **if defined**(k) **then return**(x'=x);
else return(false); **|**

Ox():=k ← mark;**return**(x);

Questa equivalenza fra game cattura e definisce il concetto di funzione one-way. La funzione f, infatti, è one-way se e solo se vale l'equivalenza *nNXXX*, e quindi il game a sinistra e quello a destra del simbolo $\approx_{pow}$ sono indistinguibili. Supponiamo, infatti, che la funzione f non sia one-way, esisterà dunque un avversario efficiente A che può invertire f. A sarà dunque in grado di distinguere i due membri dell'equivalenza $L \approx_{pow} R$. Potendo invertire f, A sarà in grado di osser-

[7] Questo non toglie che esistono funzioni che si avvicinano all'idea che abbiamo di funzioni one-way, per esempio SHA1, o MD5.

[8] P è l'insieme dei problemi risolvibili nel caso pessimo in un numero di passi polinomiale nell'input. NP invece, è l'insieme dei problemi per cui dato un certo valore, si può verificare in tempo polinomiale se questo è o meno soluzione del problema (ovvero è l'insieme dei problemi con certificazione polinomiale.). Non è ancora noto se questi insiemi siano o meno lo stesso insieme.

vare comportamenti diversi fra i game a seconda che A chiami Oeq() in L o in R. Supponiamo, dunque, che A chiami Oy() ottenendo l'immagine di un valore mediante f, sia questa y; poichè f non è one-way, A inverte la funzione ottenendo un x' tale che $f(x') = y$. Adesso A non deve far altro che richiamare Oeq passando a questo come argomento x'. Se l'Oeq richiamato appartiene al primo membro dell'equivalenza, allora Oeq() ritornerà sempre true, mentre se appartiene al secondo ritornerà sempre false. In questo modo A osserva dei comportamenti diversi fra i due game, e quindi può distinguerli. Supponiamo invece che f sia effettivamente one-way. Allora non esisterà nessun attaccante efficiente che riesca ad invertire f. L'unico modo che ha quindi un attaccante A per invertire la funzione, è chiamare Ox() e ottenere così una preimmagine valida. Se però l'attaccante chiama Ox() allora L e R sono effettivamente indistinguibili (a meno di probabilità trascurabili). Notiamo infatti nella definizione dell'equivalenza che Ox() è definito in maniera diversa in L ed R. In L, Ox() si occupa semplicemente di ritornare il valore del dominio x, tale che $f(x) = y$. In R invece Ox() prima di ritornare x imposta al valore *mark* la variabile k. Ora, l'Oeq si comporterà in maniera diversa a seconda che appartenga ad L o ad R. Se infatti l'Oeq() richiamato appartiene ad L allora Oeq() si proccuperà semplicemente di ritornare il valore booleano dell'espressione $x' = x$. Se invece, l'Oeq() richiamato appartiene ad L allora Oeq() si comporterà in maniera differente a seconda che la variabile k abbia o meno un valore (ovvero sia stata o meno definita da una precedente chiamata a Ox). Se quindi k è definito Oeq si comporterà esattamente come si sarebbe comportato l'Oeq di L. Se invece k non è stato definito allora siginifica che l'attaccante non ha richiamato Ox, quindi è lecito supporre che non sia riuscito ad invertire f e quindi Oeq ritorna false[9]. In questo modo i due game sono effettivamente indistinguibili, perchè non presentano differrenze di comportamento che A possa notare, e quindi ai suoi occhi sono indistinguibili. Quindi L e R sono è indistinguibile *se e solo se* f è one-way; quindi la precedente equivalenza è una definizione ben posta di funzione one-way. Un'ulteriore definizione viene fornita al CryptoVerif, quella di funzione hash. Questa definizione viene fornita al tool attraverso il seguente codice:

equiv
foreach iH $\leq$ **nH do** OH(x:hashinput) := **return**(hash(x));
$\approx_{pow}$
foreach iH $\leq$ **nH do** OH(x:hashinput) := **find u** $\leq$ **nH suchthat** (**defined**(x[u],r[u]) $\wedge$ x= x[u])) **then** return(r[u])

[9] È importante notare che l'attaccante ha comunque la possibilità di indovinare x anche senza richiamare Ox ma questa è una probabilità trascurabile. Di questa probabilità viene comunque tenuto conto nell'equivalenza. L'equivalenza infatti, vale a meno di una certa probabilità *pow* (trascurabile in questo caso).

else r $\stackrel{R}{\leftarrow}$ hashoutput;; **return**(r);.

La funzione hash è intesa implementata nel modello dell'oracolo random: se la funzione non è mai stata richiamata su un particolare valore x_0 allora viene ritornato un valore casuale, altrimenti viene ritornato lo stesso valore che era stato ritornato precedentemente. Questo viene fatto salvando il valore ritornato per un particolare input in un array e poi, al momento della chiamata, si effettua un look up nell'array. Infine abbiamo una semplice teoria equazionale per descrivere alcune proprietà delle funzioni in gioco.

Per esempio:

```
forall r:seed, x:D, x':D; (x' = invf(skgen(r),x)) = (f(pkgen(r),x') = x).
forall r:seed, x:D; f(pkgen(r), invf(skgen(r), x)) = x.
forall r:seed, x:D; invf(skgen(r), f(pkgen(r), x)) = x.
```

Le precedenti regole servono a definire $invf$ come funzione inversa di f e viceversa. Mentre le seguenti:

```
forall k:skey, x:D, x':D; (invf(k,x) = invf(k,x')) = (x = x').
forall k:pkey, x:D, x':D; (f(k,x) = f(k,x')) = (x = x').
forall k:pkey, x:D, x':D; (f'(k,x) = f'(k,x')) = (x = x).
```

modellano l'iniettività delle funzioni $invf$, f', f[10]. Quella che segue è una spiegazione delle parti più importanti dell'output di CryptoVerif quanto questo viene eseguito sull'input prima descritto. Nell'appendice XX si può trovare l'input completo, mentre in [BP06] possiamo trovare una spiegazione delle fasi più importanti dell'output di CryptoVerif e infine [BP] troviamo l'ouput completo di CryptoVerif sull'input di FDH.

[10]Non serve ai fini della dimostrazione modellare l'iniettività della funzione $invf'$.

Capitolo 3

Risultati Raggiunti

Scopo principale di questo lavoro è la dimostrazione mediante CryptoVerif di un risultato classico della crittografia computazionale.

3.1 Teorema

Il teorema è il seguente:

Teorema 3.1 *Se esiste un generatore pseudocasuale G' con fattore d'espansione $l'(n) = n + 1$ allora per ogni polinomio $p(n) > n$ esiste un generatore pseudocasuale G con coefficiente d'espansione $l(n) = p(n)$.*

Si può trovare una dimostrazione *classica* di questo teorema in [KL07]. La dimostrazione classica procede....

Per

Capitolo 4

Conclusioni

Bibliografia

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.

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

[BAN90] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[BJST08] Bruno Blanchet, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 87–99, Tokyo, Japan, March 2008. ACM.

[BP] Bruno Blanchet and David Pointcheval. sequences of games of fdh. `http://www.cryptoverif.ens.fr/FDH/fdh.pdf`.

[BP06] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In Cynthia Dwork, editor, *CRYPTO'06*, volume 4117 of *Lecture Notes on Computer Science*, pages 537–554, Santa Barbara, CA, August 2006. Springer Verlag.

[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.

[Kem87] Richard A. Kemmerer. Analyzing encryption protocols using formal verification authentication schemes. In *CRYPTO*, pages 289–305, 1987.

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

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949.

[Sho] Victor Shoup. Sequences of games: A tool for taming complexity in security proofs. `http://eprint.iacr.org/2004/332.pdf`.