Introduzione

Da quando negli anni anni '80 ha cessato di essere un'arte per assurgere al rango di scienza, la crittografia moderna si è proposta innanzi tutto di fornire definizioni rigorose dei concetti con cui aveva e 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 *incondizionatamente* dei risultati, 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; ogni dimostrazione è infatti del tipo: *Se X allora Y* dove X contiene un quantificatore esistenziale (e.g. se esiste una funzione one-way allora esiste un generatore pseudocasuale con fattore d'espansione polinomiale). Quindi, semplicemente, nel caso in cui le assunzioni dovessero essere confutate si renderebbero poco interessanti le dimostrazioni[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 anche queste ampiamente accettate 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 ampiamen-

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

te trattato in [AG99, BAN90, Kem87, Pau98]. Il secondo modello trova le basi in lavori di 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 di più alto livello di questa e 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 ma questa non è del tutto esatto. 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 le due teorie, 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, con tutti i benefici che il modello formale comportava, e estendere questo risultato al modello computazionale senza troppi problemi. Un'altra strada, invece, è quella che prevede di lavora-

[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.

re 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 campi di utilizzo 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 già accennato, dare prima una definizione di cosa si intenda 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, infatti, 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 decifrare il messaggio si è alla ricerca di una chimera, perché come si vedrà nel proseguo, posto che si abbia sufficiente tempo a disposizione si può sempre riuscire a decifrare con *certezza* il messaggio; 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 della crittografia computazionale è la cosidetta "tecnica per riduzione". Le dimostrazioni di sicurezza basate su questa tecnica consistono nel mostrare che se esiste un aversario che 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

[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.

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 rappresentare 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

[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.

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, 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 *PPT* (i.e. *Probabilistic 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 $\forall n > C$: $\mu(n) < \frac{1}{p(n)}$.*

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

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 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.*

[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.

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 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 «l'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].

[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.

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). 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 Game

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

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

ad ambiente e da dimostrazione a dimostrazione (sia che queste siano fatte manualmente, sia che queste siano automatiche e quindi dipendenti dal framework in cui vengono costruite). Intuitivamente però, i giochi 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 restituire il valore, possono effettuare calcoli, dichiarare ed utilizzare variabili che non saranno visibili all'esterno.
Si deve pensare agli oracoli come a 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 giochi fanno, è quello di modellare il protocollo crittografico reale in un gioco 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 gioco G_i ottenendo un gioco G_{i+1} tale che G_i e G_{i+1} siano computazionalmente indistinguibili. Le modifiche che si effettuano fra un gioco e un altro devono introdurre delle differenze computazionalmente irrilevanti. Queste modifiche possono essere viste come regole di riscrittura delle distribuzioni di probabilità delle variabili utilizzate nei giochi. Se, per esempio, in un gioco G_i un processo ha a che fare con un variabile casuale, e nel gioco 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 giochi non sono distinguibili se non con probabilità trascurabile[14]. Alla fine si arriva ad un gioco 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 gioco finale l'attaccante non ha possibilità di vincere e il gioco finale è computazionalmente indistinguibile dal penultimo, questo lo è dal terzultimo e così via fino a G_0, allora il gioco finale è computazionalmente indistinguibile dal primo[15]. Adesso, quindi, se nel gioco iniziale esiste la possibilità che un evento

[14]In teoria si può ammettere anche un passaggio da gioco a gioco che valga con probabilità p non trascurabile, ovvio che poi si giungerà ad un gioco finale che sarà distinguibile da quello iniziale per una probabilità non trascurabile di valore almeno p. Quindi questo tipo di passaggio non è utile.

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

avvenga, e nel gioco finale no, si può dare un limite superiore alla probabilità che l'evento avvenga nel gioco iniziale. Questo limite è la somma di tutte le probabilità con cui un attaccante riesce a distinguere un gioco dal successivo all'interno della sequenza.

È importante dire che anche nel modello formale si possono utilizzare le sequenze di gioco. La differenza è che due giochi successivi non sono computazionalmente indistinguibili, ma sono perfettamente indistinguibili. In particolare quello che si vuole sottolineare è che se in una sequenza di giochi G_b e G_a sono l'uno il successore dell'altro, allora i due giochi devono appartenere ad una stessa classe di equivalenza che è indotta dalla particolare relazione di equivalenza che il modello in cui si sta costruendo la catena di giochi 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 abbastanza recente[1] che lavora direttamente nel modello computazionale.
CryptoVerif è stato scritto da Bruno Blanchet[2] nel linguaggio di programmazione OCaml. Si possono trovare più informazioni riguardo al tool nella home page di Blanchet[3]. Questo tool è utilizzato di solito per dimostrare proprietà di segretezza e autenticazione. Le prove in questione si basano sulla tecnica delle sequenze di giochi. Il tool è liberamente scaricabile[4] sotto la licenza CeCill[5]. CryptVerif è stato già utilizzato per dimostrare la correttezza di alcuni protocolli crittografici o schemi di firma, come per esempio: FDH [BP06], Kerberos [BJST08].
Scopo di questo capitolo è quello di descrivere, in modo non troppo formale[6], un sottoinsieme 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 linguaggio di CryptoVerif.

[1] La sua prima release stabile risale al 2006.
[2] Ricercatore al LIENS (Computer Science Laboratory of Ecole Normale Supérieure)
[3] `http://www.di.ens.fr/~blanchet/index-eng.html`
[4] `http://www.cryptoverif.ens.fr/cryptoverif.html`
[5] `http://www.cecill.info/licences/`
[6] 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

Di seguito si cercherà di fornire maggiori spiegazioni senza però, per ora, entrare troppo nel dettaglio. Un tipico file di input per CryptoVerif ha la seguente forma:

```
<declaration>* process <odef>.
```

2.1.1 Dichiarazioni

Una dichiarazione (declaration) fornisce a CryptoVerif delle informazioni su come comportarsi nella dimostrazione; spesso è la descrizione di una primitiva crittografica che CryptoVerif può assumere come vera per costruire le sue prove. In generale una dichiarazione può essere:

Simbolo funzionale. La dichiarazione di una funzione si dà insieme ai tipi dei parametri in input (se ce ne sono) e il tipo del valore di ritorno. Per esempio, con il seguente frammento di codice:

```
fun kgen(keyseed):key.
```

si dichiara una funzione `kgen: keyseed`→ `key`. Intuitivamente il simbolo `kgen` rappresenta una funzione per generare chiavi crittografiche a partire semi casuali.

Tipo. L'utente può definire dei propri tipi i cui valori saranno trattati come stringhe di bit. Ovvero i valori dei tipi sono sempre dei sottoinsiemi di $\{0,1\}^*$. Per esempio, con il seguente frammento di codice:

```
type keyseed[fixed].
```

si dichiara un nuovo tipo `keyseed`. La parola chiave `fixed` serve ad informare CryptoVerif che è possibile estrarre valori casuali da quell'insieme. Un'altra opzione molto utilizzata è `large`. Un tipo dichiarato come `large` gode della proprietà che, scegliendo casualmente due valori dal dominio del tipo, la probabilità di ottenere due valori identici è bassa.

Proprietà. È possibile modellare alcune proprietà dei simboli di funzione che si specificano, come per esempio l'iniettività oppure la monotonia. Per esempio, con il seguente frammento di codice:

```
forall x, y: myType; (f(x) = f(y)) = (y = x).
```

definiamo l'iniettività della funzione `f`. Infatti possiamo sostituire il membro a sinistra dell'equazione con quello di destra solo se la funzione `f` è iniettiva.

Regola di Riscrittura. La dichiarazione di una regola di riscrittura consiste in una coppia di espressioni. Ogniqualvolta CrytptoVerif incontra la prima può decidere di sostituirla con la seconda. Per esempio, con il seguente frammento di codice:

```
not(not(true))=false.
not(not(false))=true.
```

informiamo CryptoVerif che ogni qualvolta incontra una doppia negazione del valore `true` può sostituirlo con il valore `false`, e viceversa. Si badi: se si dessero, invece delle precedenti, queste regole di riscrittura:

```
false=not(not(true)).
true=not(not(false)).
```

si andrebbe probabilmente incontro ad un loop infinito durante la fase di elaborazione di CryptoVerif. Infatti nel momento in cui il tool riconoscesse nel codice il valore `true`, procederebbe con una riscrittura di questo nel frammento `not(not(false))` il quale a sua volta sarebbe riscritto in `not(not(not(not(true))))` e cosi via senza poter mai terminare.

Probabilità. In CryptoVerif è possibile dichiarare delle probabilità. Le probabilità non solo possono essere trattate come *oggetti* (numeri) a se stanti, ovvero come dei valori statici, ma possono anche essere utilizzate come funzioni di altri argomenti. Con il seguente codice frammento:

```
proba POW.
```

si dichiara una probabilità `POW`.

Costante. Dichiarando una costante `k` si comunica a CryptoVerif che quel dato, una volta inizializzato, non potrà cambiare valore durante l'esecuzione del protocollo. Per esempio, con il seguente frammento di codice:

```
const mark:bitstring.
```

Si dichiara una costante chiamata `mark` di tipo `bitstring`.

Evento. Un evento permette di rappresentare alcune particolari situazioni nel flusso d'esecuzione del protocollo in esame. Per esempio: si potrebbe pensare di dichiarare un evento *forge* da *sollevare* nel momento in cui un avversario riesca a *forgiare* una firma valida per un messaggio mediante il seguente frammento di codice:

```
event forge.
```

Si osservi ora il seguente codice:

```
find u <= qS suchthat defined(m[u]) && (m' = m[u]) then
    end
else
   event forge.
```

il costrutto `find` verrà analizzato successivamente, per ora è sufficiente sapere che il costrutto `find` ha un ramo che viene eseguito quando una particolare condizione è verificata (il ramo then) e un ramo che viene seguito quando la condizione del ramo then non si verifica. Nel frammento precedente non ci sono dichiarazioni, infatti il frammento serve per dare un esempio di come sollevare un evento. Nel ramo else, dunque, viene sollevato l'evento forge. Infatti, come sarà meglio spiegato nel paragrafo relativo all'esempio FDH, il ramo else rappresenta la situazione in cui un avversario ha fornito una coppia `(m,` σ`)` tale che σ è una firma valida per il messaggio `m` e nessun oracolo ha mai rilasciato tale firma.

Equivalenza. La dichiarazione di un'equivalenza fra giochi permette di dare a CryptoVerif una regola di riscrittura che può valere a meno di una certa probabilità (eventualmente nulla). Affermando che un gioco è equivalente ad un altro a meno di una certa probabilità, si dà la possibilità al tool di sostituire espressioni che compaiono nel primo gioco con le equivalenti che compaiono nel secondo, tenendo comunque conto della probabilità in gioco. Più in generale, quando si dà un'equivalenza fra giochi a meno di una certa probabilità `p` si comunica a CryptoVerif che i giochi in questione non possono essere distinti con una probabilità maggiore di `p`. Ovviamente se la probabilità è nulla allora i giochi non possono essere distinti. Per esempio, con il seguente frammento di codice:

```
equiv
      foreach iK <= nK do r <-R seed; (
          Opk() := return(pkgen(r)) |
          foreach iF <= nF do x <-R D;
           (Oant() := return(invf(skgen(r),x)) |
            Oim() := return(x)
               )
           )
   <=(0)=>
      foreach iK <= nK do r <-R seed; (
         Opk() := return(pkgen(r)) |
         foreach iF <= nF do x <-R D;
          (Oant() := return(x) |
                   Oim() := return(f(pkgen(r), x))
              )
          )
```

si definisce un'equivalenza fra giochi che permette di definire `invf` come funzione inversa di `f`. I giochi in questione sono rispettivamente quello a sinistra e quello a destra dei simboli `<=(0)=>`. Poiché quest'equivalenza deve valere sempre e non a meno di una probabilità, la quantità tra parentesi è zero (fra i due giochi).

Parametro. Un parametro (di cui si sa solo che assumerà valori polinomiali nel parametro di sicurezza) può essere utilizzato per vari scopi. Il più importante è sicuramente effettuare lookup in array che contengono un numero di oggetti pari al parametro. Per esempio, con il seguente codice:

```
param qS.
```

si dichiara un parametro `qS` di cui sappiamo che assumerà valori polinomiali nel parametro di sicurezza.

Oracolo. È possibile dichiarare oracoli aggiuntivi che successivamente possono essere richiamati durante l'esecuzione del protocollo crittografico. Un oracolo[7], in CryptoVerif, non è altro che un processo che può prendere dei dati in input e restituirne in output. Un oracolo può, nel suo corpo, effettuare dei calcoli i quali devono essere considerati invisibili per un ipotetico attaccante. Per esempio, con il seguente frammento di codice:

```
let processS =
  foreach iS <= qS do
  OS(m:bitstring) :=
   return(invf(sk, hash(m))).
```

si dichiara un processo `processS` che non è altro che la replicazione, limitata polinomialmente dal parametro `qS`, dell'oracolo `OS`, il cui corpo è un'istruzione che ritorna un valore.

Query. Mediante un'istruzione di query è possibile istruire il tool su cosa si vuole venga dimostrato. Si potrebbe infatti richiedere al tool di dimostrare l'indistinguibilità di una variabile da un oggetto casuale, provandone così la pseudocasualità. Per esempio, con il seguente frammento di codice:

```
query secret1 x.
```

si chiede a CryptoVerif di cercare di dimostrare che x è indistinguibile da una stringa casuale.

Di solito la maggior parte delle dichiarazioni (o comunque quelle più importanti e utilizzate) vengono fornite a CryptoVerif in un file separato (una sorta di libreria). Grazie a questa possibilità è possibile dare una volta per tutte alcune definizioni molto utili in crittografia (e.g funzione one-way) e, una volta fatto, è possibile utilizzarle in tutti gli schemi crittografici che le utilizzano semplicemente specificando il file di libreria corretto.

[7] Il concetto di oracolo qui non ha molto a che fare con gli oracoli che spesso si trattano in altri ambiti dell'informatica.

2.1.2 Definizione del protocollo

Dopo una serie di dichiarazioni, si procede inserendo la parola chiave `process` e si continua specificando, mediante un oracolo, il protocollo di cui si desiderano dimostare delle proprietà. In CryptoVerif, come già detto, un oracolo non è altro che un processo o un'opportuna composizione di processi. I processi sono specificati mediante un calcolo di processi; questo mette a disposizione vari operatori per comporre in vario modo dei processi di base, al fine di ottenere ancora dei processi. Per esempio, dati due processi A e B è possibile ottenere un altro processo $A|B$ (leggasi A parallelo B) in cui i due processi sono eseguiti in parallelo; oppure è possibile ottenere il processo $A; B$ che è un processo che esegue A e poi esegue B. È possibile inoltre avere un particolare tipo di parallelismo in cui si crea un numero di copie, limitato polinomialmente, di un particolare processo (in questo CryptoVerif si è sicuramente ispirato [MRST06]. Quest'ultimo lavoro si è ispirato a sua volta a [AG99] introducendo però un'innovazione che è appunto la nozione di replicazione limitata polinomialmente). I processi nell'ambito della descrizione dei protocolli crittografici devono essere intesi come le entità che nel mondo reale utilizzano il protocollo stesso. Se per esempio un protocollo prevede che un utente A spedisca un messaggio cifrato ad un utente B si può pensare di modellare sia A che B attraverso due processi separati. Attraverso la grammatica in BNF che segue possiamo descrivere in modo più formale la composizione di oracoli.

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

Dove `seq <X>` è una qualsiasi sequenza finita di `X`. Si noti la terza produzione in cui si specifica che l'oracolo *nullo* (ovvero quello che non fa nulla) è un oracolo valido all'interno del calcolo. O ancora, un oracolo può essere la composizione parallela di più oracoli o la replica limitata da un valore un polinomiale[8] (mediante il `foreach`) di un altro oracolo. Il fatto che un oracolo possa essere composto in maniera sequenziale con un altro oracolo verrà spiegato meglio a breve.

Abbiamo parlato di modi di comporre oracoli per ottenere altri oracoli; non abbiamo però descritto cosa può fare un oracolo, in particolare non si è

[8] Ovviamente polinomiale nel parametro di sicurezza.

descritto come questi siano costruiti. Le operazioni che può fare sono varie, infatti può:

- terminare.
- ritornare un valore al chiamante (che può essere anche l'attaccante).
- sollevare un evento.
- assegnare ad una variabile un valore, il quale può essere anche il risultato restituito da un altro oracolo
- assegnare ad un variabile un valore scelto casualmente da un particolare dominio.
- avere comportamenti diversi a seconda dell'esito di un test.
- ricercare elementi all'interno di un array.

Il corpo di un oracolo può essere descritto in maniera più formale attraverso la seguente grammatica in BNF:

```
<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> ]
```

Innanzi tutto si noti come nella grammatica precedente grazie all'ultima produzione sia possibile ritornare alla grammatica predecedentemente esposta il cui simbolo iniziale era `<odef>`; questo perchè deve essere possibile alla fine dell'implementazione di un oracolo definirne un altro in modo da avere un costrutto per la sequenzializzazione degli oracoli. Infatti, dopo un `return`, mediante il simbolo di sequenzializzazione (il simbolo ";"), è possibile definire un oracolo che verrà eseguito dopo quello precedente. È importante notare il costrutto `let` con il quale è possibile cercare di decomporre il valore a destra del simbolo di uguaglianza in un pattern specificato e, a seconda che questa decomposizione abbia o meno successo, è possibile richiamare un oracolo o un altro. Un pattern può essere:

- una variabile.
- una tupla di valori.

Per esempio, il frammento di codice:

```
let x = A(r) in
return(x)
```

non fa altro che assegnare ad `x` (che può anche non essere dichiarata e il cui tipo viene inferito dal tipo di ritorno di `A`) il valore che la funzione `A` ritorna su input `r`, e poi ritorna il valore di `x`. Si noti che se, come nel caso precedente, il pattern è una variabile, l'effetto, da un punto di vista pratico, è un semplice assegnamento di un valore ad una variabile. Questo assegnamento ha sempre successo, e degenera quindi in una dichiarazione e inizializzazione di una variabile il cui *scope* è l'oracolo che si trova dopo la parola chiave `in`; se il pattern è una variabile quindi, il ramo `else` non viene mai eseguito. Come già accennato è possibile assegnare dei valori scelti in maniera casuale a delle variabili. Se per esempio `x` è una variabile e `D` è un tipo dichiarato con la clausola `fixed`, allora il seguente codice:

```
x <-R D;
```

assegna ad `x` un valore scelto in maniera casuale da `D`. In questo tipo di assegnamento a destra troviamo sempre un dominio di valori dichiarato precedentemente con la clausola `fixed`
L'assegnamento non casuale invece ha la seguente forma:

```
x:Type <- G(r);
```

con il codice precedente si dichiara `x` come variabile di tipo `Type` e si assegna a questa il valore ritornato dalla funzione `G`. Si osservi ora come nel seguente frammento di codice:

```
MyOracle() :=
    r <-R A;
    x:MyType <- f(r);
    return(x);
    (Z | X | Y)
```

venga definito il corpo dell'oracolo `MyOracle`. In `MyOracle` per prima cosa si assegna alla variabile `r` un valore scelto casualmente dall'insieme dei valori del tipo `A`, successivamente si assegna alla variabile `x` il valore resitutito dalla

funzione `f` applicato al valore `r`, infine si ritorna il valore `x`, successivamente viene eseguito un altro processo che è il risultato della composizione in parallelo dei processi `Z`, `X` e `Y`.

Un costrutto fondamentale del linguaggio implementato da CryptoVerif è il costrutto `find`[9] che permette di effettuare un lookup su un array alla ricerca di elementi che soddisfino particolari proprietà. Inoltre, permette di compiere azioni diverse a seconda che questi elementi vengano trovati o meno. Un tipico esempio di utilizzo del costrutto `find` può essere il seguente frammento di codice:

```
let processT =
     OT(m':bitstring, s:D) :=
     if f(pk, s) = hash(m') then
     find i <= qS suchthat defined(m[i]) && (m' = m[i]) then
         end
     else
         event forge.
```

In questo frammento viene definito un processo `processT` come un oracolo `OT` che prende in input degli argomenti e che, quando viene chiamato, effettua un test; se questo è positivo esegue il ramo `then` dell'`if`. Nel ramo `then` viene effettuata una ricerca nell'array `m` mediante il contatore `i`. Se viene trovato un `i` tale per cui l'iesimo elemento dell'array `m` è definito (ovvero gli è stato precedentemente assegnato un valore) e tale per cui l'iesimo elemento dell'array ha un valore uguale a quello fornito in input, allora viene eseguito il ramo `then` del `find` che è semplicemente la terminazione dell'oracolo. Il costrutto `find` permette anche la scansione di diversi array, ovvero si può effettuare una ricerca all'interno di diversi array cercando elementi che verifichino diverse condizioni. Se diverse condizioni vengono verificate e quindi c'è la possibilità che debbano essere eseguiti diversi rami `then` allora CryptoVerif ne sceglie uno casualmente ed esegue quello.

2.2 La Tecnica di Dimostrazione

Come già detto in precedenza CryptoVerif basa le sue prove sulla tecnica delle sequenze di giochi. Questi giochi però differiscono fra loro di probabilità che non sono trattate asintoticamente infatti come spiegato più avanti CryptoVerif si basa sul modello esatto e non asintotico. CryptoVerif parte

[9] Come si afferma anche in [BP06] è probabilmente il maggior apporto fornito da questo tool rispetto ad altri linguaggi che lavorano invece su liste.

da un gioco iniziale fornito dall'utente (in cui un evento negativo può accadere[10]) e mediante equivalenze, che possono o meno introdurre differenze, cerca di modificare un gioco in un altro fino ad arrivare ad un gioco finale che soddisfi le richieste fatte dall'utente. Questa tecnica è basata innanzi tutto su una particolare relazione di cui a breve si darà una definizione. Prima di dare questa definizione è opportuno spiegare prima la notazione che sarà utilizzata da qui in poi.
Di solito indicheremo con la lettera t dei tempi d'esecuzione di processi. Se A e B sono due processi allora con la scrittura $A|B$ indichiamo un processo che è la composizione parallela di A con B. Nella composizione parallela di due processi qusti possono procedere indipendentemente ma possono anche interagire attraverso operazioni di input e output. In questo modo se ha la composisione parallela di A con B allora un'operazione di output di A può fornire l'input di un'operazione di input di B e viceversa, ottenendo quindi una comunicazione fra processi. Con i simboli $C[\cdot]$ e $D[\cdot]$ si indicano dei contesti. Un contesto è un processo con un *buco*, ovvero un processo che ha come argomento un altro processo e va in esecuzione in parallelo a questo. Di solito un contesto è un attaccante che cerca di distinguere un processo che gli viene passato in input. Sia $C[\cdot]$ un contesto e siano P e Q dei processi, allora C distingue i due processi se $C[P]$ (ovvero $C|P$) ha un comportamento diverso da $C[Q]$. Infatti se quando C interagisce con il processo P osserva un comportamento diverso da quello che osserva quando interagisce con Q allora C può effettivamente distinguere i due processi. Se a è una qualsiasi stringa (supponiamo binaria per semplicità) e Q un processo, allora con la notazione $Pr[Q \rightsquigarrow a]$ si intende la probabilità che Q abbia a come output.
Se Q è un processo e $\mathcal{E}$ è una sequenza ordinata di eventi allora con i simboli $Pr[Q \rightsquigarrow \mathcal{E}]$ si indica la probabilità che Q esegua la sequenza ordinata di eventi $\mathcal{E}$.

Definizione 2.1 (Equivalenza Osservazionale Relativa alla Probabilità p) *Siano P e Q due processi allora P e Q sono osservazionalmente equivalenti relativamente a p (indicheremo la relazione con il simbolo $\approx_p$) se e solo se* $\forall t, C, a, \mathcal{E}$ $|Pr[C[P] \rightsquigarrow a] - Pr[C[Q] \rightsquigarrow a]| \leq p(t)$ $|Pr[C[P] \rightsquigarrow \mathcal{E}] - Pr[C[Q] \rightsquigarrow \mathcal{E}]| \leq p(t)$

Questa definizione vuole catturare il concetto che due processi sono equivalenti a meno di una probabilità p se nessun avversario che ha tempo di esecuzione t riesce a distinguerli con una probabilità maggiore di $p(t)$. Intendendo con $p(t)$ che la probabilità è funzione del tempo d'esecuzione di chi

[10]In questo contesto con evento non si intende necessariamento un oggetto dichiarato con il costrutto `event` ma una qualsiasi situazione *negativa* per il protocollo.

cerca di distinguere. Si può dire che questa relazione sia un adattamento della nozione di equivalenza osservazionale (del modello formale) in modo che questa corrisponda all'indistinguibilità del modello computazionale (questo nel momento in cui la probabilità che permette ai due processi di essere in relazione è trascurabile). Se due processi sono legati da questa relazione d'equivalenza allora sono equivalenti a meno di una certa probabilità nel senso che, qualsiasi attaccante (contesto) C che va in esecuzione per un tempo t ha una probabilità di distinguere questi due processi al massimo $p(t)$. Questa relazione d'equivalenza gode di alcune proprietà interessanti fra cui le seguenti:

1. *Se* $P \approx_u Q$ *e* $Q \approx_r R$ *allora* $P \approx_{u+r} R$
2. *Se* Q *esegue un evento* `e` *con probabilità al massimo* p *e* $Q \approx_r S$ *allora* S *esegue l'evento* `e` *con probabilità al massimo* $p + r$
3. *Se* $Q \approx_p S$ *e* C *si esegue in tempo* t_C *allora* $C[Q] \approx_r C[S]$ *dove* $r(t) = p(t + t_C)$

Quindi CryptoVerif, in generale, parte da una gioco G_0 che corrisponde ad un protocollo reale (di cui si vuole provare qualche proprietà) e costruisce una sequenza di giochi osservazionalmente equivalenti (secondo la definizione 2.1): $G_0 \approx_{p_1} G_1 \approx_{p_2} G_2 \cdots \approx_{p_n} G_{p_n}$; data questa sequenza, attraverso le proprietà 1 e 2, conclude che $G_0 \approx_{\sum_{i=1}^{n} p_i} G_{p_n}$, riuscendo così a dare un limite alla probabilità che un attaccante riesca a distinguere G_0 da G_{p_n}. Ora se in G_{p_n} non esiste la possibilità che un evento negativo possa accadere, visto che $G_0 \approx_{\sum_{i=1}^{n} p_i} G_{p_n}$, si può dare un limite superiore alla probabilità che l'evento avvenga in G_0. Questo limite è $\sum_{i=1}^{n} p_i$.
Come è stato spiegato precedentemente le regole di trasformazione da un gioco ad un altro sono date mediante delle equivalenze. Dato un gioco G, per utilizzare una generica regola di trasformazione $L \approx_p R$, CryptoVerif trova un contesto C tale che $G \approx_0 C[L]$ semplicemente apportando delle modifiche sintattiche al gioco G, costruisce inoltre un gioco G_s tale che $G_s \approx_0 C[R]$. CryptoVerif grazie alla Proprietà 3 può dedurre che $G \approx_p G_s$. Con questa tecnica CryptoVerif riesce a costruire catene di giochi che permettono di provare delle proprietà di alcuni protocolli. Le probabilità utilizzate nelle sequenze di giochi create da CryptoVerif non sono trattate con la notazione asintotica, questo perchè CryptoVerif adotta l'*exact security framework* [BR96]. Questo fatto permette al tool di calcolare in modo preciso le probabilità, invece di considerare l'andamento di queste quando il parametro di sicurezza tende ad infinito. Il fatto che CryptoVerif lavori nel modello esatto non significa che non può darci risultati estendibili a probabilità trascurabili.

Se per esempio alla fine di una prova CryptoVerif ha applicato k volte un'equivalenza (con k costante o valore polinomiale nel parametro di sicurezza) che vale a meno di una probabilità p allora in output sarà reso noto il fatto che la catena di giochi vale a meno della probabilità kt. Ora se la probabilità t è noto essere trascurabile allora lo sarà anche kt e quindi l'attaccante avrà una probabilità trascurabile di distinguere il gioco finale da quello iniziale; altrimenti significa che l'attaccante ha una probabilità non trascurabile pari esattamente a kt di vincere il gioco iniziale e quindi di distinguere questo dal gioco finale. Il calcolo di processi che CryptoVerif implementa è fondamentalmente quello che si trova in [BJST08] con però delle differenze. Una fra le più importanti è appunto quella che CryptoVerif può trasformare un gioco in un altro quando questi appartengono ad una stessa classe d'equivalenza secondo la definizione 2.1, mentre nel calcolo sviluppato in [BJST08] la definizione di relazione d'equivalenza su cui si basano le trasformazioni è diversa. In [BJST08] si considerano equivalenti giochi che sono distinguibili a meno di una probabilità la cui unica peculiarità è di essere trascurabile e nelle catene di giochi quindi la probabilità di distnguere il gioco iniziale da quello finale è ovviamente trascurabile[11]. In CryptoVerif invece si considerano i valori precisi delle probabilità in funzione di parametri dei giochi come per esempio numero di oracoli, lunghezza delle stringhe tempo d'esecuzione dell'attaccante ecc,ecc... Ecco quindi che CryptoVerif permette di calcolare in modo preciso le probablità in gioco, senza che queste vengano inglobate (e quindi perse) nella notazione asintotica. L'utilità principale di CryptoVerif è infatti quella di *fare bene i conti* delle dimostrazioni basate su giochi.

2.3 Un Esempio: FDH

Full Domain Hash è uno schema di firma che segue il paradigma *hash-and-sign*[12]. Quella che a breve seguirà è la descrizione dello schema di firma FDH nel linguaggio di CryptoVerif. In quest'esempio la funzione f sarà la funzione la cui inversa viene utilizzata per firmare messaggi, mentre la funzione $hash$ sarà la funzione utilizzata per ottenere il valore hash di una stringa di bit.
L'input fornito a CryptoVerif è costituito, fondamentalmente, da due parti:

- Un gioco iniziale G_0, in cui si modella la sicurezza dello schema.

[11] Si sa infatti che la somma di funzioni valori trascurabili rimane tale.

[12] 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

- Alcune equivalenze necessarie a CryptoVerif per effettuare le modifiche ai giochi.

La descrizione dello schema di firma FDH viene data al CryptoVerif attraverso il seguente gioco G_0:

```
foreach iH <= qH do
  OH(x : bitstring) := return(hash(x)) |
  Ogen() := r <-R seed;
    pk <- pkgen(r );
    sk <- skgen(r );
    return(pk ); (
    foreach iS <= qS do
         OS(m : bitstring) := return(invf(sk , hash(m))) |
         OT (m1 : bitstring, s : D) := if f(pk , s) = hash(m1 ) then
             find u <= qS suchthat (defined(m[u]) && m1 = m[u]) then
                  end
             else
                  event forge
)
```

Quella che segue è una breve spiegazione di questo gioco. Possiamo vedere come in questo gioco si forniscano `qH` copie dell' oracolo `OH` le quali ritornano il valore hash della stringa che gli si fornisce in input, ovvero `x`.
Abbiamo poi l'oracolo `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.
Vengono fornite, mediante il costrutto `foreach`, `qS` copie dell'oracolo `OS` le quali si occupano di restituire in output una firma del messaggio che gli viene dato in input. La firma di un messaggio `m`, data una chiave privata `sk` mediante la funzione `f`, viene effettuata restituendo il valore `invf(sk, m)`.
Infine viene fornito un oracolo `OT` che si occupa di verificare se la firma `s` è valida per il messaggio `m1`.
In particolare se `hash(m1)` $\neq$ `f(pk, s)` allora la firma non è valida per il messaggio. Se invece `hash(m1) = f(pk, s)`, l'oracolo si occupa di verificare se, per il messaggio `m1`, sia stata mai rilasciata una firma dall'oracolo `OS`. In caso affermativo il processo termina, altrimenti significa che 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 comportamento di questo. Possiamo immaginare un attaccante come un altro processo non meglio specificato messo

in parallelo con questo gioco.
Per poter trasformare un gioco in un altro, CryptoVerif ha bisogno di alcune definizioni di primitive crittografiche da poter utilizzare.
Le definizioni in CryptoVerif vengono fornite attraverso delle equivalenze che rappresentano regole di riscrittura sui processi e che possono valere a meno di una certa probabilità. Queste regole di riscrittura di un generico elemento L in un generico elemento R, possono essere valide 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 giochi.

Prima di descrivere la seconda parte dell'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 2.2 (Funzione One-Way) *Una funzione* $f : \{0,1\}^* \to \{0,1\}^*$ *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 nell'input.*

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 perché tutta la crittografia moderna è fondata sull'ipotesi che esistano le funzioni one-way. Si parla di ipotesi perché non è stata ancora dimostrata l'esistenza di funzioni di questo genere[13]. La dimostrazione dell'esistenza di funzioni one-way comporterebbe, tra l'altro, anche la risoluzione della famosa questione riguardo agli insiemi P e NP[14]. 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*.

[13] Questo non toglie che esistano funzioni che si avvicinino all'idea che abbiamo di funzioni one-way, per esempio le funzioni hash SHA1, o MD5 oppure una funzione f che prende in input due primi p e q in base binaria e ne restituisce il prodotto.

[14] 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.

Siamo ora pronti per vedere come il concetto di funzione one-way venga modellato in CryptoVerif. Si tratta di dare una definzione di one-wayness per mezzo di un'equivalenza fra giochi.
La definizione è la seguente:

```
equiv
       foreach iK <= nK do r <-R seed; (
       Opk() := return(pkgen(r)) |
       foreach iF <= nF do x <-R 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 <-R seed; (
       Opk() := return(pkgen'(r)) |
       foreach iF <= nF do x <-R 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() := let k:bitstring = mark in return(x))).
```

Per ora, non è interessante per quale probabilità questi giochi siano equivalenti, ecco perchè si è semplicemente indicato il valore della probabilità con *pow* pedice al simbolo $\approx$ senza specificare ulteriormente.

Questa equivalenza fra giochi cattura e definisce il concetto di funzione one-way. La funzione `f` infatti, è one-way se e solo se vale l'equivalenza di cui sopra, e quindi il gioco 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 $L \approx_{pow} R$. Questo perché potendo invertire `f`, A sarà in grado di osservare comportamenti diversi fra i giochi a seconda che A chiami `Oeq()` in L o in R. Procediamo con una sorta di esperimento mentale.
Supponiamo che A chiami `Oy()` ottenendo l'immagine di un valore mediante `f`, sia questa y; poiché `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 `Oeq()` è richiamato dal 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 giochi, 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 una probabilità trascurabile di cui parleremo dopo). 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, `Oeq()` si comporterà in maniera diversa a seconda che sia richiamato da L o da R. Se infatti `Oeq()` è chiamato da L, allora `Oeq()` si proccuperà semplicemente di ritornare il valore booleano dell'espressione `x'=x`. Se invece, `Oeq()`è chiamato da 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 definita, allora `Oeq()` si comporterà esattamente come si sarebbe comportato `Oeq()` di L. Se invece `k` non è stata definita 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.
È importante notare che l'attaccante avrebbe comunque la possibilità di indovinare `x` anche senza richiamare `Ox()`, ma questa è una probabilità trascurabile per definizione di one-wayness. Di questa probabilità viene comunque tenuto conto nell'equivalenza. L'equivalenza infatti, vale a meno di una certa probabilità *pow*. In questo modo i due giochi, sono effettivamente indistinguibili, perché non presentano differenze di comportamento che A possa notare, e quindi ai suoi occhi sono indistinguibili. Quindi L e R sono indistinguibili *se e solo se* `f` è one-way e di conseguenza la precedente equivalenza è una definizione ben posta di funzione one-way.

Un'ulteriore definizione che viene fornita a CryptoVerif è quella di funzione hash. Questa definizione viene fornita al tool attraverso il seguente codice:

```
fun hash(hashinput):hashoutput.
equiv foreach iH <= nH do
        OH(x:hashinput) := return(hash(x)) [all]
        <=(0)=>
      foreach iH <= nH do
```

```
OH(x:hashinput) :=
    find[unique] u <= nH suchthat defined(x[u],r[u]) &&
            otheruses(r[u]) && x= x[u] then
                return(r[u])
    else
       r <-R 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 restituito un valore casuale, altrimenti viene restituito lo stesso valore che era stato dato in output precedentemente. Questo viene fatto salvando il valore restituito per un particolare input in un array e poi, al momento della chiamata, si effettua un lookup 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`[15]. Nell'appendice A si può trovare l'input completo, mentre in [BP06] possiamo trovare una spiegazione delle fasi più importanti dell'output di CryptoVerif e infine all'url `http://www.cryptoverif.ens.fr/FDH/fdh.pdf` troviamo l'ouput completo di CryptoVerif sull'input descritto precedentemente.

[15]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 caso particolare di un risultato classico della crittografia computazionale. La dimostrazione procederà utilizzando CryptoVerif per dimostrare il caso base di una una prova che sarà in generale di tipo induttivo.

3.1 Il Teorema

Il risultato nella sua generalità è il seguente teorema:

Teorema 3.1 *Se esiste un generatore pseudocasuale* $\hat{G} : \{0,1\}^n \to \{0,1\}^{n+1}$ *(con fattore d'espansione* $\hat{l}(n) = n + 1$*) allora per ogni polinomio* $p(n)$ *(tale che* $\forall n\ p(n) > n$*) esiste un generatore pseudocasuale* G *con coefficiente d'espansione* $l(n) = p(n)$*.*

Si può trovare una dimostrazione *classica*[1] di questo teorema in [KL07]. La dimostrazione classica è di tipo costruttivo e procede in modo iterativo. La costruzione inizia applicando ad un seme casuale r di lunghezza n il generatore pseudocasuale $\hat{G}$. $\hat{G}(r)$ sarà una stringa pseudocasuale di $n+1$ bit, di questi uno andrà a costituire parte dell'output finale mentre gli altri n fungeranno da seme per una successiva applicazione di $\hat{G}$. È lecito utilizzare n bit dell'output di $\hat{G}$ come successivo input di questo, infatti il risultato di $\hat{G}$ è una stringa pseudocasuale[2] e quindi utilizzabile come seme di $\hat{G}$ tanto quanto lo è una stringa effettivamente casuale. La dimostrazione procede poi estendendo questa iterazione ad un numero polinomiale di volte e dimostrando

[1] Per classica qui si intende *non formale* nel senso informatico del termine. Si intende cioè che il linguaggio con cui la dimostrazione è stata fatta utilizza anche un linguaggio che non è modellato attraverso un sistema formale. La dimostrazione è scritta infatti in un linguaggio naturale.

[2] E quindi anche una sua parte è pseudocasuale.

attraverso una tecnica *ibrida* che il risultato di questa iterazione è comunque una stringa pseudocasuale. Quella ibrida è una tecnica molto utilizzate per provare l'indistinguibilità di due distribuzioni statistiche ed è la naturale applicazione del concetto di sequenza di giochi a distribuzioni statistiche. La tecnica ibrida definisce una serie di distribuzioni che si situano fra due distribuzioni *estreme* che sono quelle che si vogliono dimostrare indistinguibili. In particolare le distribuzioni estreme della dimostrazione sono la distribuzione H_0 definita dai possibili valori di $G(r)$ con r estratto casualmente da $\{0,1\}^n$ e $H_{p(n)-n}$ definita dai valori di r con r scelto casualmente da $\{0,1\}^{p(n)}$. Alla fine dei calcoli la dimostrazione prova che se ci fosse una differenza sensibile fra le distribuzioni estreme allora necessariamente dovrebbe anche esserci una differenza sensibile fra due distribuzioni vicine. E poichè questo sarebbe assurdo, perchè negherebbe che $\hat{G}$ è un generatore pseudocasuale, giunge al risultato desiderato, ovvero che le distribuzioni estreme sono indistinguibili che equivale a dire che G è un generatore pseudocasuale.

Il risultato provato in questo lavoro è, come gia detto, molto meno generale del teorema classico ed è enunciato nel seguente:

Teorema 3.2 *Se esiste un generatore pseudocasuale* $\hat{G} : \{0,1\}^n \to \{0,1\}^{n+1}$ *(con fattore d'espansione* $\hat{l}(n) = n + 1$*) allora per ogni costante* $k > 1$ *esiste un generatore pseudocasuale* G *con coefficiente d'espansione* $l(n) = n + k$*.*

Si dimostrerà il teorema attraverso un procedimento induttivo che sfrutta CryptoVerif. In particolare, la dimostrazione si articolerà in due fasi come ogni prova induttiva:

- *Caso Base.* Mediante CryptoVerif si dimostrerà il teorema per $k = 2$.
- *Caso Induttivo.* Si dimostrerà che CryptoVerif riesce a dimostrare il teorema per $k = i + 1$ assumendo di essere riuscito a dimostrarlo per $k = i$.

Quando d'ora in poi si userà l'espressione *essere riuscito a dimostrare* riferendoci a CrytoVerif si intenderà che CryptoVerif ha fornito una sequenza di giochi nell'ultimo della quale l'asserto vale.

3.2 Dimostrazione del Caso Base

Si cercherà ora di dimostrare il caso base del procedimento induttivo prima accennato. Ovvero si descriverà un fil di input per CryptoVerif che permette a questo di provare che: Se esiste un generatore pseudocasuale $\hat{G} : \{0,1\}^n \rightarrow \{0,1\}^{n+1}$ (con fattore d'espansione $\hat{l}(n) = n + 1$) allora, per $k = 2$, esiste un generatore pseudocasuale G con coefficiente d'espansione $l(n) = n + k$. L'input che è stato fornito a CryptoVerif inizia con le seguenti dichiarazioni:

```
1 type nbits   [fixed].
2 type np1bits [fixed].
3 type np2bits [fixed].
```

Alla linea 1 definiamo il tipo *nbits* ovvero il tipo delle stringhe di lunghezza n. Questo tipo rappresenterà l'insieme $\{0,1\}^n$. Alla linea 2 invece viene dichiarato un tipo per rappresentare le stringhe appartenenti all'insieme $\{0,1\}^{n+1}$, mentre nelle righe 3 lo stesso viene fatto per le stringhe appartenenti all' insieme $\{0,1\}^{n+2}$. Si noti come ogni tipo sia dichiarato con il modificatore $fixed$ al fine di poter estrarre elementi casuali dai domini di quei tipi. Con la linea di codice seguente:

```
5 param n1.
```

Si dichiara un parametro $n1$. Questo parametro assumerà valori polinomiali nel parametro di sicurezza. Sarà utilizzato principalmente per fornire al distinguitore un numero polinomiale di oracoli con cui interagire. Quella che segue è una dichiarazione di un simbolo funzionale con i relativi tipi di input e di output.

```
7 fun concatnp1(np1bits, bool):np2bits.
```

In particolare la funzione prende in input come argomenti una stringa di lunghezza $n + 1$ e un bit, quest'ultimo rappresentato attraverso un valore booleano. L'output sarà una stringa di lunghezza $n+2$. Come si può intuire dal nome della funzione, questa rappresenta la concatenazione di una stringa di lunghezza $n+1$ con un bit. Mediante la seguente equivalenza non facciamo altro che implementare la funzione dichiarata prima. La funzione `concatnp1` viene descritta mediante un'equivalenza che rappresenta il seguente asserto: *la concatenazione di una stringa casuale di* $n+1$ *bit con un bit casuale è una stringa casuale di* $n + 2$ *bit.* . L'equivalenza è la seguente:

```
10 equiv
11       foreach i1<=n1 do
```

```
12              r <-R np1bits;
13              b <-R bool;
14              OGet():=return (concatnp1(r,b))
15       <=(0)=>
16      foreach i1 <=n1 do
17              w <-R np2bits;
18              OGet():=return(w).
```

In questo caso l'equivalenza non vale a meno di una probabilità ma sempre. Infatti è sempre vero che la concatenazione di una stringa casuale e di un bit casuale è una stringa casuale. Con il seguente frammento di codice:

```
38 fun G'(nbits): np1bits.
39 equiv
40     foreach i1<=n1 do
41                 r <-R nbits;
42                 OGet():=return (G'(r))
43       <=(POW)=>
44      foreach i1 <=n1 do
45                 x <-R np1bits;
46                 OGet():=return(x).
```

per prima cosa si dichiara un simbolo funzionale `G'` che sta per il $\hat{G}$ della dimostrazione classica. La dichiarazione viene ovviamente con i tipi di input e output del generatore pseudocasuale. Si noti come l'equivalenza valga a meno di una certa probabilità. Infatti una stringa casuale è, in generale, di natura diversa da una stringa pseudocasuale sebbene siano distinguibili solo per una probabilità trascurabile che è rappresentata appunto da `POW`. La probabilità `POW` è la probabilità che un attaccante efficiente riesca a distinguere una stringa casuale da una pseudocasuale. La probabilità `POW` è trascurabile per definizione di pseudocasualità. Con il seguente frammento di codice non facciamo altro che istruire CryptoVerif su cosa vogliamo che venga dimostrato.

```
query secret1 w.
```

Il precedente frammento è lecito perchè ciò che si vuole è che il tool provi che la stringa `w` sia indistinguibile da una stringa casuale. Nell'ultimo frammento di codice si vedrà come sia definita questa stringa. L'ultimo frammento di codice rappresenta una sorta di *protocollo* che funge da gioco iniziale:

```
50 process
51         G():=
```

```
52              r<-R nbits;
53              let x' = G'(r) in
54              let y' = getn(x') in
55              let b' = getlast(x') in
56              w:np2bits <-concatnp1(G'(y'), b');
57              return
```

in questo gioco si applica due volte il generatore pseudocasuale `G'` la prima volta alla stringa `r` e la seconda al risultato fornito dalla prima applicazione di `G'`. In particolare si noti come la variabile `r` venga inizializzata con un valore casuale preso dall'insieme delle stringhe binarie di lunghezza n, e come successivamente si proceda con l'assegnamento ad `x'` della stringa pseudocasuale `G'(r)`. Dopo questo passaggio, i primi n bit della stringa `x'` vengono estratti e assegnati alla variabile `y'` mentre l'$n+1$-esimo bit viene assegnato alla variabile `b'`. Adesso è possibile utilizzare nuovamente `G'` su input `y'` e concatenare quindi `G'(y')` con `b'` ottenendo la stringa `w`. `w` quindi è il risultato di un duplice utilizzo del generatore pseudocasuale `G'`. Dimostrando la proprietà di one-session secrecy per la stringa `w` CryptoVerif non fa altro che dimostrare la pseducasualità di questa.

3.2.1 I Giochi della Prova

Il seguente è il gioco iniziale in cui si utilizza due volte il generatore pseudocasuale `G'`; la prima applicandolo ad un seme casuale `r` e la seconda applicandolo a parte dell'output di `G'(r)`.

```
Game 1 is
O() :=
    r <-R nbits;
    x': np1bits <- G'(r);
    y': nbits <- getn(x');
    b': bool <- getlast(x');
    w: np2bits <- concatnp1(G'(y'), b');
    return()
```

CryptoVerif prova ad applicare le equivalenze che gli sono state fornite e trova che può applicare solo la definizione di `G'`. Si noti come CryptoVerif cambi alcuni nomi. Per esempio `i1` diventa `i1_5` nel membro sinistro dell'equivalenza e `i1_6` nel membro destro. Questa ridenominazione avviene per evitare situazioni in cui si abbia uno stesso nome per due oggetti diversi. Si noti infine che l'equivalenza vale a meno della probabilità `POW`. Questa probabilità è asintoticamente trascurabile (per definizione di pseudocasualità) ma nel modello esatto se ne deve tener conto comunque.

```
Applying equivalence
equiv foreach i1_5 <= n1 do
              r <-R nbits;
              OGet() := return(G'(r))
<=(set(proba <= POW))=>
      foreach i1_6 <= n1 do
              x <-R np1bits;
              OGet() := return(x)

[Excluding set(dist 1->2, proba <= POW)] yields
```

Nel seguente gioco, ovvero il 2, viene inizializzata una nuova variabile `x_7` con un valore casuale dal dominio `np1bits`, questa variabile viene poi assegnata a quella che prima conteneva il risultato di `G'(r)` e cioè `x'`.

```
Game 2 is
O() :=
     x_7 <-R np1bits;
     x': np1bits <- x_7;
```

```
     y': nbits <- getn(x');
     b': bool <- getlast(x');
     w: np2bits <- concatnp1(G'(y'), b');
     return()
```

Nel gioco 2 c'è un assegnamento che è inutile ovvero quello che coinvolge la variabile `x'`; infatti si può eliminare `x'` e sostituirla con `x_7`. Il risultato di questa operazione lo si trova nel gioco 3:

```
Applying remove assignments of useless yields

Game 3 is
O() :=
     x_7 <-R np1bits;
     y': nbits <- getn(x_7);
     b': bool <- getlast(x_7);
     w: np2bits <- concatnp1(G'(y'), b');
     return()
```

Applicando ora la seguente equivalenza (che si ricordi vale sempre e non a meno di una probabilità):

```
Applying equivalence
equiv foreach i1_3 <= n1 do
              r <-R np1bits; (
              OGetn() := return(getn(r)) |
              OGetlast() := return(getlast(r)))
<=(0)=>
      foreach i1_4 <= n1 do
              x <-R nbits;
              xl <-R bool; (
              OGetn() := return(x) |
              OGetlast() := return(xl))

yields
```

è possibile modificare il gioco 3 nel seguente gioco in cui le variabili `xl_9` e `x_8` sono inizializzate con dei valori casuali estratti rispettivamente dal dominio dei booleani e dal dominio `nbits` ovvero dalle stringhe di lunghezza n.

```
Game 4 is
O() :=
     xl_9 <-R bool;
```

```
     x_8 <-R nbits;
     y': nbits <- x_8;
     b': bool <- xl_9;
     w: np2bits <- concatnp1(G'(y'), b');
     return()
```

Si può vedere come anche nel gioco 4 si possono trovare degli assegnamenti inutili ovvero quelli che coinvolgono le variabili `y'` e `b'`. È possibile infatti, eliminare queste variabili sostituendo ogni occorrenza di queste con le variabili `xl_9` e `x_8`. Effettuando queste modifiche si ottiene il seguente gioco:

```
Applying remove assignments of useless yields

Game 5 is
O() :=
xl_9 <-R bool;
x_8 <-R nbits;
w: np2bits <- concatnp1(G'(x_8), xl_9);
return()
```

Nel gioco 5 l'unica cosa che CryptoVerif può fare è quella di applicare nuovamente la definizione di `G'`:

```
Applying equivalence
equiv foreach i1_5 <= n1 do
              r <-R nbits;
              OGet() := return(G'(r))
<=(set(proba <= POW))=>
      foreach i1_6 <= n1 do
              x <-R np1bits;
              OGet() := return(x)
[Excluding set(dist 5->6, proba <= POW)] yields
```

ottenendo il seguente gioco in cui sia `xl_9` che `x_10` sono stringhe casuali estratte rispettivamente dal dominio dei booleani e dal dominio delle stringhe di lunhezaz $n + 1$. Si ottiene quindi il gioco 6:

```
Game 6 is
O() :=
     xl_9 <-R bool;
     x_10 <-R np1bits;
     w: np2bits <- concatnp1(x_10, xl_9);
     return()
```

```
Applying equivalence
equiv foreach i1_1 <= n1 do
       r <-R np1bits;
       b <-R bool;
       OGet() := return(concatnp1(r, b))
<=(0)=>
      foreach i1_2 <= n1 do
      w <-R np2bits;
      OGet() := return(w)
yields
```

in questo gioco CryptoVerif non fa altro che applicare la definizione di `concatnp1` che afferma che la concatenazione di una stringa casuale di n bit e di un bit casuale è una stringa casuale di $n + 2$ bit. Otteniamo quindi il gioco finale, ovvero il gioco 7:

```
Game 7 is
O() :=
     w_11 <-R np2bits;
     w: np2bits <- w_11;
     return()
```

In quest'ultimo gioco alla variabile `w` è assegnato il valore di una stringa `w_11` che è scelta in maniera completamente casuale (con il costrutto `<-R`) dal dominio del tipo `np2bits`. Di conseguenza la stringa `w` è in questo gioco un variabile casuale a tutti gli effetti. In questo gioco, quindi, l'attaccante non può distinguere la stringa `w` da una stringa casuale perchè `w` stessa è casuale, di conseguenza in questo gioco l'attaccante non può vincere. CryptoVerif ha quindi dimostrato l'asserto iniziale, ovvero che dato un generatore pseudocasuale con fattore d'espansione $n + 1$ se ne può ottenere uno con coefficiente d'espansione $n + 2$. Ecco che tutto ciò ci porta alla parte finale dell'output di CryptoVerif in cui questo ci dice per quali probabilità vale la prova:

```
RESULT Proved secrecy of w excluding
set(dist 1->2, proba <= POW) U
set(dist 5->6, proba <= POW)
RESULT Proved secrecy of w with probability 2. * POW
All queries proved.
```

siccome nella catena di giochi la definizione basata su equivalenza di `G'` è stata applicata due volte è ovvio che la catena di giochi sia valida a meno di `2 * POW` dove `POW` è la probabilità per cui vale la definizione di `G'`.

3.3 Il passo induttivo

Quello che si cercherà ora di fare è di provare che CryptoVerif riesce a dimostrare il teorema per $k = i+1$ sfruttando l'ipotesi che riesca a dimostrarlo per $k = i$. Per poter dimostrare l'asserto per $k = i + 1$ CryptoVerif, in generale, partirà da un file di input che contiene delle dichiarazioni e un gioco iniziale. Qui di seguito si darà uno schema di questo file. Nell'input i valori i e $i + 1$ stanno al posto degli effettivi valori di i. Così il tipo: `type npibits` dovrà essere inteso come un tipo nel cui nome la lettera i è stata sostituita con un particolare valore.

In particolare il file dovrà avere:

1. la dichiarazione `type npbits [fixed].`. Questo tipo si trova in ogni file di input per ogni k.

2. $i + 1$ dichiarazioni:

   - `type np1bits [fixed].` Questo tipo si trova in ogni file di input per ogni k.
   - `type np2bits [fixed].`
   - ...
   - ...
   - `type npibits [fixed].`
   - `type npi+1bits [fixed].`

   La lettera i sarà sostituita dal particolare valore numerico che assume, stessa cosa per $i + 1$ allora si avrà `type np5bits [fixed].` e `type np6bits [fixed].`

3. i dichiarazioni di simboli funzionali:

   - `fun concatnp1(np1bits,bool):np2bits.`
   - `fun concatnp2(np2bits, bool):np3bits.`
   - ...
   - ...
   - `fun concatnpi(npibits, bool):npi+1bits.`

4. $i + 1$ equivalenze del tipo:

- ```
  equiv
        foreach i1<=n1 do
                r<-R npibits;
                b<-R bool;
                Oget():=return(concatnpi(r, b))
        <=(0)=>
         foreach i1<=n1 do
                w<-R npi+1bits;
                Oget():=return(w).
  ```

  Anche in questa equivalenza i va sostituita con il valore che questa assume.

5. le dichiarazioni dei seguenti simboli funzionali:

   - `fun getn(np1bits):nbits.`. Questo simbolo si trova in ogni file di input per ogni k.
   - `fun getlast(np1bits):bool.`. Questo simbolo si trova in ogni file di input per ogni k.

6. la definizione dei simoboli funzionali precedenti:

   - ```
     equiv
     foreach i1<=n1 do
           r <-R np1bits;(
           OGetn():=return (getn(r)) |
           OGetlast():=return (getlast(r)))
     <=(0)=>
    foreach i1 <=n1 do
            (
            OGetn():= x <-R nbits;return(x) |
            OGetlast():= xl <-R bool;return(xl)
).
     ```

7. la dichiarazione del generatore pseudocasuale con coefficiente d'espansione $n + 1$. Questo simbolo si trova in ogni file di input per ogni k.

   - ```
     fun G'(nbits): np1bits.
     ```

8. la definizione, mediante equivalenza, del generatore pseudocasuale con coefficiente d'espansione $n + 1$:
     ```

```
•       equiv
        foreach i1<=n1 do
                    r <-R nbits;
                    OGet():=return (G'(r))
          <=(POW)=>
         foreach i1 <=n1 do
                    x <-R np1bits;
                    OGet():=return(x).
```

9. la clausola: `query secret w.`. Questa clausola si trova in ogni file di input per ogni k.

10. la definizione del protocollo iniziale, mediante un gioco, in cui si descrive il generatore pseudocasuale con coefficiente d'espansione $n + i + 1$:

```
• process
        O():=
              r<-R nbits;
              let x1 = G'(r) in
              let y1 = getn(x1) in
              let b1 = getlast(x1) in
              let x2=G'(y1) in
              let y2=getn(x2) in
              let b2=getlast(x2) in
              ...
              ...
              ...
              let xi=G'(yi-1) in
              let yi=getn(xi-1) in
              let bi=getlast(xi-1) in
  w:npi+1bits<-concatnpi(
             concatnpi-1(...concatnp1(G'(yi), b1), b2),... bi);
  return()
```

Vediamo un esempio: se $k = i + 1 = 4$, allora il file di input per provare l'asserto è il seguente:

```
1 type nbits [fixed].      (* stringhe di lunghezza  n *)
2 type np1bits [fixed].   (* stringhe di lunghezza  n+1 *)
3 type np2bits [fixed].   (* stringhe di lunghezza  n+2 *)
```

```
 4
 5 type np3bits [fixed].   (* stringhe di lunghezza  n+3 *)
 6 type np4bits [fixed].   (* stringhe di lunghezza  n+4 *)
 7
 8 param n1.
 9
10 (* La concatenazione di una stringa random di n+1 bits
     e un bit random è una stringa random di n+2 bits. *)
11 fun concatnp1(np1bits,bool):np2bits.
12 fun concatnp2(np2bits, bool):np3bits.
13 fun concatnp3(np3bits, bool):np4bits.
14
15 (* La concatenzione di stringhe random è una stringa random*)
16 equiv
17       foreach i1<=n1 do
18             r <-R np1bits;
19             b <-R bool;
20             OGet():=return (concatnp1(r,b))
21       <=(0)=>
22      foreach i1 <=n1 do
23             w <-R np2bits;
24             OGet():=return(w).
25
26 equiv
27     foreach i1<=n1 do
28         r<-R np2bits;
29         b2<-R bool;
30         Oget():=return(concatnp2(r, b2))
31         <=(0)=>
32     foreach i1<=n1 do
33         w<-R np3bits;
34         Oget():=return(w).
35
36 equiv
37     foreach i1<=n1 do
38         r<-R np3bits;
39         b3<-R bool;
40         Oget():=return (concatnp3(r, b3))
41         <=(0)=>
42     foreach i1<=n1 do
43         w<-R np4bits;
```

```
44         Oget():=return (w).
45
46
47
48
49 (* Estrae i primi n bits da una stringa di n+1 bits *)
50 fun getn(np1bits):nbits.
51 (* Estrae l'ultimo bit da una stringa di n+1 bits *)
52 fun getlast(np1bits):bool.
53 equiv
54        foreach i1<=n1 do
55              r <-R np1bits;(
56              OGetn():=return (getn(r)) |
57              OGetlast():=return (getlast(r)))
58        <=(0)=>
59       foreach i1 <=n1 do
60               (
61               OGetn():= x <-R nbits;return(x) |
62               OGetlast():= xl <-R bool;return(xl)
63               ).
64 proba POW.
65 (* Generatore pseudorandom con fattore n+1 *)
66 fun G'(nbits): np1bits.
67 equiv
68     foreach i1<=n1 do
69                 r <-R nbits;
70                 OGet():=return (G'(r))
71       <=(POW)=> (* da definire *)
72      foreach i1 <=n1 do
73                 x <-R np1bits;
74                 OGet():=return(x).
75
76 query secret w.
77
78 process
79         O():=
80              r<-R nbits;
81              let x1 = G'(r) in
82              let y1 = getn(x1) in
83              let b1 = getlast(x1) in
84              let x2=G'(y1) in
```

```
 85              let y2=getn(x2) in
 86              let b2=getlast(x2) in
 87              let x3=G'(y2) in
 88              let y3=getn(x3) in
 89              let b3=getlast(x3) in
 90 w:np4bits <-concatnp3(concatnp2(concatnp1(G'(y3), b1), b2), b3);
 91 return()
```

3.3.1 L'Ipotesi Induttiva

Prima di procedere è bene precisare che l'espressione: *per $k = i$ CryptoVerif riesca a dimostrare l'asserto*[3] significa che: per $k = i$ CryptoVerif partendo da un gioco iniziale[4] $G_0(i)$, descritto nel seguente frammento di codice:

```
O():=
  r<-R nbits;
  let x' = G'(r) in
  let y' = getn(x') in
  let b' = getlast(x') in
  let x''=G'(y') in
  let y''=getn(x'') in
  let b''=getlast(x'') in
  ...
  ...
  ...
w:npibits<-concatnpi-1(
            concatnpi-2(...concatnp1(G'(yi), b1), b2),... bi);
return()
```

riesce a giungere al seguente game finale $G_n(i)$:

```
O():=
  w_i <-R npibits;
  w <- w_i;
return ();
```

Dove la stringa `w` non può essere distinta da una stringa casuale. Si noti che, proprio per costruzione dell'input, le istruzioni di $G_0(i)$ sono un sottoinsieme delle istruzioni del gioco iniziale per $k = i + 1$, sia questo $G_0(i + 1)$. Infatti $G_0(i + 1)$ ha il seguente aspetto:

[3] Che è l'ipotesi induttiva che verrà sfruttata.

[4] Che si può ottenere analizzando lo scheletro fornito precedentemente per $k = i + 1$ ed applicandolo ad $k = i$.

```
O():=
  r<-R nbits;
  let x' = G'(r) in
  let y' = getn(x') in
  let b' = getlast(x') in
  let x''=G'(y') in
  let y''=getn(x'') in
  let b''=getlast(x'') in
  ...
  ...
  ...
w:npi+1bits<-concatnpi(
            concatnpi-1(...concatnp1(G'(yi), b1), b2),... bi);
return()
```

Inoltre il file di input per $k = i+1$ avrà sempre un'equivalenza in più rispetto al file per $k = i$. Questa equivalenza è quella che serve a concatenare una stringa casuale di $n+i$ bit e un bit casuale, in una stringa casuale di $n+i+1$ bit; si sta parlando dell'equivalenza relativa al simbolo funzionale *concatnpi*. Quindi CryptoVerif partendo da $G_0(i+1)$ e utilizzando *solo* le regole di riscrittura relative a $k = i$ giunge al gioco $G_{n-1}(i+1)$[5]:

```
Game n-1 is
O() :=
xl <-R bool;
w_z <-R npibits;
w: npi+1bits <- concatnpi(w_z, xl);
return()
```

in cui tutte le stringhe che, oltre ad appartenere al gioco $G_0(i+1)$, appartenevano anche a $G_0(i)$ sono confluite, grazie alle equivalenze, in w_z. Si osservi, tra l'altro che questo gioco, penultimo nella sequenza per $k = i+1$, è un sovrainsieme dell'ultimo gioco per $k = i$. Ora utilizzando la sola regola di riscrittura *concatnpi*, l'unica in più rispetto al file per $k = i$, CryptoVerif può passare al seguente gioco:

```
Game n is
O() :=
w_a <-R npi+1bits;
w: npi+1bits <- w_a
return()
```

[5]Il penultimo della sequenza. Si vedrà infatti che con un ultima applicazione di una regola si giungerà al gioco finale

Dove la variabile w è effettivamente casuale.

Non si mostra la sequenza di giochi per $k = i$ perchè per ipotesi induttiva si suppone che esista. Di questa sequenza però si conosce il gioco iniziale, che per costruzione è un sottoinsieme del gioco iniziale per $k = i + 1$.

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.

[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.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with rsa and rabin. In *EUROCRYPT*, pages 399–416, 1996.

[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.

[MRST06] John C. Mitchell, Ajith Ramanathan, Andre Scedrov, and Vanessa Teague. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. *Theor. Comput. Sci.*, 353(1-3):118–164, 2006.

[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`.