Introduzione

Negli anni anni '80 la crittografia ha cessato di essere un' arte per assurgere allo stato di scienza.
Come scienza, la crittografia moderna si propone innanzi tutto di fornire definizioni rigorose dei concetti con cui ha a che fare (e.g. *segretezza*) per poi apportare prove dei propri asserti basandosi su argomentazioni di carattere logico-matematico, ovvero di dimostrazioni.
Sebbene in alcuni casi sia possibile (ed è stato fatto, i.e [Sha49]) *incondizionatamente* provare 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; questo perché ogni dimostrazione è del tipo: *Se X allora Y* (e.g. se esiste una funzione one-way allora esiste un generatore pseudocasuale con fattore d'espansione polinomiale). Semplicemente, nel caso in cui le assunzioni dovessero essere confutate si renderebbero poco interessanti le dimostrazioni che si basano su queste[1]. È però giusto dire che le assunzioni che vengono fatte sono largamente ritenute vere; questo anche grazie al fatto che alcune di esse si possono dedurre da altre ipotesi 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

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

che hanno una semantica probabilistica. Il primo modello è stato ampiamente trattato in [AG99, BAN90, Kem87, Pau98]. Il secondo modello trova le basi in lavori di 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

[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.

troppi problemi. Un' altra strada, invece, è quella che prevede di lavorare direttamente nel modello computazionale senza preoccuparsi di rispettare le forti ipotesi che erano state usate per dimostrare il risultato raggiunto in [AR07]. Rispetto al passato, oggi giorno, la crittografia non è utilizzata solo in ambiente militare. I suoi 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 *BPP* (i.e. *Bounded Probability Polynomial Time*).
Un modo più formale di vedere questo tipo di algoritmi è il seguente: si suppone che la macchina di Turing che esegue l'algoritmo, oltre a ricevere l'input, diciamo x, riceva anche un input ausiliario r. Questa stringa di bit r, rappresenta una possibile sequenza di lanci di moneta dove è stato associato al valore 0 la croce e al valore 1 la testa (o anche viceversa ovviamente). Qualora la macchina dovesse effettuare una scelta casuale, non dovrà far altro che prendere il successivo bit dalla stringa r, e prendere una decisione in base ad esso (è, in effetti, come se avesse preso una decisione lanciando una moneta). Ecco quindi che il nostro tipico avversario si configura come un algoritmo polinomiale probabilistico. È inoltre giustificato cercare di rendere sicuri[5] gli schemi crittografici rispetto, principalmente, a questo tipo di avversario. Non è quindi necessario dimostrare che un particolare schema crittografico sia inviolabile, ma basta dimostrare che:

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

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

1.2 Funzioni Trascurabili e non ...

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

Definizione 1.1 *Funzione Trascurabile. Sia $\mu : \mathbb{N} \to \mathbb{R}^+$ una funzione. Si dice che μ è trascurabile se e solo se per ogni polinomio p, esiste $C \in \mathbb{N}$ tale che $\forall n > C$: $\mu(n) < \frac{1}{p(n)}$.*

[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 Games

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

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

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 game possono essere immaginati come un insieme di azioni, modellate in una particolare algebra di processi, che servono a specificare il comportamento dei partecipanti al gioco, ovvero le entità che partecipano come *principals* al protocollo crittografico.
Il lettore troverà utile pensarli, almeno nell'ambito di questa tesi, come insiemi di processi che, fra le altre cose, forniscono un'interfaccia all'attaccante attraverso degli *oracoli* che possono resitituire dei valori all'attaccante. Questi oracoli, prima di 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 game fanno, è quello di modellare il protocollo crittografico reale in un game iniziale G_0. In G_0 esiste la probabilità non nulla che un evento *negativo* possa accadere (immaginiamolo come una sorta di vittoria da parte dell'attaccante). Ora, in generale, si procede effettuando delle modifiche al game G_i ottenendo un game G_{i+1} tale che G_i e G_{i+1} siano computazionalmente indistinguibili. Le modifiche che si effettuano fra un game e un altro devono introdurre delle differenze computazionalmente irrilevanti. Queste modifiche possono essere viste come regole di riscrittura delle distribuzioni di probabilità delle variabili in gioco nei games. Se, per esempio, in un game G_i un processo ha a che fare con un variabile casuale, e nel game G_{i+1} questa viene sostituita con una variabile che invece è pseudocasuale, non vengono introdotte modifiche computazionalmente rilevanti e quindi il passaggio è lecito, visto che i due games non sono distinguibili se non con probabilità trascurabile. Alla fine si arriva ad un game G_f in cui *l'evento* non può accadere. Se l'evento non può accadere, l'attaccante non ha possibilità di vincere. Se, quindi, nel game finale l'attaccante non ha possibilità di vincere e il game finale è computazionalmente indistinguibile dal penultimo, questo lo è dal terzultimo e così via fino a G_0, allora il game finale è computazionalmente indistinguibile dal primo[14]. Adesso, quindi, se nel game iniziale esiste la possibilità che un evento avvenga, e nel game finale no, si può dare un limite superiore alla probabilità che l'evento avvenga nel game iniziale. Questo limite è la somma di tutte le probabilità con cui un attaccante riesce a distinguere un game dal successivo

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

all'interno della sequenza.

È importante dire che anche nel modello formale si possono utilizzare le sequenze di game. La differenza è che due games successivi non sono computazionalmente indistinguibili, ma sono perfettamente indistinguibili. In particolare quello che si vuole sottolineare è che se in una sequenza di game G_b e G_a sono l'uno il successore dell'altro, allora i due game devono appartenere ad una stessa classe di equivalenza che è indotta dalla particolare relazione di equivalenza che il modello in cui si sta costruendo la catena di games sfrutta. Nel modello formale questa relazione sarà l'indistinguibilità perfetta (meglio nota come equivalenza osservazionale) mentre in quello computazionale sarà l'indistinguibilità computazionale.

Capitolo 2

CryptoVerif

CryptoVerif è un *dimostratore* automatico di sviluppo 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 per dimostrare proprietà di segretezza e autenticazione. Le prove in questione si basano sulla tecnica delle sequenze di game. Il tool è liberamente scaricabile[4] sotto la licenza CeCill[5]. CryptVerif è stato già utilizzato per dimostrare la correttezza di alcuni protocolli crittografici, come per esempio: FDH [BP06], Kerberos [BJST08].
Scopo di questo capitolo è quello di descrivere, in modo non troppo formale[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

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

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

Si cercherà di fornire maggiori spiegazioni senza però, per ora, entrare troppo nel dettaglio. Una dichiarazione (declaration) serve a CryptoVerif per avere delle informazioni su come comportarsi nella dimostrazione; spesso è una 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 viene insieme ai tipi dei parametri in input (se ce ne sono) e il tipo del valore di ritorno. Per esempio, con il seguente codice:

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

si dichiara una funzione $kgen : keyseed \rightarrow key$. Intuitavamente il simbolo *kgen* rappresenta una funzione per generare chiavi crittografiche a partire da un seme casuale.

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

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

si dichiara un nuovo tipo *keyseed*. La parola chiave $fixed$ serve per informare CryptoVerif che deve essere possibile estrarre valori casuali da quell'insieme di valori. 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 trascurabile.

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 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 è f iniettiva.

Regola di Riscrittura. Una dichiarazione di una regola di riscrittura consiste in una coppia di espressioni. Ogni qualvolta CrytptoVerif incontra la prima può decidere di sostituirla con la seconda. Per esempio, con il seguente 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 che se si dessero, invece delle precedenti, queste regole di riscrittura:

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

si andrebbe incontro ad un probabile 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 usate come *oggetti* a se stanti, ovvero come dei valori statici, ma possono anche essere utilizzate come funzioni di altri argomenti. Con il seguente codice:

```
proba POW.
```

si dichiara una probabilità POW.

Costante. Dichiarando una costante k si dice a CryptoVerif che quel dato, una volta inizializzato, non può cambiare di valore durante l'esecuzione del protocollo. Per esempio, con il seguente 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 di un protocollo. Per esempio si potrebbe pensare di dichiarare un evento *forge* da *sollevare* nel momento in cui un avversario riesce a *forgiare* una firma valida per un messaggio. Per esempio, con il seguente codice:

```
event forge.
```

si dichiara un evento forge. Si osservi il seguente codice:

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

nel ramo else 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. Una dichiarazione di un'equivalenza fra game permette di dare a CryptoVerif una regola di riscrittura che può valere a meno di una certa probabilità. Affermando che un game è equivalente ad un altro a meno di una certa probabilità, si dà la possibilità al tool di sostituire espressioni che compaiono nel primo game con le equivalenti che compaiono nel secondo, tenendo comunque conto della probabilità in gioco. Per esempio, con il seguente 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 game che permette di definire la funzione `invf` come l'inversa di `f`. Poichè quest'affermazione non vale a meno di una probabilità ma è vera sempre la quantità tra parentesi è zero (fra i due game). Si possono però specificare, come si vedrà successivamente, delle equivalenze fra game che valgono a meno di una certa probabilità.

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 in CryptoVerif non è altro che un processo il quale 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 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`.

Query. Mediante un'istruzione di query è possibile istruire il tool su cosa si vuole venga dimostrato. Per esempio è possibile richiedere al tool di dimostrare l'indistinguibilità di una variabile da un oggetto casuale, provandone così la pseudocasualità. Per esempio, con il seguente 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, specificata da linea di comando mediante l'opzione -lib[7]). Grazie a questa possibilità è possibile dare una volta per tutte alcune definizioni molto utili in crittografia (e.g funzione one-way). Una volta date è possibile utilizzarle in tutti gli schemi crittografici che le utilizzano semplicemente specificando il file di libreria corretto.
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 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 disposizioni 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 al

[7]Per maggiori informazioni si può leggere il manuale del tool liberamente scaricabile dallo stesso url del tool.

precedente calcolo di processi che si può trovare in [MRST06] quest'ultimo ispiratosi a sua volta a [AG99] introducendo però la fondamentale 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>
```

Si noti la terza produzione in cui si specifica che un oracolo può essere l'oracolo *nullo* ovvero che non fa niente. O ancora, un oracolo può essere la composizione parallela di più oracoli o la replica limitata (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 descritto come questi siano costruiti.
Un oracolo come già detto è un processo e come tale può fare delle operazioni. 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 un 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 valori all'interno di un array di valori.

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> ]
```

Si noti come dopo un *return* mediante il simbolo di sequenzializzazione 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 funzione (eventualmente applicata a degli argomenti).
- una tupla di valori.

Per esempio, con il codice:

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

non si fa altro che assegnare ad x (il cui tipo viene inferito dal tipo di ritorno di A) il valore che la funzione A ritorna su input r per poi ritornare il valore x. Si noti che se 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 delel 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.
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 il seguente frammento di codice:

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

Si può notare come venga definito il corpo dell'oracolo $MyOracle$. Questo per prima cosa assegna alla variabile r un valore scelto casualmente dall'insieme dei valori di A, successivamente assegna alla variabile x il valore resitutito dalla funzione f applicato al valore r, infine 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 $find$ (come si afferma anche in [BP06] è probabilmente il maggior apporto fornito da questo tool rispetto ad altri linguaggi che lavorano invece su liste) che permette di effettuare un lookup su un array alla ricerca di un elemento che soddisfa particolari proprietà, e compiere azioni diverse a seconda che questo elemento venga trovato o meno. Un tipico esempio di utilizzo del costrutto $find$ può essere analizzato nel 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. Quando viene chiamato effettua

un test e 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 preceddentemente assegnato un valore) e tale per cui l'iesimo elemento dell'array ha un valore uguale al valore 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 elmenti che verifichino diverse condizioni. Se diverse condizioni vengono verificate e quindi c'è la possibilità che debbano essere eseguiti diversi rami *then* ognuno associato con condizioni diverse allora CryptoVerif ne sceglie uno casualmente ed esegue quello.

2.2 Un Esempio: FDH

Full Domain Hash è uno schema di firma che segue il paradigma *hash-and-sign*[8]. 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 un'hash di una stringa di bit.
L'input che si fornisce al CryptoVerif è costituito, fondamentalmente, da due parti:

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

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

```
foreach iH <= qH do
  OH(x : bitstring) := return(hash(x)) |
  Ogen() := r <-R seed;
    pk <- pkgen(r );
    sk <- skgen(r );
    return(pk ); (
```

[8]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

```
    foreach iS <= qS do
         OS(m : bitstring) := return(invf(sk , hash(m))) |
         OT (m : bitstring, s : D) := if f(pk , s) = hash(m ) then
             find u <= qS suchthat (defined(m[u]) && m = m[u]) then
                  end
             else
                  event forge
)
```

Procediamo adesso con una breve spiegazione di questo game. Possiamo vedere come in questo game si forniscano qH copie dell' oracolo OH le quali ritornano l'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 $m^{'}$.
In particolare se $hash(m^{'}) \neq f(pk, s)$ allora la firma non è valida per il messaggio. Se invece $hash(m^{'}) = f(pk, s)$, l'oracolo si occupa di verificare se, per il messaggio $m^{'}$, è stata mai rilasciata una firma dall'oracolo OS. In caso affermativo il processo termina, altrimenti significa che 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 game.
Per poter trasformare un game 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 possono essere viste come regole di riscrittura delle distribuzioni di probabilità delle variabili in gioco. Queste regole di riscrittura di un generico elemento L in un generico elemento R, possono 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 games; di solito però questa differenza introdotta è rilevata dall'avversario solo con probabilità trascurabile oppure

con probabilità trascurabilmente vicina a qualche costante come per esempio $\frac{1}{2}$.
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 2.1 *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.*

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, sebbene la maggior parte degli informatici ne sia convinta, non è stata ancora dimostrata l'esistenza di funzioni di questo genere[9]. La dimostrazione dell'esistenza di funzioni one-way comporterebbe, tra l'altro, anche la risoluzione della famosa questione riguardo agli insiemi P e NP[10]. Il fatto che però qualcuno un giorno possa dimostrare che $P \neq NP$ non dimostrerebbe affatto l'esistenza delle funzioni one-way. Infatti una funzione one-way deve, come prima sottolineato, essere non invertibile in modo efficiente, sempre e non solo nel *caso pessimo*.
Siamo ora pronti per vedere come il concetto di funzione one-way venga modellato in CryptoVerif. Si tratta di dare una definzione di one-wayness per mezzo di un'equivalenza fra games. La definizione è la seguente:

```
equiv
       foreach iK <= nK do r <-R seed; (
       Opk() := return(pkgen(r)) |
       foreach iF <= nF do x <-R D;
```

[9] Questo non toglie che esistano funzioni che si avvicinino all'idea che abbiamo di funzioni one-way, per esempio SHA1, o MD5.

[10] 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.

```
          (Oy() := return(f(pkgen(r), x)) |
           foreach i1 <= n1 do Oeq(x' : D) := return(x' = x) |
           Ox() := return(x)))

<=( nK * nF * POW(time + (nK-1) * time(pkgen)
      + (#Oy-1) * time(f)) )=>

    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))).
```

Questa equivalenza fra games 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 game a sinistra e quello a destra del simbolo $\approx_{pow}$ sono indistinguibili. Supponiamo infatti che la funzione f non sia one-way, esisterà dunque un avversario efficiente A che può invertire f. A sarà dunque in grado di distinguere i due membri dell'equivalenza $L \approx_{pow} R$. Questo perché potendo invertire f, A sarà in grado di osservare comportamenti diversi fra i game 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 game, e quindi può distinguerli. Supponiamo, invece, che f sia effettivamente one-way. Allora non esisterà nessun attaccante efficiente che riesca ad invertire f. L'unico modo che ha quindi un attaccante A per invertire la funzione è chiamare Ox() e ottenere così una preimmagine valida. Se però l'attaccante chiama Ox() allora L e R sono effettivamente indistinguibili (a meno di 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. Di questa probabilità viene comunque tenuto conto nell'equivalenza. L'equivalenza infatti, vale a meno di una certa probabilità *pow* (trascurabile in questo caso). In questo modo i due games, 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 al CryptoVerif è quella di funzione hash. Questa definizione viene fornita al tool attraverso il seguente codice:

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

La funzione hash è intesa implementata nel modello dell'oracolo random: se la funzione non è mai stata richiamata su un particolare valore x_0 allora viene 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 look up nell'array. Infine abbiamo una semplice teoria equazionale per descrivere alcune proprietà delle funzioni in gioco. Per esempio:

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

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

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

modellano l'iniettività delle funzioni $invf$, f', f[11]. 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 in [BP] troviamo l'ouput completo di CryptoVerif sull'input descritto precedentemente.

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

Capitolo 3

Risultati Raggiunti

Scopo principale di questo lavoro è la dimostrazione mediante CryptoVerif di un risultato classico della crittografia computazionale. Dopo un'attenta analisi si è giunti alla conclusione che, a causa dell'insufficiente sistema di tipi di CryptoVerif, non sarebbe stato possibile mediante questo dimostrare il risultato nella sua generalità; ecco quindi che si è scelto di utilizzare il tool per dimostrare una caso particolare del teorema.

3.1 Teorema

Il risultato nella sua generalità è il seguente teorema:

Teorema 3.1 *Se esiste un generatore pseudocasuale* $G' : \{0,1\}^n \to \{0,1\}^{n+1}$ *con fattore d'espansione* $l'(n) = n + 1$ *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* di questo teorema in [KL07]. La dimostrazione classica procede....

Si è invece provato il risultato con CryptoVerif per un numero $p(n)$ che è una costante. In particolare mediante CryptoVerif si è provato che:

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

3.2 L'input

Il codice che verrà descritto permette a CryptoVerif di dimostrare che se si ha a disposizione un generatore pseudorandom con fattore di espansione $n+1$ allora si può avere un generatore pseudorandom con fattore di espansione $n+3$. 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].
4 type np3bits [fixed].
```

Queste linee di codice servono per dichiarare dei tipi. Alla linea 1 dichiariamo il tipo delle stringhe di lunghezza n, ovvero definiamo il tipo *nbits*. 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 e 4 lo stesso viene fatto per le stringhe casuali appartenenti agli insiemi $\{0,1\}^{n+2}$, e $\{0,1\}^{n+3}$. Si noti come ogni tipo sia dichiarato con il modificatore $fixed$. Infatti CryptoVerif permette una scelta casuale all'interno di un dominio se e solo se questo è dichiarato come fixed. 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. Quelle che seguono sono delle dichiarazioni di simboli di funzioni con i relativi tipi di input e di output.

```
7 fun concatnp1(np1bits,bool):np2bits.
8 fun concatnp2(np1bits,bool, bool):np3bits.
```

In particolare la prima 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 $n+2$. La seconda funzione invece prende in input una stringa di $n+1$ bits e due bits, anche questi rappresentati mediante valori booleani. Il risultato di questa funzione sarà una stringa di lunghezza $n+3$. Come si può intuire dai nomi delle funzioni, queste rappresentano la concatenazione di stringhe di lunghezza $n+1$ con uno e due bits rispettimavamente. Mediante le seguenti equivalenze non facciamo altro che implementare le funzioni dichiarate prima. Per esempio la

funzione concatnp1 viene descritta mediante un'equivalenza che rappresenta il seguente asserto: la concatenazione di una stringa casuale di $n+1$ bits con un bit casuale è una stringa casuale di $n+2$ bits.

```
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).
```

Una definizione simile è stata data per la funzione *concatpn*2:

```
20 equiv
21     foreach i1<=n1 do
22             r<-R np1bits;
23             b'<-R bool;
24             b''<-R bool;
25             OGet():=return(concatnp2(r, b'', b'))
26     <=(0)=>
27     foreach i1<=n1 do
28             w<-R np3bits;
29             OGet():=return(w).
```

Infatti concatenare due bits casuali ad una stringa casuale di lunghezza $n+1$ è equivalente a scegliere in maniera casuale una stringa dall'insieme $\{0,1\}^{n+3}$. 31 fun getn(np1bits):nbits. 32 fun getlast(np1bits):bool. 33 equiv 34 foreach i1¡=n1 do 35 r ¡-R np1bits;(36 OGetn():=return (getn(r)) — 37 OGetlast():=return (getlast(r))) 38 ¡=(0)=¿ 39 foreach i1 ¡=n1 do 40 (41 OGetn():= w ¡-R nbits;return(w) — 42 OGetlast():=wl ¡-R bool;return(wl) 43). 44 fun getn1(np2bits):nbits. 45 fun getlast1(np2bits):bool. 46 equiv 47 foreach i1¡=n1 do 48 r ¡-R np2bits;(49 OGetn():=return (getn1(r)) — 50 OGetlast():=return (getlast1(r))) 51 ¡=(0)=¿ 52 foreach i1 ¡=n1 do 53 (54 OGetn():= w ¡-R nbits;return(w) — 55 OGetlast():=wl ¡-R bool;return(wl) 56). 57 58 fun G'(nbits): np1bits. 59 equiv 60 foreach i1¡=n1 do 61 r ¡-R nbits; 62 OGet():=return (G'(r)) 63 ¡=(0)=¿ (* To define *) 64 foreach i1 ¡=n1 do 65 w ¡-R np1bits; 66 OGet():=return(w). 67 68 query secret1 w3. 69 70 71 process 72 O():= 73 r¡-R nbits; 74 let x' = G'(r) in 75 let y' = getn(x') in 76 let b' = getlast(x') in 77 let x"=G'(y') in 78 let y"=getn(x") in 79 let b"=getlast(x") in 80 w3:np3bits ¡-concatnp2(G'(y"), b", b'); 81 return

3.3 L'output

L'output di CryptoVerif è il seguente:

```
Doing expand if, let, find... No change.
Doing simplify... Run simplify 1 time(s). Fixpoint reached.
No change.
Doing move all binders... No change.
Doing remove assignments of useless... No change.
Trying equivalence
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
... Succeeded.
Doing simplify... Run simplify 1 time(s). Fixpoint reached.
No change.
Doing move all binders... No change.
Doing remove assignments of useless... Done.
Trying equivalence
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
... Failed.
Trying equivalence
equiv foreach i1_7 <= n1 do r <-R np2bits; (
  OGetn() := return(getn1(r)) |
  OGetlast() := return(getlast1(r)))
<=(0)=>
      foreach i1_8 <= n1 do w <-R nbits; wl <-R bool; (
  OGetn() := return(w) |
  OGetlast() := return(wl))
... Failed.
Trying equivalence
equiv foreach i1_5 <= n1 do r <-R np1bits; (
  OGetn() := return(getn(r)) |
  OGetlast() := return(getlast(r)))
<=(0)=>
      foreach i1_6 <= n1 do w <-R nbits; wl <-R bool; (
  OGetn() := return(w) |
  OGetlast() := return(wl))
... Succeeded.
```

```
Doing simplify... Run simplify 1 time(s). Fixpoint reached.
No change.
Doing move all binders... No change.
Doing remove assignments of useless... Done.
Trying equivalence
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
... Succeeded.
Doing simplify... Run simplify 1 time(s). Fixpoint reached.
No change.
Doing move all binders... No change.
Doing remove assignments of useless... Done.
Trying equivalence
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
... Failed.
Trying equivalence
equiv foreach i1_7 <= n1 do r <-R np2bits; (
  OGetn() := return(getn1(r)) |
  OGetlast() := return(getlast1(r)))
<=(0)=>
      foreach i1_8 <= n1 do w <-R nbits; wl <-R bool; (
  OGetn() := return(w) |
  OGetlast() := return(wl))
... Failed.
Trying equivalence
equiv foreach i1_5 <= n1 do r <-R np1bits; (
  OGetn() := return(getn(r)) |
  OGetlast() := return(getlast(r)))
<=(0)=>
      foreach i1_6 <= n1 do w <-R nbits; wl <-R bool; (
  OGetn() := return(w) |
  OGetlast() := return(wl))
... Succeeded.
Doing simplify... Run simplify 1 time(s). Fixpoint reached.
No change.
Doing move all binders... No change.
Doing remove assignments of useless... Done.
Trying equivalence
```

```
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
... Succeeded.
Doing simplify... Run simplify 1 time(s). Fixpoint reached.
No change.
Doing move all binders... No change.
Doing remove assignments of useless... No change.
Trying equivalence
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
... Failed.
Trying equivalence
equiv foreach i1_7 <= n1 do r <-R np2bits; (
  OGetn() := return(getn1(r)) |
  OGetlast() := return(getlast1(r)))
<=(0)=>
      foreach i1_8 <= n1 do w <-R nbits; wl <-R bool; (
  OGetn() := return(w) |
  OGetlast() := return(wl))
... Failed.
Trying equivalence
equiv foreach i1_5 <= n1 do r <-R np1bits; (
  OGetn() := return(getn(r)) |
  OGetlast() := return(getlast(r)))
<=(0)=>
      foreach i1_6 <= n1 do w <-R nbits; wl <-R bool; (
  OGetn() := return(w) |
  OGetlast() := return(wl))
... Failed.
Trying equivalence
equiv foreach i1_3 <= n1 do r <-R np1bits; b' <-R bool; b'' <-R bool; OGet()
<=(0)=>
      foreach i1_4 <= n1 do w <-R np3bits; OGet() := return(w)
... Succeeded.
Doing simplify... Run simplify 1 time(s). Fixpoint reached.
No change.
Doing move all binders... No change.
Doing remove assignments of useless... No change.
Proved one-session secrecy of w3
```

```
==================== Proof starts ======================
Initial state
Game 1 is
O() :=
r <-R nbits;
x': np1bits <- G'(r);
y': nbits <- getn(x');
b': bool <- getlast(x');
x'': np1bits <- G'(y');
y'': nbits <- getn(x'');
b'': bool <- getlast(x'');
w3: np3bits <- concatnp2(G'(y''), b'', b');
return()


Applying equivalence
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
 yields

Game 2 is
O() :=
w_11 <-R np1bits;
x': np1bits <- w_11;
y': nbits <- getn(x');
b': bool <- getlast(x');
x'': np1bits <- G'(y');
y'': nbits <- getn(x'');
b'': bool <- getlast(x'');
w3: np3bits <- concatnp2(G'(y''), b'', b');
return()


Applying remove assignments of useless yields

Game 3 is
O() :=
w_11 <-R np1bits;
y': nbits <- getn(w_11);
b': bool <- getlast(w_11);
```

```
x'': np1bits <- G'(y');
y'': nbits <- getn(x'');
b'': bool <- getlast(x'');
w3: np3bits <- concatnp2(G'(y''), b'', b');
return()


Applying equivalence
equiv foreach i1_5 <= n1 do r <-R np1bits; (
  OGetn() := return(getn(r)) |
  OGetlast() := return(getlast(r)))
<=(0)=>
      foreach i1_6 <= n1 do w <-R nbits; wl <-R bool; (
  OGetn() := return(w) |
  OGetlast() := return(wl))
 yields

Game 4 is
O() :=
wl_13 <-R bool;
w_12 <-R nbits;
y': nbits <- w_12;
b': bool <- wl_13;
x'': np1bits <- G'(y');
y'': nbits <- getn(x'');
b'': bool <- getlast(x'');
w3: np3bits <- concatnp2(G'(y''), b'', b');
return()


Applying remove assignments of useless yields

Game 5 is
O() :=
wl_13 <-R bool;
w_12 <-R nbits;
x'': np1bits <- G'(w_12);
y'': nbits <- getn(x'');
b'': bool <- getlast(x'');
w3: np3bits <- concatnp2(G'(y''), b'', wl_13);
return()
```

```
Applying equivalence
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
 yields

Game 6 is
O() :=
wl_13 <-R bool;
w_14 <-R np1bits;
x'': np1bits <- w_14;
y'': nbits <- getn(x'');
b'': bool <- getlast(x'');
w3: np3bits <- concatnp2(G'(y''), b'', wl_13);
return()


Applying remove assignments of useless yields

Game 7 is
O() :=
wl_13 <-R bool;
w_14 <-R np1bits;
y'': nbits <- getn(w_14);
b'': bool <- getlast(w_14);
w3: np3bits <- concatnp2(G'(y''), b'', wl_13);
return()


Applying equivalence
equiv foreach i1_5 <= n1 do r <-R np1bits; (
  OGetn() := return(getn(r)) |
  OGetlast() := return(getlast(r)))
<=(0)=>
      foreach i1_6 <= n1 do w <-R nbits; wl <-R bool; (
  OGetn() := return(w) |
  OGetlast() := return(wl))
 yields
```

```
Game 8 is
O() :=
wl_13 <-R bool;
wl_16 <-R bool;
w_15 <-R nbits;
y'': nbits <- w_15;
b'': bool <- wl_16;
w3: np3bits <- concatnp2(G'(y''), b'', wl_13);
return()


Applying remove assignments of useless yields

Game 9 is
O() :=
wl_13 <-R bool;
wl_16 <-R bool;
w_15 <-R nbits;
w3: np3bits <- concatnp2(G'(w_15), wl_16, wl_13);
return()


Applying equivalence
equiv foreach i1_9 <= n1 do r <-R nbits; OGet() := return(G'(r))
<=(0)=>
      foreach i1_10 <= n1 do w <-R np1bits; OGet() := return(w)
 yields

Game 10 is
O() :=
wl_13 <-R bool;
wl_16 <-R bool;
w_17 <-R np1bits;
w3: np3bits <- concatnp2(w_17, wl_16, wl_13);
return()


Applying equivalence
equiv foreach i1_3 <= n1 do r <-R np1bits; b' <-R bool; b'' <-R bool; OGet()
<=(0)=>
      foreach i1_4 <= n1 do w <-R np3bits; OGet() := return(w)
```

```
 yields

Game 11 is
O() :=
w_18 <-R np3bits;
w3: np3bits <- w_18;
return()


RESULT Proved one-session secrecy of w3
All queries proved.
```

Capitolo 4

Conclusioni

Bibliografia

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

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

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

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

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

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

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

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

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

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

[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`.