Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

SULLA SICUREZZA DI ALCUNI PROTOCOLLI PER IL VOTO CRITTOGRAFICO

Relatore:
Chiar.mo Prof.
UGO DAL LAGO

Presentata da:
GIULIA BACCOLINI

Sessione III
Anno Accademico 2015-2016

Alla mia famiglia.

Capitolo 1

Introduzione

A seguito del costante ed attuale sviluppo della tecnologia, in alcuni Stati, come ad esempio la Finlandia [3] e la Svizzera [4], è nata l'idea di trasformare il tradizionale sistema elettorale in uno elettronico, sfruttando la tecnologia moderna in modo tale da consentire agli elettori di esprimere il proprio voto restando nella propria abitazione, ove ci sia una connessione ad Internet. Infatti, uno dei problemi principali più comuni registrati ad ogni elezione è la scarsa affluenza alle urne. Ma la tecnologia è veramente in grado di stimolare la partecipazione dei cittadini alle elezioni? Alcuni esperti affermano di sì (si veda [Cap08]), poiché il voto elettronico facilita la sua espressione a chi non si reca alle urne perché fisicamente impossibilitato o perché momentaneamente all'estero. Oltre che per consentire l'espressione del voto senza doversi recare ai seggi, l'idea di sperimentare questo nuovo sistema elettorale è nata per garantire una maggiore sicurezza in termini di privacy, eleggibilità, verificabilità ed accuratezza. Infatti, il sistema elettorale elettronico dovrebbe garantire, rispettivamente, che il voto espresso da ogni elettore rimanga segreto, che tutti i voti validi vengano conteggiati correttamente nel risultato finale, che ogni elettore possa assicurarsi personalmente che il proprio voto venga preso in carico, che sia possibile controllare l'idoneità al voto di ogni elettore e la non alterazione o eliminazione di un voto valido [Adi08]. Questo nuovo sistema, inoltre, dovrebbe consentire all'organo elettorale di risparmiare tempo, denaro e di arginare i problemi legati agli inevitabili errori umani, quali, ad esempio, l'errato conteggio dei voti o la difficoltà nell'interpretare la scrittura degli elettori. Oltre alla necessità di garantire una maggiore sicurezza dell'intero sistema elettorale, altri motivi stanno spingendo diversi paesi verso questo *radicale* cambiamento: nel sistema tradizionale, forma, dimensione, colore, distanza tra i simboli e la loro disposizione nello spazio della scheda elettorale sono diventati elementi di contestazione, insieme ad una serie di valutazioni tattiche e logistiche circa le regole di arruolamento degli scrutatori e dei rappresentanti di lista, il numero dei seggi e delle sezioni elettorali, la loro

collocazione sul territorio, e così via. Nello specifico, la composizione grafica delle schede elettorali è stata oggetto di studio e contestazioni per diversi anni: la posizione dei simboli dei partiti e dei nomi dei candidati è stata considerata una variabile influente sul risultato finale delle elezioni. Infine, si pensava che il dover garantire la sicurezza in termini di privacy, verificabilità, eleggibilità e accuratezza (spiegata in precedenza), ricadesse in primo luogo sugli scrutatori e sul presidente, la cui selezione rappresentava, come già accennato, un nodo cruciale nell'assicurare l'imparzialità dello scrutinio e quindi la sua legittimazione politica e sociale [Cap08]. Un ultimo fondamentale problema è la coercizione, che deve assolutamente essere presa in considerazione quando si parla di sicurezza del sistema elettorale tradizionale: se un'entità malintenzionata riesce a costringere un elettore a votare in un certo modo (talvolta anche offrendo in cambio del denaro), il risultato delle elezioni sarà sicuramente alterato. Nel sistema elettorale elettronico la coercizione è un problema meno rilevante (ma non inesistente), poiché gli elettori votano in un luogo privato (solitamente da soli o, al massimo, con membri della famiglia) ed è quindi molto difficile corromperli nel momento esatto della votazione [AD+09]. Tuttavia, ci sono molti altri momenti nel quotidiano in cui un elettore può essere corrotto: è per questo motivo che anche nel sistema elettorale elettronico il problema della coercizione, seppur meno rilevante, dovrebbe essere preso in considerazione.

Data la grande importanza di garantire la sicurezza del sistema elettorale, nonché la mitigazione dei problemi appena trattati, alcuni paesi tra cui Estonia e Norvegia (dettagli nella prossima sezione), hanno ritenuto necessaria l'introduzione di un nuovo sistema elettorale, di carattere elettronico. Era, inoltre, fondamentale fornire la descrizione di un protocollo che lo rappresentasse e la dimostrazione formale delle proprietà di sicurezza ad esso legate, anche in accordo con quanto introdotto dall'avvento della crittografia moderna nel XX secolo. Il protocollo *Helios*, descritto in [Adi08], è tra i primi protocolli di voto elettronico ad essere stato sperimentato e reso disponibile all'utilizzo (si faccia riferimento a [1]). Questo protocollo è stato costruito sulla base del protocollo di voto elettronico *Minivoting*, oggetto di questo lavoro di tesi. Per definire *sicuri* questi protocolli, è necessario dare una dimostrazione rigorosa e formale delle loro proprietà di sicurezza. Tale dimostrazione può essere costruita sia "su carta" che mediante strumenti automatici o semiautomatici. Le dimostrazioni "su carta" sono ritenute sufficienti per provare le proprietà di sicurezza di un protocollo; nonostante ciò, sono spesso soggette ad errori. I *tool* automatici e semiautomatici, invece, sono molto più precisi e utili per costruire dimostrazioni molto più solide.

Sono tanti gli strumenti disponibili che consentono di realizzare tali dimostrazioni. Tra tutti, questo lavoro si concentra su due in particolare, `CryptoVerif` ed `EasyCrypt`, che consentono, come richiesto, di descrivere protocolli crittografici e di dimostrarne particolari proprietà di sicurezza. Hanno potere espressivo diverso e si comportano in maniera differente, perciò

è utile gestire le dimostrazioni sfruttandoli entrambi. Il primo, `CryptoVerif`, è un dimostratore automatico: si comporta come *black box*, riceve il protocollo e la proprietà di sicurezza da dimostrare in input e restituisce una sequenza di giochi che dimostrano quest'ultima. Proprio perchè si comporta come una *scatola nera*, il suo potere espressivo è minore rispetto a quello di `EasyCrypt`, ma è comunque uno strumento ritenuto efficace nel gestire le dimostrazioni. Il secondo, `EasyCrypt`, è un dimostratore semiautomatico che richiede un input maggiormente dettagliato: oltre al protocollo ed alla proprietà di sicurezza, prende in input anche i giochi della prova. Come detto, questo strumento è più solido e potente del precedente. Nel corso degli anni 2000, l'utilizzo di questi due dimostratori ha permesso di descrivere formalmente protocolli crittografici (ad esempio, ElGamal e FDH), primitive crittografiche (come lo schema di codifica a chiave pubblica) ma anche algoritmi matematici (ad esempio PRG, generatori pseudo casuali), e di dimostrarne l'effettiva sicurezza.

Come discusso precedentemente, questo lavoro di tesi si dedica alla dimostrazione formale della sicurezza di un protocollo di voto elettronico, *Minivoting*, nel modello computazionale. Precisiamo che nessun argomento di questo lavoro sarà trattato nel modello formale di *Dolev-Yao*: ognuno di essi verrà discusso esclusivamente nel modello computazionale. *Minivoting* sarà trattato, illustrato ed analizzato in diversi ambiti: inizialmente verrà posta l'attenzione sulle primitive crittografiche alla base del protocollo stesso, tra cui, ad esempio, lo schema di crittografia asimmetrica omeomorfa; successivamente verrà descritto, dettagliatamente ed in modo formale, il protocollo vero e proprio; in seguito si darà una dimostrazione, inizialmente informale anche se precisa, della sua sicurezza. Infine, verranno studiati, analizzati e sfruttati i dimostratori `CryptoVerif` e `EasyCrypt` per dimostrare, in maniera rigorosa e formale, la sicurezza del protocollo in analisi, partendo dalla descrizione, a livello di codice, di quest'ultimo. Oltre a *Minivoting* nella sua versione più semplice, verranno presentate altre due versioni, ossia *Minivoting Threshold* e *Minivoting Zero-Knowledge*, che non verranno però analizzate formalmente mediante i dimostratori `CryptoVerif` e `EasyCrypt`. Il lavoro di tesi qui presentato è organizzato nel modo che segue:

- a questa breve introduzione fa seguito, a conclusione del Capitolo 1, una discussione sullo stato dell'arte relativo ai protocolli di voto elettronico e alla loro adozione nei sistemi elettorali di alcuni Stati Europei;

- segue il Capitolo 2, contenente le descrizioni del sistema elettorale tradizionale e del voto elettronico, corredate dall'illustrazione dei problemi ad essi legati, insieme ad alcuni cenni storici riguardanti i sistemi di voto elettronico e meccanico;

- il Capitolo 3 illustra diversi concetti della crittografia moderna, quali l'avvento della figura dell'avversario ed il modello computazionale, insieme alla descrizione formale dettagliata delle primitive e degli schemi crittografici su cui si basa *Minivoting*;

- il Capitolo 4 si dedica interamente alla descrizione formale del protocollo di voto elettronico *Minivoting*, prima in versione semplice, poi nelle due versioni leggermente più complesse.

- nel Capitolo 5 troviamo la parte sperimentale del lavoro legata a `CryptoVerif`. Il capitolo inizia con la presentazione di quest'ultimo, seguita dalla descrizione delle strutture e della sintassi relativa al linguaggio usato dal dimostratore. Infine vengono presentati, nell'ordine, la descrizione del protocollo *Minivoting* in `CryptoVerif`, i giochi della dimostrazione della sua sicurezza, ed i risultati ottenuti corredati da alcuni sviluppi futuri;

- il Capitolo 6, similmente al precedente, tratta la parte sperimentale del lavoro legata ad `EasyCrypt`. Inizia presentando il dimostratore, il suo linguaggio, il proof assistant e la logica su cui si basa. Seguono la descrizione di *Minivoting* in `EasyCrypt`, i giochi della dimostrazione della sua sicurezza, ed i risultati finali ottenuti. Questo capitolo è seguito dalle conclusioni relative all'intero lavoro svolto.

1.1 Stato dell'Arte

Nel XX secolo sono stati molti i tentativi di aumentare l'affluenza dei cittadini alle elezioni. L'idea più efficace è stata l'introduzione del sistema di voto elettronico. L'avvento di questo nuovo sistema si è tradotto presto in risparmio di tempo e riduzione di costi, portando così ad una maggiore partecipazione dei cittadini alle elezioni. Tuttavia, mentre i problemi di affidabilità del sistema (come, per esempio, l'alterazione di un voto dovuta ad un *malware* presente nel dispositivo personale usato dall'elettore per votare) devono ancora essere del tutto risolti, le tecniche crittografiche alla base del nuovo sistema elettorale sono risultate potenti ed ottimali nella dimostrazione della sicurezza del sistema stesso. Sebbene al giorno d'oggi la maggior parte dei protocolli di voto elettronico esistenti siano considerati sicuri e siano adottati dal sistema elettorale di alcuni Stati Europei, nel modello crittografico computazionale non esiste una prova rigorosa e formale della loro effettiva sicurezza [BW14]. Al contrario, questa viene data nel modello crittografico formale. Ne troviamo un esempio in [Cor15], dove viene formalmente analizzato un protocollo di voto elettronico teorico, detto *FOO92* [FO+92], non prima di averne definito le proprietà di sicurezza desiderate, in particolare quella legata alla privacy: un sistema elettorale

elettronico dovrebbe garantire che il voto espresso da ogni elettore rimanga segreto; per ottenere ciò, la maggior parte di questi sistemi prevede di cifrare il voto con la chiave pubblica e di inviare poi il crittogramma alle autorità. Tuttavia, siccome gli elettori votano dal proprio dispositivo, è difficile garantire protezione da entità malintenzionate: una contromisura è usare un *code-sheet* per nascondere il voto all'interno del dispositivo e renderlo meno accessibile. Nonostante si possano descrivere e dimostrare molte proprietà nel modello formale, gli strumenti che esso mette a disposizione (come ad esempio `Avispa` [AB+05], oppure `Schyter` [Cre08]) non sono abbastanza potenti per dimostrare le proprietà più complesse, oppure per descrivere ed implementare le primitive crittografiche necessarie. Il protocollo *FOO92*, ad esempio, si basa, tra gli altri, sullo schema di firma *blinded*. Questo schema, descritto in [BW14], è troppo complesso per alcuni strumenti come `APTE` [Che14], che non è in grado di implementarlo. Perciò, il modello formale consente sempre di dare una dimostrazione rigorosa teorica della sicurezza di un protocollo di voto elettronico (si faccia, ad esempio, riferimento a [FO+92]): talvolta, però, non possiede gli strumenti adatti a darne una a livello pratico [Cor15].

Al giorno d'oggi, come detto, esistono diversi paesi che si affidano al sistema elettorale elettronico. Tra questi, l'Estonia usa un sistema molto semplice: ogni elettore custodisce la propria *ID card*, contenente la chiave per poter applicare la firma digitale. Dopo aver scelto e cifrato il voto, l'elettore lo firmerà utilizzando tale chiave. Essendo molto semplice, questo sistema presenta degli evidenti difetti: ad esempio, gli elettori non possono verificare che il loro voto sia stato ricevuto nè conteggiato nel risultato finale. Una descrizione maggiormente accurata del sistema elettorale elettronico Estone si può trovare in [DF+14]. Il protocollo di voto elettronico usato in Norvegia, invece, è leggermente più complesso. Una volta votato, gli elettori ricevono un SMS contenente un codice utile per verificare la corretta ricezione del voto. Tuttavia, il controllo della correttezza del risultato finale è affidato ad un'entità esterna (approvata dalle autorità). Il protocollo Norvegese si può trovare nel dettaglio in [Gjo11]. Tra i protocolli in fase di sperimentazione e non ancora adottati da alcun sistema elettorale, troviamo *Helios* e *Civitas*. Il primo, costruito partendo dal protocollo *Minivoting* oggetto di questa tesi, è usato per eleggere l'IARC (*International Association for Cryptologic Research*) già dal 2010. Inoltre, è stato sperimentato in Belgio, durante le elezioni del Presidente Universitario di Lovaino [AD+09]. Sfrutta delle primitive crittografiche potenti ed efficaci, che garantiscono privacy e verificabilità. La sua descrizione la si può trovare in [Adi08]. Il secondo è uno dei pochi protocolli elettronici a garantire resistenza alla coercizione. Quando un elettore viene influenzato, questo è invitato a votare usando delle credenziali false, in modo che l'entità malintenzionata non riesca a distinguerle da quelle vere. L'autorità ricevente tale voto ne verificherà l'invalidità per poi scartarlo. Questo protocollo non viene usato in un sistema elettorale reale

a causa della sua elevata complessità, che si registra essere quadratica nel numero dei voti inviati [Cor15].

Capitolo 2

I Sistemi Elettorali

Al giorno d'oggi, il meccanismo delle elezioni può fare affidamento su due sistemi elettorali differenti: quello tradizionale e quello elettronico. Questo capitolo li esamina entrambi, partendo da una discussione sulle problematiche legate al sistema elettorale tradizionale, che porteranno all'introduzione del voto, prima meccanico e poi elettronico, ancor prima del XX secolo. Precisiamo che questo lavoro di tesi si concentra interamente sull'implementazione concreta dei sistemi elettorali, tralasciando tutte le nozioni di Diritto Costituzionale.

2.1 Sistema Elettorale Tradizionale

Le società democratiche sono invariabilmente fondate sul meccanismo delle elezioni. I cittadini sono chiamati a votare, cioè ad esprimere una scelta fra più candidati, più liste o più partiti, oppure a rispondere *si* o *no* ad un certo quesito. Nei sistemi elettorali tradizionali, le elezioni si svolgono in un certo numero di seggi distribuiti sul territorio nazionale, regionale o comunale: l'elettore riceve una scheda cartacea e la compila esprimendo la propria preferenza. Al termine delle votazioni, gli scrutatori esaminano le schede valutandone la correttezza e procedono allo scrutinio finale.
Sebbene continui ad essere utilizzato, il sistema elettorale tradizionale presenta diversi difetti [CC03], collegabili a:

- *Tempo.* Lo scrutinio richiede molto tempo; infatti, dal termine del periodo in cui si può votare alla pubblicazione dei risultati possono passare diversi giorni. L'operazione stessa di voto è lenta, in quanto possono votare soltanto poche persone alla volta, mentre le altre fisicamente presenti al seggio sono costrette ad aspettare.

- *Costo.* Indire le elezioni rappresenta una spesa notevole per la Pubblica Amministrazione dello Stato, a causa del consistente impiego di risorse umane e di attrezzature richieste per allestire i seggi.

- *Errore Umano.* Le operazioni di votazione e, soprattutto, conteggio dei voti, sono soggette ad errori. Spesso capita che i multipli conteggi che vengono effettuati manualmente dagli scrutatori non coincidano tra loro.

- *Ambiguità.* Una scheda compilata a mano dall'elettore può presentare delle ambiguità, come una crocetta fuori posto o un nome scritto con calligrafia difficile da interpretare. Queste possono provocare interpretazioni (e quindi conteggi) diverse da parte di chi esamina le schede.

- *Distanza dal Seggio.* L'obbligo dell'elettore di doversi recare al seggio è una delle cause di astensione dal voto. Si pensi agli anziani, che devono arrivare a piedi al seggio, ma anche al problema degli elettori all'estero, impossibilitati a raggiungere le sedi preposte al voto: possono votare per corrispondenza, ma è una procedura che richiede dispendio di tempo perciò molti tendono ad evitarla.

Il buon funzionamento e l'affidabilità di un sistema elettorale tradizionale dipende quindi dalla corretta amministrazione delle elezioni popolari. Gli elettori dovrebbero ricevere la garanzia che il loro voto sia stato correttamente interpretato e che tutti i voti validi siano stati correttamente conteggiati. Inoltre, dovrebbe essere garantita l'impossibilità di coercizione tra elettori, anche quando gli elettori stessi sono disposti ad essere influenzati. Ognuno dei difetti che abbiamo elencato costituisce un punto in cui il sistema tradizionale deve sicuramente migliorare: come possono gli elettori avere sufficiente fiducia nel sistema ed evitare di incorrere in eventuali coercizioni? La storia racconta di un grande sforzo e di una grande attenzione da parte delle società democratiche per garantire che le elezioni fossero svolte legalmente, senza coercizione o atti fraudolenti; questo, in primo luogo, voleva garantire ad ogni candidato, lista o partito la stessa probabilità degli altri di essere eletto. Inoltre, era molto importante che la fiducia degli elettori nel sistema rimanesse forte e solida. Siccome il buon funzionamento del sistema elettorale tradizionale era (ed è) fragile e complesso, l'apporto di modifiche, anche minime, al sistema stesso ha comportato lunghi dibattiti. Anche tutt'ora questo accade perché basta una lieve *falla* nel sistema elettorale per generare una minaccia alla legalità e alla correttezza delle elezioni [Rub01]. La storia insegna anche che, fino al giorno d'oggi, diverse elezioni sono state manipolate al fine di influenzare il loro esito. Tutto ciò ha contribuito all'idea di un sistema elettorale elettronico che sostituisse quello tradizionale. Vediamo, nella prossima sezione, che cos'è il voto elettronico e quali proprietà di sicurezza dovrebbe rispettare un ipotetico protocollo alla base di un sistema elettorale elettronico.

2.2 Voto Elettronico

Il voto può essere considerato come una transazione tra l'elettore ed il sistema elettorale; il voto elettronico quindi è l'implementazione di questa transazione con strumenti informatici. Il voto elettronico può essere espresso in due modi:

1. *Voto da remoto.* Da casa, dal lavoro, o da una qualsiasi postazione con un collegamento a Internet. Le persone che non hanno a disposizione un collegamento ad Internet possono recarsi in specifici seggi, debitamente attrezzati.

2. *Voto nei tradizionali seggi.* Mediante il sistema di voto tradizionale, esprimendo il voto nei seggi e utilizzando strumenti informatici solo per il conteggio dei risultati [Roy88].

Una potenziale soluzione ai problemi descritti nella precedente sezione è l'introduzione del sistema elettorale elettronico con espressione del voto da remoto. La comodità di votare in qualsiasi momento e in qualsiasi luogo (purché si disponga di una connessione Internet), incrementa il numero di partecipanti alle elezioni. Tuttavia, il sistema di voto elettronico da remoto non è pensato per elezioni in larga scala, a causa dell'impatto che ha la (scarsa) fiducia delle persone nella tecnologia. Descriviamo, infine, le proprietà di sicurezza che un buon protocollo di voto elettronico deve soddisfare per poter essere sicuro e quindi applicabile [CC03]:

- *Eleggibilità*: solo gli elettori che hanno diritto di voto possono votare, e possono farlo solo una volta.

- *Accuratezza*: non deve essere possibile alterare un voto, eliminare dal conteggio finale un voto valido e conteggiarne uno non valido.

- *Privacy*: nessuna autorità può associare una scheda elettorale ad un preciso elettore, e nessun elettore può provare di aver votato in un certo modo.

- *Verificabilità*: gli elettori devono essere sicuri che il proprio voto sia stato conteggiato in modo corretto.

- *Efficienza*: le operazioni di voto devono essere svolte in un tempo ragionevole, e un elettore non deve aspettare che un altro finisca per poter votare. Gli elettori devono poter votare in poco tempo, con la minima attrezzatura necessaria, senza avere particolari conoscenze informatiche.

- *Responsabilità*: possibilità di identificare chi non ha votato. In alcuni paesi è obbligatorio votare, a meno di una valida giustificazione. Tut-

tavia, poiché in alcuni paesi, come ad esempio l'Italia, votare non costituisce un obbligo ma un diritto ed un dovere, consideriamo opzionale questa proprietà.

- *Mobilità*: non ci devono essere restrizioni riguardo al luogo in cui un elettore esprime il proprio voto.
- *Flessibilità*: il sistema deve supportare diversi formati di scheda (preferenze multiple, referendum, risposte aperte, eccetera).
- *Scalabilità*: la dimensione dell'organo votante non deve influire in modo consistente sull'efficienza di tutto il sistema.

Notiamo che il sistema elettorale tradizionale soddisfa le proprietà di eleggibilità, privacy, responsabilità, flessibilità e scalabilità.

2.3 Cenni Storici sui Sistemi di Voto Elettronico e Meccanico

Il primo strumento per votare elettronicamente risale a 150 anni fa, precisamente al 1849, quando Martin de Brettes inventò e realizzò un telegrafo elettronico *decision-making*, che consentiva agli elettori di comunicare, a distanza, il voto scelto alle autorità adibite al raccoglimento, allo spoglio e al conteggio dei voti [AS16]. Tuttavia, il primo strumento per il voto elettronico ad essere stato brevettato fu inventato da Thomas Edison nel 1869: il registratore di voto elettronico [Edi69]. Questo registratore era uno strumento elettronico composto da un piccolo interruttore, che poteva essere spostato in due diverse posizioni, e da un dispositivo in grado di inviare dei segnali. Durante le elezioni, ad ogni elettore veniva fornito uno di questi registratori: l'elettore doveva spostare l'interruttore a destra per votare *sì*, oppure a sinistra per votare *no* (o viceversa). Veniva così inviato un segnale ad un registratore centrale, controllato dalle autorità; la struttura di questo registratore era divisa in due colonne di metallo, una etichettata *sì* e l'altra etichettata *no*; la prima raccoglieva i segnali corrispondenti al voto *sì*, mentre la seconda quelli corrispondenti al voto *no*. Infine, un addetto al controllo del registratore inseriva sotto di esso (a contatto con l'etichetta delle colonne) un foglio trattato chimicamente, su cui il registratore stampava i voti non appena riceveva il segnale corrispondente: quando la corrente passava attraverso la carta, le sostanze chimiche presenti in essa si decomponevano, lasciando trasparire il nome della colonna che aveva ricevuto il segnale. Seppur molto ingegnosa per quegli anni, questa macchina era molto complessa da usare e richiedeva una manutenzione molto accurata; il problema più rilevante, tuttavia, era l'enorme quantità di segnali che il registratore centrale riceveva al secondo, e che non riusciva ovviamente a gestire: la conseguenza fu che i voti venivano stampati uno sopra all'altro o, talvolta, non stampati.

Per tutte queste ragioni, l'invenzione di Edison venne presto accantonata [Kri14].

Nel 1889, a New York, Jacob H. Myers brevettò la prima macchina meccanica a leva per il voto elettronico, e la chiamò *Myers Automatic Booth* [BE+07]. A differenza del registratore inventato da Edison, questa era uno strumento meccanico di conformazione più piccola e semplice, dotato di una leva e di un'interfaccia. L'interfaccia mostrava simultaneamente il quesito elettorale e tutti i nomi dei candidati, delle liste o dei partiti fra cui scegliere: per votare, l'elettore doveva posizionare la leva in corrispondenza della scelta desiderata e, appena usciva dalla cabina elettorale, un meccanismo di conteggio automatico dei voti incrementava di una unità il totale relativo al voto appena espresso. Come miglioria rispetto al registratore elettronico, la macchina meccanica a leva preveniva il sovraccarico nella ricezione dei voti, accelerava il processo di conteggio dei voti e, soprattutto, riduceva la possibilità di brogli, poiché i voti venivano conteggiati dalla macchina stessa (a differenza del sistema precedente, in cui i voti venivano conteggiati dall'addetto al controllo del registratore). Tuttavia, la critica ricorrente alla macchina a leva era che essa non consentiva il riconteggio dei voti, posto che questi venissero registrati esclusivamente in modo aggregato. Non era nemmeno un sistema al sicuro dai tentativi di coercizione. Questo strumento per il voto elettronico venne usato per la prima volta nel 1892 a Lockport, New York. Nel 1930 ci fu il *boom* del suo utilizzo: era sfruttato da tutte le principali città degli Stati Uniti d'America [AS16].

Nonostante la macchina meccanica a leva per il voto elettronico venne adottata dai sistemi elettorali di molti paesi americani, la volontà e la necessità di sanare i suoi punti deboli portò, nel 1960, all'invenzione di un nuovo sistema di voto: le schede perforate [BE+07]. Con questo nuovo sistema, ogni elettore, una volta arrivato al seggio elettorale, riceveva una scheda cartacea in cui erano elencati i candidati, le liste o i partiti fra cui poter scegliere. Per esprimere il proprio voto, l'elettore doveva effettuare diverse perforazioni sulla scheda, seguendo le indicazioni fornitegli dagli scrutatori presenti al seggio. Questo nuovo sistema, tuttavia, ebbe vita breve, poiché era molto difficile per gli scrutatori leggere i tanti piccoli fori su ogni scheda elettorale (e quindi calcolare correttamente e con precisione il risultato delle elezioni). Infatti, nel 1962, nacque il *Precinct Counted Optical Scan* (PCOS), un sistema a scansione ottica che sostituiva il precedente [KK05]. Con questo nuovo sistema, ogni elettore presente al seggio riceveva una scheda elettorale cartacea, che elencava i nomi dei candidati (liste o partiti) alle elezioni: accanto ad ogni nome era raffigurato un *pallino vuoto*. Per esprimere la propria preferenza, l'elettore doveva annerire il *pallino* situato di fianco al candidato scelto. Una volta votato, l'elettore inseriva la scheda cartacea all'interno di uno scan ottico, che controllava eventuali anomalie (come scheda bianca o scheda non annerita negli spazi appositi). Al termine delle elezioni, gli scan ottici di ogni seggio calcolavano automaticamente il

risultato finale. Il sistema PCOS presentava diverse problematiche: l'annerimento parziale di un *pallino* poteva essere interpretato in maniera diversa da scan di seggi diversi; un'entità malintenzionata poteva manomettere gli scan o influenzare gli elettore a votare in un certo modo; in tutti questi casi, il risultato delle elezioni sarebbe stato sicuramente alterato.

A causa degli evidenti problemi delle schede perforate e dei sistemi PCOS, e del costante sviluppo tecnologico, nel 1986 venne abbandonato l'ultilizzo dello scan ottico e introdotto il *Direct Recording by Electronics* (DRE), grazie a John M. Davis e Shelby Thomas [DT96]. Come il precedente, anche questo dispositivo prevedeva che gli elettori, per votare, si recassero ai seggi. Presso ogni cabina elettorale veniva collocato un computer, il cui monitor illustrava una copia fedele delle schede elettorali cartacee. Gli elettori potevano esprimere la propria preferenza sfruttando il *touch screen* a disposizione, oppure utilizzando dei pulsanti posti ai lati del monitor. Espresso il voto, il computer stampava la scheda elettorale con la preferenza indicata, in modo che l'elettore potesse verificarla prima che la scheda stampata venisse inserita automaticamente in un contenitore non più accessibile. Inoltre, il voto espresso veniva salvato dal DRE in una componente di memoria al suo interno. Quando l'elettore usciva dalla cabina, il DRE inviava il voto memorizzato ad un server centrale, adibito al raccoglimento dei voti in maniera aggregata. Al termine delle elezioni, il server centrale effettuava il conteggio finale e forniva il risultato. Sebbene più tecnologico dei precedenti, anche questo nuovo sistema di voto elettronico era soggetto ad attacchi di entità malintenzionate, che potevano infettare le macchine con virus per manipolare e modificare i voti salvati in memoria. Dopo questo sistema vennero sperimentati altri sistemi di voto, come ad esempio il voto inviato mediante posta elettronica. Tuttavia, nessuno dei sistemi di voto elettronico descritti fin'ora garantiva segretezza e incoercibilità: potevano essere usati per elezioni su minima scala, certo non per eleggere istituzioni locali o nazionali. I primi sistemi elettorali per il voto elettronico ad essere considerati affidabili nascono dopo l'avvento della crittografia moderna nel XX secolo. Vediamo perché nel prossimo capitolo (per ulteriori dettagli circa la storia dei sistemi di voto elettronico si veda [AS16] e [Kri14]).

Capitolo 3

Crittografia Moderna

Fino al ventesimo secolo, la crittografia era considerata un'arte. La costruzione di schemi complessi e di algoritmi che li forzassero si basava quasi interamente sulla creatività e sulle capacità personali, poiché a quel tempo non era stata ancora formalizzata la nozione di *buon codice*, ossia alcuna nozione che descrivesse quando un codice potesse considerarsi sicuro. Alla fine del XX secolo, però, l'immagine della crittografia era cambiata completamente. Da semplice arte era diventata *scienza*. L'introduzione di nuovi concetti quali, ad esempio, la complessità computazionale e gli algoritmi probabilistici, fece sì che la crittografia moderna iniziasse a prendere forma. Non ne esiste una definizione formale: la crittografia moderna viene riconosciuta come *"lo studio scientifico di tecniche per proteggere informazioni digitali, transizioni e calcoli distribuiti"* [KL07]. La crittografia è quindi passata da forma d'arte a scienza, e come tale fornisce strumenti atti a proteggere i sistemi usati dalla gente comune in tutto il mondo. Come ogni scienza, anche la crittografia moderna si basa su dei principi:

1. il primo passo nel risolvere qualsiasi problema crittografico è la formulazione rigorosa e precisa della **definizione** di sicurezza;

2. quando la sicurezza di uno schema crittografico si basa su un'**assunzione** non dimostrata, quest'ultima deve essere definita con precisione. Inoltre, tale assunzione deve essere il più semplice possibile;

3. gli schemi crittografici devono essere accompagnati da una rigorosa **dimostrazione** della loro sicurezza, definita secondo il primo principio, rispetto alle ipotesi (se necessarie), come indicato nel principio 2.

3.1 Il Ruolo dell'Avversario

Una delle novità introdotte dalla crittografia moderna è la formalizzazione del ruolo dell'avversario. L'avversario rappresenta l'entità malintenzionata

che cerca di forzare lo schema di codifica al fine di trarre informazioni segrete. L'introduzione dell'avversario è chiaramente una nota *stonata* quando si parla di sicurezza di un cifrario. Tuttavia, nonostante l'avversario sia considerato un pericolo per la segretezza delle informazioni, la sua introduzione nella crittografia moderna è ritenuta fondamentale. Perché? La minaccia dell'avversario ha portato i crittografi a ideare dei cifrari costituiti da algoritmi sempre più potenti e di complessità sempre più elevata. Lo scopo era quello di ridurre la probabilità che un avversario riuscisse a forzare uno schema di codifica ed ottenere informazioni private. La complessità per i crittografi stava nel cercare di costruire schemi difficili da forzare per gli avversari, ma abbastanza facili da utilizzare per le entità coinvolte nello schema stesso. Un esempio pratico è dato da un algoritmo basato sulle funzioni *one-way*: funzioni facili da calcolare, ma complesse da invertire. Auguste Kerckhoffs, crittografo tedesco, divenne famoso per i sei principi che pubblicò in *La Cryptographie Militaire* [Ker83]; il principio più importante, e anche quello su cui si basa questa discussione, è il seguente:

A cipher must be practically, if not mathematically, indecipherable.

Secondo questo principio, un cifrario, per essere considerato sicuro, è sufficiente che sia abbastanza complesso da non poter essere forzato in tempo ragionevole e senza alcuna probabilità di successo rilevante. Perciò, più sono potenti gli strumenti usati dall'avversario, più dovranno essere complessi gli algoritmi del cifrario. Concludiamo questa discussione illustrando gli attacchi ad un cifrario più diffusi, in ordine crescente di pericolosità:

- Ciphertext-only attack (COA). È il tipo di attacco più semplice. L'avversario si limita ad osservare i crittogrammi ed a tentare di determinare il testo in chiaro cifrato inizialmente.

- Known-plaintext attack (KPA). L'avversario conosce una o più coppie testo in chiaro/crittogramma, cifrate con la stessa chiave. Il suo scopo è quello di determinare il testo in chiaro cifrato all'inizio sfruttando le coppie che ha a disposizione.

- Chosen-plaintext attack (CPA). In questo attacco, l'avversario ha la possibilità di ottenere il crittogramma corrispondente a qualsiasi testo in chiaro a sua scelta. Anche in questo caso, l'avversario cercherà di determinare il messaggio cifrato inizialmente.

- Chosen-ciphertext attack (CCA). È l'attacco più pericoloso. All'avversario è data la facoltà di conoscere il testo in chiaro corrispondente a qualsiasi crittogramma di sua scelta. Non può però chiedere di decifrare il crittogramma corrispondente al messaggio cifrato all'inizio. Lo scopo è quello di determinare proprio quest'ultimo messaggio.

Notiamo che i primi due attacchi (COA, KPA) sono attacchi di tipo *passivo*: l'avversario si limita ad osservare il canale di comunicazione, ricevere crittogrammi (e talvolta anche i testi in chiaro corrispondenti) e lanciare il suo attacco. Gli ultimi due (CPA, CCA), invece, sono attacchi di tipo *attivo*: l'avversario può, in maniera adattiva, operare cifrature e decifrature su testi di sua scelta. Per ulteriori approfondimenti si faccia riferimento a [KL07].

Nella prossima parte di questo capitolo verranno introdotti e spiegati nel dettaglio tutti i concetti della crittografia moderna che sono utilizzati in questo lavoro di tesi.

3.2 Modello Computazionale

Nel vasto mondo della crittografia esistono almeno due modi differenti per studiare i protocolli crittografici: il modello **formale** e quello **computazionale**. Il primo, chiamato anche modello di *Dolev-Yao* (autori dei primi articoli in cui venne formalizzato), si basa principalmente sul linguaggio matematico per la definizione idealizzata delle capacità dell'avversario [DY83]. Il secondo, su cui si basa l'intero lavoro presentato in questa tesi, è il modello computazionale (ne troviamo un esempio in [GM84]). Questo modello, criticato da molti ma condiviso da altrettanti, ha una visione più realistica rispetto a quello formale: sostiene che le funzioni crittografiche non debbano essere viste come formule matematiche, ma come veri e propri algoritmi probabilistici e/o deterministici. Una delle più grandi critiche che i sostenitori di questo modello presentano a quello formale è che descrivendo le primitive crittografiche come espressioni matematiche formali assumono, implicitamente, che siano corrette ed inviolabili (poiché lo sono le espressioni stesse); questo non è del tutto esatto, poiché un avversario ha sempre una probabilità (anche minima) di forzare lo schema. Tra i due modelli, quindi, c'è un divario che sembrerebbe incolmabile, ma che in realtà incolmabile non è [AR00]. Descriviamo, in questa sezione, le parti del modello computazionale su cui basiamo il nostro lavoro: parleremo di approccio asintotico (dando la definizione di funzione trascurabile e avversario PPT), indistinguibilità e dimostrazioni *game-based*.

Approccio Asintotico. L'approccio che stiamo per descrivere, è alla base del modello computazionale su cui lavoriamo. In questo approccio (approfondito in [KL07]), sia il tempo in cui lavora l'avversario che la sua probabilità di successo (si vedano le definizioni 3.2 e 3.3) sono funzioni di un parametro n, propriamente detto *parametro di sicurezza*, noto anche all'avversario. Perciò, sia il tempo che la probabilità di successo sono funzioni di questo valore. Più nello specifico:

1. L'efficienza dell'avversario è catturata dalla nozione di tempo polinomiale probabilistico.

2. La probabilità di successo dell'avversario è catturata dalla nozione di funzione trascurabile.

Definizione 3.1. (Algoritmo Probabilistico) *Un algoritmo si dice probabilistico se produce bit casuali, ciascuno indipendentemente uguale a 0 con probabilità* $\frac{1}{2}$ *e uguale a 1 con probabilità* $\frac{1}{2}$.

Definizione 3.2. (Tempo Polinomiale Probabilistico) *Un algoritmo probabilistico si dice lavorare in tempo polinomiale se* $\exists p$ *polinomio tale che l'avversario lavora in tempo* $p(n)$ *su input n, indipendentemente dalle scelte probabilistiche che effettua.*

Un esempio di algoritmo probabilistico che lavora in tempo polinomiale è rappresentato nel test di primalità di Miller-Robin [Ra80] in cui, preso un numero molto grande se ne determina, appunto, la primalità. I risultati di questo test indicano che tale algoritmo fa parte della classe degli algoritmi PPT.
L'avversario con cui ci misuriamo nel nostro lavoro è un'entità che si assume malintenzionata con risorse di calcolo limitate. Questo significa che viene scelto di porre un limite alla sua potenza di calcolo: l'avversario avrà quindi tempo a disposizione limitato e capacità di calcolo finita. Importante è sottolineare che queste risorse saranno parametrizzate sul parametro di sicurezza definito sopra; il nostro avversario, quindi, sarà un avversario PPT. Essendo *passivo*, può soltanto osservare i crittogrammi che transitano nel canale di comunicazione (che nel nostro caso è il *bulletin board*, ma lo vedremo meglio nel prossimo capitolo), senza poterli modificare. Perciò, l'unico attacco possibile è il cosiddetto *Ciphertext-Only-Attack*, in cui può soltanto ottenere testi cifrati.

Definizione 3.3. (Funzione Trascurabile) *Sia* ε*:* $\mathbb{N} \to \mathbb{R}^+$ *una funzione. Si dice che* μ *è* ***trascurabile*** *se e solo se per ogni polinomio p, esiste C* $\in \mathbb{N}$ *tale che* $\forall n > C$*:* $\varepsilon(n) < \frac{1}{p(n)}$.

Una funzione trascurabile, quindi, è una funzione che tende molto rapidamente a 0 (come, ad esempio, la funzione $\frac{1}{2^n}$). In crittografia, questa nozione rappresenta il concetto di scarsa probabilità, intesa come probabilità di un avversario di forzare uno schema. Quindi, al crescere del parametro di sicurezza n in uno schema generico Π, vogliamo che la probabilità di un avversario di forzare Π tenda a 0, in modo da poter considerare sicuro tale schema.

Proposizione. (Chiusura delle Funzioni Trascurabili) *Siano* ε_1 *e* ε_2 *due funzioni trascurabili e p un polinomio.*

1. *La funzione* ε_3 *definita come* $\varepsilon_3(n) = \varepsilon_1(n) + \varepsilon_2(n)$ *è trascurabile.*
2. *Per ogni polinomio* p *positivo, la funzione* ε_4 *definita come* $\varepsilon_4(n) = p(n) \cdot \varepsilon_1(n)$ *è trascurabile.*

Il secondo punto di questa definizione implica che, se un determinato evento accade con probabilità trascurabile in un certo esperimento, allora lo stesso evento accade sempre con probabilità trascurabile se quell'esperimento viene ripetuto un numero polinomiale di volte (questa affermazione si basa sul concetto di *Union Bound*, descritto nella Proposizione A.7. in [KL07]).

Dimostrazione. Dimostriamo entrambi i punti della Proposizione.

1. Siano ε_1, ε_2 due funzioni trascurabili. Dobbiamo dimostrare che per un qualsiasi intero $c \in \mathbb{N}$, riusciamo a trovare n_0 tale che $\forall n \geq n_0$ e $\varepsilon_1(n) + \varepsilon_2(n) \leq n^{-c}$. Consideriamo quindi un $c \in \mathbb{N}$ arbitrario. Siccome vale che $c{+}1 \in \mathbb{N}$ e ε_1, ε_2 sono trascurabili, esistono n_{ε_1} e n_{ε_2} tali che:

   $$\forall n \geq n_{\varepsilon_1},\ \varepsilon_1(n) \leq n^{-(c+1)} \text{ e } \forall n \geq n_{\varepsilon_2},\ \varepsilon_2(n) \leq n^{-(c+1)}.$$

   Scegliamo $n_0 = \max(n_{\varepsilon_1}, n_{\varepsilon_2}, 2)$. Per ogni $n \geq n_0 \geq 2$ varrà:

   $$\varepsilon_1(n) + \varepsilon_2(n) \leq n^{-(c+1)} + n^{-(c+1)} = 2n^{-(c+1)} \leq n \cdot n^{-(c+1)}.$$

   Quindi avremo che $\varepsilon_1(n) + \varepsilon_2 \leq n^{-c}$, che sarà una somma trascurabile.

2. Siano ε_1 una funzione trascurabile e p un polinomio. Dobbiamo dimostrare che per un qualsiasi intero $c \in \mathbb{N}$, riusciamo a trovare n_0 tale che $\forall n \geq n_0$ e $\varepsilon_1(n) \cdot p(n) \leq n^{-c}$.
   Prendiamo un qualunque $d \in \mathbb{N}$ e assumiamo che $\varepsilon_1(n) \leq \frac{1}{n^d}$, $\forall n \geq n_{\varepsilon_1}$. Sappiamo che $\varepsilon_1(n) \leq \frac{1}{n^q}$, $\forall n \geq n_{\varepsilon_1}^q$ e $q \in \mathbb{N}$.
   Scegliamo $n_0 = \max(n_{\varepsilon_1}, n_{\varepsilon_1}^{c+d}, 2)$. Per ogni $n \geq n_0 \geq 2$ varrà:

   $$p(n) \cdot \varepsilon_1(n) \leq n^c \cdot n^{-(c+d)} = n^{-d}$$

   che sarà un prodotto trascurabile.

Indistinguibilità. Un altro concetto utile al nostro modello è quello di *indistinguibilità computazionale* [Ven12], che servirà nella comprensione delle dimostrazioni basate su giochi (illustrate in seguito). Vediamo subito un esempio pratico: se prendiamo due oggetti profondamente diversi tra loro ma che nessuno riesce a distinguere, a rigor di logica questi oggetti sono equivalenti. In crittografia, questo accade se nessun algoritmo PPT li riesce a distinguere. Nella definizione di indistinguibilità computazionale, che daremo in un secondo momento, gli oggetti in questione sono detti *ensembles*.

Definizione 3.4. (Ensemble) *Sia I un insieme numerabile.* $X = \{X_i\}_{i \in I}$ *è un* ***ensemble*** *su I se e solo se è una famiglia di variabili statistiche aleatorie, e* X_i *è una distribuzione di probabilità su un dominio finito.*

Un ensemble è quindi una sequenza finita di distribuzioni di probabilità. $\{X_i\}$ è un ensemble su stringhe di lunghezza *i*. Vogliamo quindi definire quando due ensembles sono indistinguibili ovvero, in parole povere, se ogni algoritmo efficiente che accetta $x \in \{X_i\}$ accetta anche $y \in \{Y_i\}$.

Definizione 3.5. (Indistinguibilità Computazionale) *Due ensembles* $X = \{X_n\}$, $Y = \{Y_n\}$ *sono* ***computazionalmente indistinguibili*** *se e solo se per ogni algoritmo* $\mathcal{D} \in PPT$*(detto distinguitore) esiste* ε *trascurabile tale che:*

$$|Pr[\mathcal{D}(1^n, X_n) = 1] - Pr[\mathcal{D}(1^n, Y_n) = 1]| \leq \varepsilon(n).$$

In questa definizione, $\Pr[D(1^n, X_n) = 1]$ è la probabilità che, scegliendo x dall'ensemble $\{X_n\}$ e dandolo in input al distinguitore $\mathcal{D}$ insieme al parametro di sicurezza, $\mathcal{D}$ produca in output 1. Il parametro di sicurezza come input serve a costringere il distinguitore a lavorare in tempo polinomiale. Due ensembles sono quindi indistinguibili computazionalmente quando la probabilità che un distinguitore riesca a discernere i valori provenienti da un insieme rispetto all'altro è trascurabile.
Ad esempio, siano $X = \{X_n\}_{n \in \mathbb{N}}$ e $U = \{U_n\}_{n \in \mathbb{N}}$ due ensembles. $\{X_n\}$ è uniformemente distribuito su stringhe di lunghezza polinomiale in n, mentre $\{U_n\}$ è uniformemente distribuito su stringhe di lunghezza pari a n. Definiamo inoltre che due ensembles sono *statistically close* quando la loro differenza statistica, calcolata come:

$$\Delta(n) \stackrel{def}{=} \frac{1}{2} \cdot \sum_{\alpha} |\Pr[X_n = \alpha] - \Pr[U_n = \alpha]|$$

è trascurabile. Scegliamo, quindi, X e U tali che non siano *statistically close* e tali che siano indistinguibili in tempo polinomiale. Siccome X e U sono indistinguibili in tempo polinomiale, possiamo definire una funzione f: $\{0,1\}^* \to \{0,1\}$ tale che f valga 1 su X, pur potendo valere 0 su U (cioè, $f(x) = 1$ se e solo se $\Pr[X = x] > 0$). Quindi, siccome X e U hanno comportamenti differenti rispetto a *f*, è impossibile computare f in tempo polinomiale, quindi i due ensembles sono computazionalmente indistinguibili. Per ulteriori dettagli si veda [Gol01].
Siano invece, $\forall n \in \mathbb{N}$, X_n e Y_n due variabili statistiche random: la prima sempre uguale a n, e la seconda uguale a n o $n+1$ con probabilità $\frac{1}{2}$. I due ensembles di riferimento, ossia $X = \{X_n\}_{n \in \mathbb{N}}$ e $Y = \{Y_n\}_{n \in \mathbb{N}}$, sono computazionalmente distinguibili. Un possibile algoritmo distinguitore $\mathcal{D}$ PPT su input (y, 1^n) restituirà 1 se $y = n$, 0 altrimenti. Di conseguenza, avremo che $|\Pr[\mathcal{D}(1^n, X_n) = 1] - \Pr[\mathcal{D}(1^n, Y_n) = 1]| = \frac{1}{2}$, probabilità costante e non trascurabile: i due ensembles sono quindi computazionalmente distinguibili.

Dimostrazioni Game-Based. In crittografia moderna, le dimostrazioni di proprietà di sicurezza sono talvolta strutturate in sequenze di giochi [Sho06]. Questa tecnica, propriamente detta *game-based*, sfrutta i classici giochi in cui un'entità maligna, l'avversario, cerca di forzare uno schema, contrastato da un'entità benigna: lo sfidante. I giochi cambiano in base allo schema e alla proprietà che si vuole dimostrare, ma si possono immaginare a grandi linee come una sequenza di azioni che specificano il comportamento dei partecipanti in gioco. Avversario e sfidante, quindi, possono essere intesi in termini scientifici come processi probabilistici che comunicano tra loro all'interno del gioco, considerato di conseguenza come uno spazio probabilistico. In genere, la definizione di sicurezza è legata ad un evento particolare S: per ogni avversario efficiente, la probabilità che S si verifichi è trascurabile. Il primo passo per costruire la sequenza di giochi che serve nella dimostrazione è quello di modellare il gioco iniziale G_0. Esso corrisponde allo schema iniziale di cui si vuole dimostrare la sicurezza. Si procede effettuando modifiche al gioco G_i, ottenendo un gioco G_{i+1} tale che G_i e G_{i+1} siano indistinguibili computazionalmente (secondo la Definizione 3.5). Le modifiche che si effettuano per passare da un gioco al successivo devono essere computazionalmente irrilevanti: ad esempio, sostituire una variabile casuale con una pseudocasuale è lecito, poichè il gioco risultante è distinguibile dal precedente con probabilità trascurabile. Al termine della sequenza si giunge all'ultimo gioco G_n, in cui l'evento S non può accadere (di conseguenza, l'avversario non ha possibilità di vincere). Possiamo dedurne che l'ultimo gioco è computazionalmente indistinguibile dal primo e, quindi, *la somma di tutte le probabilità con cui l'avversario riesce a distinguere un gioco dal successivo all'interno della sequenza è la probabilità che l'evento S si verifichi nel gioco iniziale.* Si possono leggere ulteriori approfondimenti in [Sho06].

3.3 Crittografia a Chiave Pubblica

La crittografia a chiave pubblica, o asimmetrica, presenta uno schema in cui le entità che devono comunicare hanno due chiavi: la chiave pubblica (accessibile da tutte le entità) e la chiave privata (ogni entità ne possiede una, e la tiene segreta). La chiave pubblica viene facilmente generata in funzione di quella privata, ma il processo inverso è complesso, poiché ha un costo computazionale molto elevato [Sch96]. Prima di essere inviato, il messaggio viene cifrato con la chiave pubblica del destinatario, che decifrerà il crittogramma ottenuto con la propria chiave privata.

Definizione 3.6. Uno schema Π di crittografia a chiave pubblica è una tripla di algoritmi PPT:

$$\Pi = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$$

1. KeyGen è l'algoritmo di generazione delle chiavi pubblica e privata. Esso produce una coppia di chiavi $(sk,vk) \leftarrow \mathsf{KeyGen}(1^n)$, rispettivamente chiave privata e chiave pubblica.

2. Encrypt$(vk,\ m)$ è l'algoritmo di cifratura. Prende in input la chiave pubblica vk ed un messaggio m e produce un crittogramma $c \leftarrow \mathsf{Encrypt}(vk,\ m)$.

3. Decrypt$(sk,\ c)$ è l'algoritmo di decifratura. Prende in input la chiave privata sk e un crittogramma c e produce in output il messaggio decifrato $d \leftarrow \mathsf{Decrypt}(sk,\ c)$, oppure $\perp$ se il crittogramma non è valido. Questo algoritmo deve essere deterministico.

Π richiede che sia rispettato il seguente vincolo di correttezza, che deve valere se $(sk,\ vk)$ sono ottenute tramite l'algoritmo KeyGen:

$$\mathsf{Decrypt}(sk,\ \mathsf{Encrypt}(vk,\ m)) = m$$

Sicurezza di Schemi a Chiave Pubblica. Uno schema di crittografia a chiave pubblica è sicuro se, dati due messaggi ed un crittogramma, è impossibile indovinare quale dei due messaggi sia stato cifrato. Inoltre tale schema è sicuro se, dati due crittogrammi, non è possibile stabilire se essi cifrano lo stesso messaggio. Sia Π = (KeyGen, Encrypt, Decrypt) uno schema di crittografia asimmetrica. Consideriamo l'esperimento seguente.

Esperimento. $\mathrm{Pub}^{\mathcal{A}}_{\Pi}(\mathrm{n})$

1. Viene creata la coppia di chiavi $(sk,\ vk) \leftarrow \mathsf{KeyGen}(1^n)$.

2. L'avversario $\mathcal{A}$ ottiene la chiave pubblica vk.

3. Viene generato un bit $b \leftarrow \{0,1\}$ in maniera casuale

4. L'avversario $\mathcal{A}$ sceglie una coppia di messaggi $m_0,\ m_1$ della stessa lunghezza (per evitare che $\mathcal{A}$ riesca a riconoscere il messaggio cifrato proprio in base alla lunghezza).

5. Viene eseguito $c \leftarrow \mathsf{Encrypt}(vk,\ m_b)$ e c viene reso noto ad $\mathcal{A}$.

6. L'avversario $\mathcal{A}$ vince se riesce ad indovinare a quale messaggio corrisponde il crittogramma ricevuto. In tal caso, l'esperimento varrà 1.

Osservando la definizione proposta da Katz e Lindell in [KL07], possiamo asserire che lo schema crittografico a chiave pubblica Π appena presentato sia CPA-sicuro, ossia resistente ad attacchi di tipo *chosen-plaintext*. Infatti,

nel CPA *indistinguishability experiment*, se l'avversario non riesce a capire quale dei due messaggi sia stato cifrato con probabilità trascurabile maggiore di $\frac{1}{2}$, lo schema è detto CPA-sicuro.

Definizione 3.7. Uno schema di codifica a chiave pubblica $\Pi = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ è sicuro se e solo se, per ogni avversario $\mathcal{A}$ PPT, esiste una funzione trascurabile ε tale che $\Pr(\mathrm{Pub}^{\mathcal{A}}_{\Pi}(\mathrm{n}) = 1) \leq \frac{1}{2} + \varepsilon(\mathrm{n})$.

3.4 Crittografia a Chiave Pubblica Omeomorfa

La crittografia asimmetrica omeomorfa descrive un particolare schema a chiave pubblica [BW14], in cui i messaggi cifrati vengono uniti per formare un unico crittogramma. Viene introdotta nei protocolli di voto elettronico proprio per rendere più complesso risalire al voto espresso da ogni elettore. Lo schema di crittografia asimmetrica omeomorfa è identico a quello di crittografia a chiave pubblica, con l'aggiunta di un algoritmo.

Definizione 3.8 Uno schema HE di crittografia asimmetrica omeomorfa è una quadrupla di algoritmi PPT:

$$\mathrm{HE} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Add}, \mathsf{Decrypt})$$

1. $\mathsf{KeyGen}()$ è l'algoritmo di generazione delle chiavi pubblica e privata. Esso produce una coppia di chiavi $(sk,vk) \leftarrow \mathsf{KeyGen}(1^n)$, rispettivamente chiave privata e chiave pubblica.

2. $\mathsf{Encrypt}(vk,\ m)$ è l'algoritmo di cifratura. Prende in input la chiave pubblica vk ed un messaggio m e produce un crittogramma $c \leftarrow \mathsf{Encrypt}(vk,\ m)$.

3. $\mathsf{Decrypt}(sk,\ c)$ è l'algoritmo di decifratura. Prende in input la chiave privata sk e un crittogramma c e produce in output il messaggio decifrato $d \leftarrow \mathsf{Decrypt}(sk,\ c)$, oppure $\perp$ se il crittogramma è invalido. Questo algoritmo deve essere deterministico.

4. $\mathsf{Add}(vk,\ c_1,\ c_2)$ è l'algoritmo di somma. Prende in input la chiave pubblica vk e due testi cifrati $c_1,\ c_2$. Produce in output un altro crittogramma $s \leftarrow \mathsf{Add}(vk,\ c_1,\ c_2)$.

HE richiede che siano rispettati i seguenti vincoli di correttezza, che devono valere se $(sk,\ vk)$ sono ottenute tramite l'algoritmo KeyGen:

- $\mathsf{Decrypt}(sk,\ \mathsf{Encrypt}(vk,\ m)) = m$
- $\mathsf{Add}(vk,\ \mathsf{Encrypt}(vk,\ m_1),\ \mathsf{Encrypt}(vk,\ m_2)) = \mathsf{Encrypt}(vk,\ m_1{+}m_2)$

Sicurezza di Schemi a Chiave Pubblica Omeomorfa. Uno schema di crittografia asimmetrica omeomorfa è sicuro se, dati due messaggi ed un crittogramma, è impossibile indovinare quale dei due messaggi sia stato cifrato, o se sia stato cifrato uno solo dei due messaggi (e in tal caso dovrà essere impossibile indovinare quale dei due). Sia HE = (KeyGen, Encrypt, Add, Decrypt) uno schema di crittografia asimmetrica omeomorfa. Consideriamo l'esperimento seguente.

Esperimento. $\mathrm{Hom}^{\mathcal{A}}_{HE}(\mathrm{n})$

1. Viene creata la coppia di chiavi $(sk,\ vk) \leftarrow \mathsf{KeyGen}(1^n)$.
2. L'avversario $\mathcal{A}$ ottiene la chiave pubblica vk.
3. L'avversario $\mathcal{A}$ sceglie una coppia di messaggi $m_0,\ m_1$ della stessa lunghezza (per evitare che $\mathcal{A}$ riesca a riconoscere il messaggio criptato proprio in base alla lunghezza).
4. Viene eseguito $c_0 \leftarrow \mathsf{Encrypt}(vk,\ m_0)$ e $c_1 \leftarrow \mathsf{Encrypt}(vk,\ m_1)$.
5. Viene eseguito $c \leftarrow \mathsf{Add}(vk,\ c_0,\ c_1)$ e c viene reso noto ad $\mathcal{A}$.
6. L'avversario $\mathcal{A}$ vince se riesce ad indovinare a quale (o quali) messaggio corrisponde il crittogramma ricevuto. In tal caso, l'esperimento varrà 1.

Anche in questo caso possiamo asserire (osservando quanto scritto in [BW14] e [KL07]) che lo schema crittografico a chiave pubblica omeomorfa appena presentato sia CPA-sicuro, in quanto l'avversario non riesce a capire quale (o quali) dei due messaggi corrisponda al crittogramma ottenuto con probabilità maggiore di $\frac{1}{2}$.

Definizione 3.9. Uno schema di codifica a chiave pubblica omeomorfa HE = (KeyGen, Encrypt, Add, Decrypt) è sicuro se e solo se, per ogni avversario $\mathcal{A}$ PPT, esiste una funzione trascurabile ε tale che $\Pr(\mathrm{Hom}^{\mathcal{A}}_{HE}(\mathrm{n}) = 1) \leq \frac{1}{2} + \varepsilon(\mathrm{n})$.

Per rafforzare la sicurezza di HE, è possibile fortificare ulteriormente l'algoritmo Add usato, in modo che risulti ancora più complicato risalire a quale (o quali) dei due messaggi sia stato cifrato. Introduciamo quindi uno schema che lo consente, chiamato *ElGamal*.

3.5 ElGamal

Lo schema crittografico *ElGamal* si basa sull'algoritmo di *Diffie-Hellman* (e presenta anche il suo stesso grado di sicurezza) [ElG85]. È uno schema

di crittografia omeomorfa a chiave pubblica: di seguito ne presentiamo gli algoritmi.

Definizione 3.10. *ElGamal*, che cifra il messaggio m come una coppia (c, d), è una tripla di algoritmi:

$$\text{EG} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$$

1. $\mathsf{KeyGen}()$ è l'algoritmo di generazione della coppia di chiavi (sk, vk). Prende i due parametri (p, g) e sk in maniera random da $\mathbb{Z}^*_p$. Calcola $vk = g^{sk} \pmod p$ e ritorna la coppia (sk, vk).
2. $\mathsf{Encrypt}(m, vk)$ è l'algoritmo di cifratura. Prende un parametro r in maniera random da $\mathbb{Z}^*_p$. Calcola $c = g^r \pmod p$, $d = m \cdot vk^r \pmod p$ e ritorna (c, d).
3. $\mathsf{Decrypt}(sk, (c, d))$ è l'algoritmo di decifratura. Calcola $m = \dfrac{d}{c^{sk}} \pmod p$.

ElGamal è uno schema crittografico omeomorfo. Estrarre dal messaggio m la coppia (c, d) e dire quali dei due (o anche entrambi) siano stati cifrati è equivalente a risolvere il problema decisionale di *Diffie-Hellman* (che abbiamo detto essere molto complesso). Possiamo quindi sfruttare l'algoritmo Add presente negli schemi asimmetrici omeomorfi in questo modo:

$$\mathsf{Add}((c, d), (c', d')) = ((c \cdot c' \pmod p), (d \cdot d' \pmod p))$$

Avremo quindi che per la coppia di messaggi m_1, m_2 otteniamo un crittogramma su $m_1 \cdot m_2 \pmod p$ (e non più su $m_1 + m_2$). Questo consente il rafforzamento del concetto della proprietà di sicurezza degli schemi omeomorfi introdotti nel paragrafo ad essi dedicato.

Teorema 3.1. Lo schema di codifica a chiave pubblica EG è sicuro se vale DDH.

La dimostrazione di questo teorema si può trovare all'interno di [KL07].

3.6 Schemi di Codifica Threshold

Gli schemi di codifica *threshold* sono particolari schemi crittografici che basano la loro sicurezza sulla divisione delle chiavi private in n-1 chiavi condivise, allocate in server situati in zone diverse. Questi schemi sono ideati in modo tale che, per far si che lo schema non venga forzato, almeno t di n server devono contribuire nelle operazioni in chiave privata. Infatti, un avversario che cerca di forzare uno schema *threshold* con t server non dovrebbe essere in grado di decifrare alcun crittogramma. Tuttavia, ideare schemi crittografici *threshold* a chiave pubblica non è stato un compito così banale [LY11].

Definizione 3.11 Uno schema T di crittografia asimmetrica (t,n)-*threshold* è una tupla di algoritmi PPT:

$\mathrm{T} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{CiphertextVerify}, \mathsf{ShareDecrypt}, \mathsf{ShareVerify}, \mathsf{Combine})$

1. $\mathsf{KeyGen}(\lambda,\ t,\ n)$ è l'algoritmo di generazione delle chiavi. Prende in input un parametro di sicurezza λ e due interi t, n polinomiali in λ, con $1 \leq t \leq n$. n indica il numero dei server, mentre t indica i server che prendono parte al processo di decifratura. L'algoritmo ritorna una tripla $(pk,\ vk,\ sk) \leftarrow \mathsf{KeyGen}(\lambda,\ t,\ n)$, dove: pk è la chiave pubblica; $sk = (sk_1,...,sk_n)$ è un vettore di n chiavi private condivise; $vk = (vk_1,...,vk_n)$ è il corrispondente vettore di chiavi di verifica, che servirà per verificare la validità di una decifratura condivisa generata con la chiave sk corrispondente. Al server i è assegnata la coppia $(i,\ sk_i)$, che consentirà di derivare da qualsiasi crittogramma il testo decifrato condiviso corrispondente.

2. $\mathsf{Encrypt}(pk,\ m)$ è l'algoritmo di cifratura. Presi in input un messaggio m e la chiave pubblica pk, ritorna un crittogramma $c \leftarrow \mathsf{Encrypt}(pk,\ m)$.

3. $\mathsf{CiphertextVerify}(pk,\ c)$ è l'algoritmo di verifica dei crittogrammi. Ritorna 1 se c è ritenuto valido in base alla chiave pubblica pk, altrimenti ritorna 0.

4. $\mathsf{ShareDecrypt}(pk,\ i,\ sk_i,\ c)$ è l'algoritmo di decifratura condivisa. Ritorna una coppia con simbolo speciale $(i,\ \perp) \leftarrow \mathsf{ShareDecrypt}$ se $\mathsf{CiphertextVerify}(pk,\ c) = 0$. Altrimenti ritorna una decifratura condivisa $(i,\ \mu_i) \leftarrow \mathsf{ShareDecrypt}$.

5. $\mathsf{ShareVerify}(pk,\ vk_i,\ c,\ \mu_i)$ è l'algoritmo di verifica della decifratura condivisa. Ritorna 1 se $(i,\ \mu_i)$ è una decifratura condivisa valida; 0 altrimenti.

6. $\mathsf{Combine}(pk,\ vk,\ c,\ \{\mu_i\}_{i\in S})$ è l'algoritmo che, dati pk, vk, c e un sottoinsieme $S \subset \{1,...,n\}$ di dimensione $|S| = t$ con decifrature condivise $\{\mu_i\}_{i\in S}$, ritorna un messaggio m, oppure $\perp$ se l'insieme contiene una o più decifrature condivise invalide.

Notiamo che, in alcuni schemi *threshold*, l'algoritmo $\mathsf{CiphertextVerify}$ è accorpato all'algoritmo $\mathsf{Encrypt}$, quindi sarà quest'ultimo ad effettuare il controllo sulla validità dei crittogrammi prodotti.

Sicurezza di Schemi Threshold. Uno schema di crittografia asimmetrica *threshold* è sicuro se almeno t su n server partecipano al processo di decifratura condivisa. Sia $\mathrm{T} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{CiphertextVerify}, \mathsf{ShareDecrypt}, \mathsf{ShareVerify}, \mathsf{Combine})$ uno schema di crittografia asimmetrica *threshold*. Consideriamo l'esperimento seguente.

Esperimento. $\mathrm{Thr}_T^{\mathcal{A}}(\mathrm{n})$

1. Viene creata la tripla di chiavi $(pk,\ vk,\ sk) \leftarrow \mathsf{KeyGen}(\lambda,\ t,\ n)$. Le chiavi pk e vk vengono rese note all'avversario $\mathcal{A}$, mentre sk viene mantenuta segreta.
2. L'avversario $\mathcal{A}$, in seguito, può:
   - scegliere $i \in \{1,...,n\}$ e ottenere sk_i. Nell'intero esperimento, $\mathcal{A}$ non può ottenere più di t-1 chiavi private condivise.
   - scegliere un indice $i \in \{1,...,n\}$ e un crittogramma c. Di conseguenza, verrà decifrato $\mu_i \leftarrow \mathsf{ShareDecrypt}(pk,\ i,\ sk_i,\ c)$.
3. L'avversario $\mathcal{A}$ sceglie una coppia di messaggi $m_0,\ m_1$ della stessa lunghezza (per evitare che $\mathcal{A}$ riesca a riconoscere il messaggio criptato proprio in base alla lunghezza). Viene poi scelto casualmente un bit $b \leftarrow \{0,1\}$ e computato $c^* \leftarrow \mathsf{Encrypt}(pk,\ m_b)$.
4. L'avversario $\mathcal{A}$ può eseguire nuovamente le mosse del punto 2, ma non gli è concesso eseguire operazioni di decifratura sul crittogramma c^*.
5. L'avversario $\mathcal{A}$ vince se riesce ad indovinare il valore del bit b. In tal caso, l'esperimento varrà 1.

Facendo riferimento a quanto riportato in [LY11], possiamo affermare che lo schema di codifica *threshold* qui presentato sia CCA-sicuro: l'avversario non riesce a capire quale sia il messaggio m cifrato all'inizio dell'esperimento con probabilità maggiore di $\frac{1}{2}$, pur avendo a disposizione alcune coppie crittogramma/testo cifrato (tranne ovviamente quella corrispondente al messaggio m).

Definizione 3.12. Uno schema di codifica a chiave pubblica *threshold* $\mathrm{T} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{CiphertextVerify}, \mathsf{ShareDecrypt}, \mathsf{ShareVerify}, \mathsf{Combine})$ è sicuro se e solo se, per ogni avversario $\mathcal{A}$ PPT, esiste una funzione trascurabile ε tale che $\Pr(\mathrm{Thr}_T^{\mathcal{A}}(\mathrm{n}) = 1) \leq \frac{1}{2} + \varepsilon(\mathrm{n})$.

Vediamo, nella prossima sezione, lo specifico schema di codifica *threshold* che verrà usato in una delle versioni del protocollo elettronico *Minivoting*: *Threshold ElGamal*.

3.7 Threshold ElGamal

Lo schema crittografico *Threshold ElGamal* è uno schema di codifica *threshold* che rafforza *ElGamal*, descritto nella sezione 3.5. Coinvolge n entità, tutte partecipanti al processo di decifratura condivisa dei messaggi. Come detto, esistono anche schemi *threshold* in cui soltanto $t < n$ entità contribuiscono a tale decifratura condivisa.

Definizione 3.13. Un cifrario *Threshold ElGamal* con $t = n$ entità è una tupla di algoritmi:

$$\text{TE} = (\mathsf{KeyGen}, \mathsf{CombineKey}, \mathsf{Encrypt}, \mathsf{DecryptShare}, \mathsf{Combine})$$

1. $\mathsf{KeyGen}()$ è l'algoritmo di generazione della coppia di chiavi $(sk,\ vk)$. Ogni entità prende in input gli stessi tre parametri $(p,\ q,\ g)$ e ritorna la coppia di chiavi $(sk,\ vk)$ generata con l'algoritmo KeyGen di *ElGamal*.

2. $\mathsf{CombineKey}(vk_1 \ldots vk_n)$ è l'algoritmo di combinazione delle chiavi. Prende in input le chiavi pubbliche vk_i di ogni entità e le moltiplica: $vk \leftarrow \prod_{i=1}^{n} vk_i \pmod p$.

3. $\mathsf{Encrypt}(m,\ vk)$ è l'algoritmo di cifratura standard di *ElGamal*, basato sulla chiave pubblica vk ottenuta in precedenza, che ritorna il crittogramma b (corrispondente alla coppia $(c,\ d)$ in *ElGamal*).

4. $\mathsf{DecryptShare}(sk_i,\ b)$ è l'algoritmo di decrittazione condivisa, dove $b = (c,\ d)$ come in *ElGamal*. Prende in input una chiave privata sk_i e il crittogramma b e ritorna una decrittazione condivisa $d_i = b^{sk_i} \pmod p$.

5. $\mathsf{Combine}(b,\ \{d_i\}_{i=1}^{n})$ è l'algoritmo di ricombinazione. Prende in input un crittogramma b e un insieme $D = \{d_i\}_{i=1}^{n}$ di decrittazioni condivise. Ritorna il messaggio decifrato $d = \dfrac{b}{\prod_{i=1}^{n} d_i} \pmod p$.

Sicurezza di Threshold ElGamal. *Threshold ElGamal*, non è uno schema totalmente sicuro se alcune entità del sistema non si comportano correttamente: i parametri in input potrebbero essere alterati o il risultato in output falsificato. Tuttavia, secondo quanto dimostrato circa gli schemi di codifica *threshold* generici, ed anche in base a quanto riportato in [FP01], possiamo asserire che lo schema *Threshold ElGamal* sia CCA-sicuro (ossia che un qualsiasi avversario PPT non riesce a ricavare l'entità del messaggio iniziale con probabilità maggiore di $\frac{1}{2}$, pur avendo a disposizione coppie crittogramma/testo in chiaro).

Definite le primitive necessarie, introduciamo ora lo schema di voto elettronico su cui si basa questo lavoro di tesi, *Minivoting*.

Capitolo 4

Il Protocollo Minivoting

Lo schema crittografico *Minivoting* è uno schema di base che serve nella realizzazione del protocollo di voto elettronico *Helios*, protocollo in fase di sperimentazione in alcuni Stati Europei [BC+11]. *Minivoting* si presenta in diverse versioni: ognuna di esse basa il suo schema su uno degli schemi crittografici di cui abbiamo ampiamente discusso nel capitolo 3, ossia *ElGamal, Threshold ElGamal* e protocollo *DCP* (che fa parte degli *zero-knowledge proofs*, [GO13]). Presentiamo, di seguito, queste versioni di *Minivoting* inserendo, ogni volta, una nuova primitiva che rafforza la sicurezza dello schema stesso.

4.1 Minivoting Semplice

La prima versione dello schema, denominata MS, è appunto quella più semplice (e anche la meno sicura), che getta le prime basi per costruire in seguito una versione maggiormente sicura. Notiamo che questo protocollo può essere parametrizzato su un qualsiasi schema di crittografia asimmetrica omeomorfa.

Definizione 4.1. MS, è uno schema *Minivoting* a singola autorità, con uno spazio di messaggi pari a $\mathbb{Z}^*_p$ (n maggiore del numero degli elettori). Necessita di un *bulletin board* dove gli elettori possano pubblicare i loro messaggi (autenticati) e di un numero di elettori minore di n. MS(HE), basato su un generico schema a chiave pubblica omeomorfa HE, è così strutturato:

1. L'autorità crea una coppia di chiavi $(sk,\ vk) \leftarrow \mathrm{HE.KeyGen}(1^n)$ e pubblica vk sul *bulletin board*.

2. Gli elettori leggono la chiave pubblica vk dal *bulletin board*. Scelgono $v = 1$ per votare *si* e $v = 0$ per votare *no* e creano il voto $b \leftarrow \mathrm{HE.Encrypt}(v,\ vk)$ che pubblicheranno sul *bulletin board*.

3. L'autorità usa l'algoritmo HE.Add per sommare tutti i voti, creando un voto finale s che verrà decifrato come $d \leftarrow \text{HE.Decrypt}(s, sk)$. Infine, l'autorità conterà il numero m di voti inviati e pubblicherà il risultato sul *bulletin board* (d *si,* m-d *no*).

Sicurezza di Minivoting Semplice. Lo schema *Minivoting semplice* basa la sua sicurezza sulla difficoltà dell'avversario di risalire al messaggio inviato. Osserveremo in seguito che *Minivoting semplice* parametrizzato sullo schema di codifica *ElGamal* basa la sua sicurezza sulla complessità del problema di *Diffie-Hellman*.

Nell'esperimento di riferimento che verrà presentato qui di seguito, l'avversario può scegliere due voti per ogni elettore, il quale voterà con uno dei due (tutti gli elettori faranno la stessa scelta). Lo scopo dell'avversario è quello di determinare quale scelta sia stata fatta dagli elettori. Consideriamo quindi l'esperimento seguente.

Esperimento. $\text{Sec}^{\mathcal{A}}_{MS,HE}(\text{n})$

1. Viene scelto casualmente un bit $b \leftarrow \{0, 1\}$.

2. Una volta per ogni elettore, l'avversario $\mathcal{A}$ può scegliere due voti, v_0 e v_1.

3. Se $b = 0$, l'elettore voterà con v_0. Al contrario, se $b = 1$ l'elettore voterà con v_1. L'avversario $\mathcal{A}$ può consultare il *bulletin board* in qualsiasi momento. Quando tutti gli elettori hanno votato, $\mathcal{A}$ ottiene la somma dei voti v_0.

4. L'avversario $\mathcal{A}$ vince se riesce ad indovinare il valore del bit b.

Al termine delle elezioni l'avversario ottiene la somma dei voti v_0 invece del risultato delle elezioni; questo perché l'avversario è in grado di calcolare tale risultato, avendo scelto lui il valore dei voti v_0, v_1 con i quali far votare ogni elettore. Ovviamente l'avversario non riceve nemmeno la somma dei voti v_b, poiché sarebbe banale indovinare il valore del bit b e vincere l'esperimento.

Definizione 4.2. Uno schema *Minivoting Semplice* è CPA-sicuro se e solo se, per ogni avversario $\mathcal{A}$ PPT, esiste una funzione trascurabile ε tale che $\Pr(\text{Sec}^{\mathcal{A}}_{MS,HE}(\text{n}) = 1) \leq \frac{1}{2} + \varepsilon(\text{n})$.

Teorema 4.1. Lo schema MS(EG) è sicuro se vale DDH.

Dimostrazione. Sia MS uno schema *Minivoting Semplice* parametrizzato su uno schema di crittografia asimmetrica omeomorfa HE generico.
Consideriamo un avversario $\mathcal{A}$ PPT per tale schema MS, che possa compiere al più n mosse, con $n \in \mathbb{N}$.
Consideriamo, inoltre, una sequenza $G_0,...,G_{n+1}$ di giochi, tali che:

- in ognuno di essi venga scelto casualmente il bit $b \leftarrow \{0,1\}$.
- $\forall i \leq n+1$, nel gioco G_i si ha che:
  - nelle prime $(i-1)$ mosse, ogni elettore voterà con il voto v_0;
  - nella i-esima mossa, l'elettore voterà con il voto v_b;
  - nelle restanti $(i+1)$ mosse, ogni elettore voterà con il voto v_1.

Definiamo $\alpha(n) = \Pr[Sec^{\mathcal{A}}_{MS,HE}(n) = 1]$ come la probabilità che l'avversario $\mathcal{A}$ vinca l'esperimento Sec.
Secondo la definizione di *dimostrazioni game-based*, per forzare uno schema di codifica l'avversario dovrebbe distinguere il gioco G_0 dal gioco G_{n+1} (dove G_0 corrisponde al gioco iniziale e G_{n+1} a quello finale). Ossia, dato un algoritmo $\widetilde{\mathcal{D}}$ PPT (detto *distinguitore*) vale:

$$\Pr[\widetilde{\mathcal{D}}(G_0,G_{n+1}) = 1] \leq \alpha(n).$$

Per la stessa definizione, la probabilità α che un distinguitore possa distinguere il primo dall'ultimo gioco della dimostrazione, è pari alla somma delle probabilità che lo stesso distinguitore distingua ogni gioco della dimostrazione dal suo successivo. Possiamo quindi affermare che:

$$\alpha(n) = \sum_{i=0}^{n+1} \widetilde{\alpha}_i(n)$$

dove, $\forall i \leq n+1$:

$$\widetilde{\alpha}_i(n) = \Pr[\widetilde{\mathcal{D}}(G_i,G_{i+1}) = 1]$$

è la probabilità che il distinguitore $\widetilde{\mathcal{D}}$ distingua un gioco dal suo successivo. Otteniamo quindi che $\forall i \leq n+1$ vale:

$$\widetilde{\alpha}_i(n) = \Pr[\widetilde{\mathcal{D}}(G_i,G_{i+1}) = 1] \geq \frac{\alpha}{n+1}.$$

Siccome i giochi $G_0,...,G_{n+1}$ si differenziano per una singola mossa (la i-esima, nella quale l'elettore vota con v_b), possiamo parametrizzare lo schema MS così descritto su uno schema di codifica asimmetrico omeomorfo specifico, *ElGamal*, e dimostrare la sicurezza CPA(MS(EG)) di uno di questi giochi.
Sia quindi EG lo schema di codifica *ElGamal* e sia:

$$\varepsilon(n) \stackrel{def}{=} \Pr[PubK^{\mathcal{A}}_{EG}(n) = 1].$$

Definiamo poi un nuovo schema $\widetilde{EG}$ come una coppia di algoritmi PPT:

$$\widetilde{EG} = (\mathsf{KeyGen}, \mathsf{Encrypt})$$

1. KeyGen è uguale a quello presente in EG.
2. Encrypt(m, pk) = c, dove $c = \langle g^y, g^z \rangle$ e y,z sono scelti casualmente da $\mathbb{Z}_q$.
3. A differenza di EG, $\widetilde{EG}$ non ha gli algoritmi Add e Decrypt: tuttavia, l'esperimento $\text{PubK}^{\mathcal{A}}_{\widetilde{EG}}(\text{n})$ risulta corretto poiché dipende soltanto dagli algoritmi KeyGen ed Encrypt.

Siccome sia g^y che g^z sono indipendenti da m, si ha che:

$$\Pr[\text{PubK}^{\mathcal{A}}_{\widetilde{EG}}(\text{n}) = 1] = \tfrac{1}{2}.$$

Consideriamo l'algoritmo distinguitore $\mathcal{D}$ PPT, che tenta di risolvere il problema decisionale di *Diffie-Hellman* DDH, così funzionante:

1. Setta $pk = \langle G, q, g, g_1 \rangle$ e ottiene i due voti v_0, v_1 invocando $\mathcal{A}(pk)$ (dove $\mathcal{A}$ è l'avversario per MS(EG)).
2. Sceglie in maniera casuale $b \leftarrow \{0,1\}$.
3. Per ogni elettore setta $c_1 = g_2$ e:
   - $c_2 = g_3 \cdot v_0$ per i primi (i-1) elettori;
   - $c_2 = g_3 \cdot v_b$ per l'i-esimo elettore;
   - $c_2 = g_3 \cdot v_1$ per i restanti i+1 elettori.
4. Invoca $\mathcal{A}(\langle c_1, c_2 \rangle)$ e da esso ottiene b'.
5. Se $b = b$' allora il distinguitore $\mathcal{D}$ restituirà in output 1. Altrimenti 0.

Vediamo il comportamento del distinguitore $\mathcal{D}$ nei due schemi visti, ossia $\widetilde{EG}$ ed EG.

$\mathcal{D}$ in $\widetilde{EG}$.
Scegliamo in maniera casuale le variabili x, y, $z \in \mathbb{Z}_q$ e settiamo $g_1 = g^x$, $g_2 = g^y$ e $g_3 = g^z$.
$\mathcal{A}$ verrà invocato con input uguali a $pk = \langle G, q, g, g^x \rangle$ e $\langle c_1, c_2 \rangle = \langle g^y, g^z \cdot m_s \rangle$, dove $s \in \{b,0,1\}$.
Avremo che $\Pr[\mathcal{D}(G, q, g, g^x, g^y, g^z) = 1] = \Pr[\text{PubK}^{\mathcal{A}}_{\widetilde{EG}}(\text{n}) = 1] = \frac{1}{2}$.

$\mathcal{D}$ in EG.
Scegliamo in maniera casuale le variabili x, $y \in \mathbb{Z}_q$ e settiamo $g_1 = g^x$, $g_2 = g^y$ e $g_3 = g^{xy}$.

$\mathcal{A}$ verrà invocato con input uguali a $pk = \langle G, q, g, g^x \rangle$ e $\langle c_1, c_2 \rangle = \langle g^y, g^{xy} \cdot m_s \rangle$, dove $s \in \{b,0,1\}$.
Avremo che $\Pr[\mathcal{D}(G, q, g, g^x, g^y, g^{xy}) = 1] = \Pr[\text{PubK}^{\mathcal{A}}_{EG}(\text{n}) = 1] = \varepsilon(\text{n})$.

Siccome il problema DDH è un problema di elevata complessità relativo a G, esiste una funzione trascurabile $\widetilde{\varepsilon}$ tale che:

$$\widetilde{\varepsilon}(\text{n}) = |\Pr[\mathcal{D}(G, q, g, g^x, g^y, g^z) = 1] \text{ - } \Pr[\mathcal{D}(G, q, g, g^x, g^y, g^{xy}) = 1]| = \\ |\tfrac{1}{2} - \varepsilon(\text{n})|.$$

Ciò implica, a conclusione della dimostrazione, che $\varepsilon(\text{n}) \leq \frac{1}{2} + \widetilde{\varepsilon}(\text{n})$ e che quindi MS(EG) è sicuro.

4.1.1 Problematiche di Minivoting Semplice

Il problema principale di *Minivoting Semplice* è che esso si basa su un'autorità soltanto. Di conseguenza:

1. Un'autorità disonesta potrebbe decifrare singoli voti, violando così la privacy degli elettori.

2. Se l'autorità perdesse la propria chiave privata, lo spoglio dei voti non potrebbe essere eseguito.

Alla luce di queste problematiche, cerchiamo di rafforzare la sicurezza di *Minivoting Semplice* presentandone una nuova versione, che necessita di n autorità.

4.2 Minivoting Threshold

La seconda versione dello schema *Minivoting*, MT, è più solida della precedente perché basa i suoi algoritmi di cifratura e decifratura su uno schema di codifica *threshold* T, che è più sicuro rispetto allo schema asimmetrico omeomorfo HE su cui si basava MS.

Definizione 4.3. MT, è uno schema *Minivoting* a n autorità, con uno spazio di messaggi pari a $\mathbb{Z}^*_p$ (n maggiore del numero degli elettori). Necessita di un *bulletin board* dove gli elettori possano pubblicare i loro messaggi (autenticati) e ad un numero di elettori minore di n. MT(T), basato su un generico schema di codifica *threshold* T, è così strutturato:

1. Ogni autorità prende in input gli stessi tre parametri (p, q, g) e crea una coppia di chiavi $(sk_i, vk_i) \leftarrow \text{T.}\mathsf{KeyGen}(1^n)$. Un'autorità moltiplica tutte le chiavi pubbliche (vk_1,...,vk_n) generate, creando una chiave pubblica condivisa vk. La chiave vk, infine, verrà pubblicata sul *bulletin board*.

2. Gli elettori leggono la chiave pubblica *vk* dal *bulletin board*. Scelgono $v = 1$ per votare *si* e $v = 0$ per votare *no* e creano il voto $b \leftarrow \text{T.}\mathsf{Encrypt}(v,\ vk)$ che pubblicheranno sul *bulletin board*. In questo caso l'algoritmo Encrypt effettua anche il controllo di validità del crittogramma generato.

3. Ogni autorità decifra uno dei crittogrammi b prodotto dagli elettori, generando una decifratura condivisa $d_i \leftarrow \text{T.}\mathsf{ShareDecrypt}(sk_i,\ b)$.

4. Infine, controllata delle altre, un'autorità ricombina tutte le decifrature condivise prodotte, ottenendo $d \leftarrow \text{T.}\mathsf{Combine}(b,\ \{d_i\}_{i=1}^{n})$ che verrà pubblicato sul *bulletin board* come risultato finale delle elezioni.

Notiamo che, in *Minivoting Threshold*, è necessario che ogni autorità sia presente in entrambi gli step di decifratura.

Sicurezza di Minivoting Threshold. Anche lo schema *Minivoting Threshold* basa la sua sicurezza sulla difficoltà dell'avversario di risalire al messaggio inviato. In particolare, *Minivoting Threshold* parametrizzato sullo schema *Threshold ElGamal* (cioè MT(TE)), basa la sua sicurezza sulla complessità di *Diffie-Hellman*. Inoltre, l'introduzione di n autorità invece di una soltanto garantisce maggiore sicurezza: le autorità si controllano a vicenda, quindi nessuna potrebbe decifrare un singolo voto (violando la privacy dell'elettore) e nemmeno alterare il risultato finale.

4.2.1 Problematiche di Minivoting Threshold

Nonostante il rafforzamento dello schema mediante lo schema di codifica *threshold*, *Minivoting Threshold* non può essere sicuro se almeno una delle entità coinvolte non si comporta onestamente. Ad esempio:

- Un elettore potrebbe cifrare g^2 (quindi nella somma finale dei voti, il suo varrebbe doppio). Negli schemi più complessi rispetto al *yes/no question*, gli elettori hanno quindi più modi per imbrogliare.

- Un elettore potrebbe bloccare le elezioni inviando un crittogramma contenente g^r, con r random; nessuno sarebbe in grado di decifrare tale crittogramma né la somma dei voti, conseguentemente.

- In pratica, gli elettori potrebbero cifrare un voto non valido e nessuna autorità avrebbe la possibilità di controllarlo.

Per fortificare ancora di più lo schema e assicurarci che gli elettori cifrino il voto correttamente, introduciamo una terza versione di *Minivoting*, che si basa su un protocollo *zero-knowledge proof*.

4.3 Minivoting Zero-Knowledge

La terza ed ultima versione dello schema *Minivoting*, MP, è la più sicura delle tre. Come le precedenti, basa i suoi algoritmi sugli schemi di codifica asimmetrica omeomorfa e *threshold* e, in aggiunta, consente alle autorità di verificare che gli elettori abbiano cifrato un voto valido. Questa verifica di validità è garantita dall'applicazione del protocollo *DCP* (*Disjunctive Chaum-Pedersen*), che fa parte degli *zero-knowledge proofs*.

Definizione 4.4. MP, è uno schema *Minivoting* a n autorità, con uno spazio di messaggi pari a $\mathbb{Z}^*_p$ (n maggiore del numero degli elettori). Necessita di un *bulletin board* dove gli elettori possano pubblicare i loro messaggi (autenticati) e ad un numero di elettori minore di n. MP(HE,T), basato su un generico schema a chiave pubblica omeomorfa HE e su un generico schema di codifica *threshold* T, è così strutturato:

1. Ogni autorità prende in input gli stessi tre parametri (p, q, g) e crea una coppia di chiavi (sk, vk) $\leftarrow$ HE.KeyGen(). Un'autorità moltiplica tutte le chiavi pubbliche vk_i appena generate, creando una chiave pubblica condivisa vk. La chiave vk, infine, verrà pubblicata sul *bulletin board*.

2. Gli elettori leggono la chiave pubblica vk dal *bulletin board*. Scelgono $v = 1$ per votare *si* e $v = 0$ per votare *no* e creano il voto $b \leftarrow$ HE.Encrypt(v, vk) che pubblicheranno sul *bulletin board* insieme alla dimostrazione di correttezza del voto stesso, effettuata tramite il protocollo *DCP*.

3. Ogni autorità prende uno dei crittogrammi b prodotto dagli elettori e ne controlla la dimostrazione di validità. Se essa è corretta, l'autorità decifra b generando una decifratura condivisa $d_i \leftarrow$ T.ShareDecrypt(sk_i, b). Nel caso in cui il voto non fosse valido (dimostrazione di validità non corretta), questo viene scartato.

4. Infine, controllata delle altre, un'autorità ricombina tutte le decifrature condivise prodotte, ottenendo $d \leftarrow$ T.Combine(b, $\{d_i\}_{i=1}^n$) che verrà pubblicato sul *bulletin board* come risultato finale delle elezioni.

Notiamo che gli algoritmi KeyGen, Encrypt, Decrypt sono quelli usati nello schema HE, mentre gli algoritmi ShareDecrypt, Combine derivano dallo schema T. Le prove di dimostrazione di correttezza del voto cifrato dagli elettori, invece, derivano dall'applicazione del protocollo *DCP*. Infine, sottolineiamo che ogni autorità deve essere presente in entrambi gli step di decifratura dei crittogrammi.

Sicurezza di Minivoting Zero-Knowledge. *Minivoting Zero-Knowledge* basa la sua sicurezza sul fatto che le autorità si controllano a vicenda, quindi nessuna di esse può violare la privacy degli elettori o alterare il risultato delle elezioni. L'aggiunta della dimostrazione di correttezza mediante il protocollo *DCP* garantisce una maggiore sicurezza, in quanto gli elettori, dovendo dimostrare la validità del voto cifrato, non hanno possibilità di alterare le elezioni. Come gli schemi *Minivoting* descritti in precedenza, *Minivoting Zero-Knowledge* parametrizzato sugli schemi di codifica *ElGamal* e *Threshold ElGamal* (cioè MP(EG,TE)) basa la sua sicurezza sulla complessità del problema decisionale di *Diffie-Hellman*. Inoltre, *Minivoting Zero-Knowledge* è sicuro contro elettori disonesti: nessun elettore sarà in grado di creare schede false e nemmeno di derivare schede elettorali da quella di un elettore onesto. Consideriamo gli esperimenti seguenti.

Esperimento. $\text{Zef}^{\mathcal{A}}_{MP,HE,T}(\text{n})$

1. Viene scelto casualmente un bit $b \leftarrow \{0, 1\}$.
2. Per ogni elettore i, l'avversario $\mathcal{A}$ può scegliere se:
   - dichiarare l'elettore i onesto e scegliere due voti v_0 e v_1. Se $b = 0$ l'elettore voterà con v_0, altrimenti con v_1;
   - dichiarare l'elettore i disonesto e generare un voto falso v_b, con b arbitrario, che verrà processato nell'elezione.

   L'avversario $\mathcal{A}$ può consultare il *bulletin board* in qualsiasi momento.
3. Quando ogni elettore ha votato, l'avversario $\mathcal{A}$ riceve la somma dei voti in questa maniera:
   - per ogni elettore onesto riceve il voto v_0;
   - per ogni elettore disonesto riceve il voto falso inviato.
4. L'avversario $\mathcal{A}$ vince se riesce ad indovinare il valore del bit b.

Definizione 4.5. Uno schema *Minivoting Zero-Knowledge* è CCA-sicuro contro la creazione di schede elettorali false se e solo se, per ogni avversario $\mathcal{A}$ PPT, esiste una funzione trascurabile ε tale che $\Pr(\text{Zef}^{\mathcal{A}}_{MP,HE,T}(\text{n}) = 1) \leq \frac{1}{2} + \varepsilon(\text{n})$.

Esperimento. $\text{Zed}^{\mathcal{A}}_{MP,HE,T}(\text{n})$

1. Viene scelto casualmente un bit $b \leftarrow \{0, 1\}$.

2. Una sola volta, l'avversario $\mathcal{A}$ può scegliere una coppia di voti v_0, v_1 della stessa lunghezza e ottenere $c \leftarrow \text{HE.}\mathsf{Encrypt}(v_b,\ vk)$ corrispondente a uno dei due.

3. Una sola volta, l'avversario $\mathcal{A}$ può far decifrare una lista di crittogrammi $(c_0, ..., c_n)$; se $\mathcal{A}$ ha già ottenuto il crittogramma c, non può includerlo nella lista.

4. L'avversario $\mathcal{A}$ vince se riesce ad indovinare il valore del bit b.

Definizione 4.6. Uno schema *Minivoting Zero-Knowledge* è CCA-sicuro contro la derivazione delle schede elettorali se e solo se, per ogni avversario $\mathcal{A}$ PPT, esiste una funzione trascurabile ε tale che $\Pr(\text{Zed}^{\mathcal{A}}_{MP,HE,T}(\text{n}) = 1) \leq \frac{1}{2} + \varepsilon(\text{n})$.

Procediamo ora con la descrizione dettagliata del primo tra i tre protocolli visti e con la prova della sua sicurezza, tutto mediante il dimostratore automatico `Cryptoverif`.

Capitolo 5

CryptoVerif

`CryptoVerif` è un *tool* automatico usato per dimostrare la sicurezza di protocolli nel modello computazionale [BC13]. Sviluppato da Bruno Blanchet nel linguaggio di programmazione OCaml, `CryptoVerif` è stato rilasciato per la prima volta nel 2006. Si basa sulla tecnica delle dimostrazioni basate su giochi (descritta nella sezione 3.2): il primo gioco costituisce l'input e descrive il protocollo vero e proprio, le primitive crittografiche usate e le proprietà di sicurezza da dimostrare; i restanti, invece, vengono generati automaticamente dal *tool*. Fino ad oggi `CryptoVerif` è già stato usato varie volte, ad esempio come dimostratore della sicurezza di FDH (*Full Domain Hash*, uno schema di firma basato sul paradigma *hash-and-sign*; approfondimenti in [Cor00]).

Questo capitolo descriverà il linguaggio che `CryptoVerif` implementa, corredato da un semplice esempio.

5.1 Sintassi

`CryptoVerif` permette di scegliere tra due tipi diversi di frontend: channels frontend o oracles frontend. La sostanziale differenza tra i due, oltre alla sintassi stessa, è che il primo usa un calcolo simile al Π-*calcolo* [SW01], mentre l'altro usa un calcolo più vicino ai *games* crittografici [Sho06]. Il lavoro svolto in questa tesi basa la sua sintassi sugli oracles frontend [BC16]. Negli oracles frontend, un tipico file di input è composto da una lista di dichiarazioni seguita dalla dichiarazione di un oracolo:

$$\langle declaration \rangle^* \texttt{ process } \langle odef \rangle$$

Vediamoli nel dettaglio.

5.1.1 Dichiarazioni

Le dichiarazioni descrivono nello specifico le primitive ed i parametri usati nel protocollo in esame. Possono essere:

- *Tipo*: `type` ⟨*ident*⟩ [[seq$^+$⟨*option*⟩]].
  Viene dichiarato un nuovo tipo, che corrisponde ad un insieme di stringhe di bits oppure a $\perp$. Facoltativamente, la dichiarazione può essere seguita da una sequenza di opzioni. Tra le più usate troviamo bounded (indica che le stringhe di bits hanno una lunghezza limitata) e fixed (indica che ogni stringa di bits ha una certa lunghezza n). Ad esempio, la coppia di chiavi pubblica e privata sarà dichiarata con tipo fixed; mentre un messaggio sarà dichiarato con il tipo bounded.

  ```
  type crypt[fixed]
  ```

  In questo esempio, dichiariamo un nuovo tipo crypt con opzione fixed.

- *Funzione*: `fun` ⟨*ident*⟩ (seq$^+$⟨*ident*⟩):⟨*ident*⟩.
  Viene dichiarata una nuova funzione, che prende in input una sequenza di tipi e ritorna un tipo come risultato.

  ```
  fun mul(plain, G):G, dove mul:plain,G -> G.
  ```

  In questo esempio dichiariamo una nuova funzione, denominata mul. Essa prende in input due parametri di tipo plain e G, e ritorna un risultato di tipo G.

- *Parametro*: `param` seq$^+$⟨*ident*⟩.
  Viene dichiarato un nuovo parametro, identificato come il parametro di sicurezza.

  ```
  param ns.
  ```

  In questo esempio dichiariamo un nuovo parametro (di sicurezza) ns.

- *Probabilità*: `proba` ⟨*ident*⟩.
  Viene dichiarata una nuova probabilità, in genere usata come input di primitive crittografiche predefinite.

  ```
  proba pDDH.
  ```

  In questo esempio dichiariamo una nuova probabilità pDDH, che potrà essere usata come valore numerico (ad esempio per indicare la probabilità che un avversario forzi un determinato schema crittografico), oppure come input di una primitiva crittografica predefinita (DDH, in questo caso).

- *Primitiva predefinita*: `expand` ⟨*ident*⟩ (seq⟨*ident*⟩).
  Viene espansa una macro mediante gli argomenti in input. Questa dichiarazione si usa principalmente per sfruttare primitive crittografiche già definite.

```
expand DDH(G, Z, g, exp, mult, pDDH).
```

  In questo esempio espandiamo la macro DDH, definendo dei valori che soddisfano l'assunzione di *Diffie-Hellman*: i parametri G, Z, pDDH (dove i primi due sono tipi e il restante è una probabilità) costituiscono l'input, mentre exp, mult e g (le prime sono funzioni, mentre g è generatore di un gruppo di elementi) vengono restituiti in output espandendo la macro.

- *Equivalenza*: `equiv` ⟨*omode*⟩ [| ... |⟨*omode*⟩] <=(⟨*proba*⟩)=> [[manual]] ⟨*ogroup*⟩ [| ... |⟨*ogroup*⟩].
  Viene dichiarata un'equivalenza tra due elementi e la relativa probabilità di distinguerli. Tali elementi possono essere funzioni, oracoli o anche parti del *game* iniziale.

```
equiv L <=(p)=> R.
```

  In questo esempio indichiamo che una macchina di Turing probabilistica distingue L da R con probabilità al massimo pari a p. La dichiarazione di un'equivalenza fra elementi permette di dare a `CryptoVerif` una regola di riscrittura della parte del *game* che la contiene, a meno di una certa probabilità. Si consente al dimostratore di riscrivere espressioni che compaiono nel primo gioco e che quindi saranno diverse nel secondo, sempre considerando la probabilità in questione. Notiamo che se la probabilità p è pari a 0, allora l'equivalenza a cui si riferisce deve sempre valere.

- *Query*: `query` seq$^+$⟨*query*⟩.
  Viene dichiarata una nuova sequenza di query, che può contenere da una a più query. Ognuna di esse indica quale proprietà di sicurezza si vuole dimostrare.

```
query secret b.
query secret1 b.
```

  In questo esempio, la prima query ci consente di dimostrare che la stringa b è indistinguibile da qualsiasi altra stringa generata casualmente, eseguendo diversi test. La seconda query, invece, dimostra la stessa proprietà della precedente, eseguendo però un solo test.

In conclusione, prima di illustrare la sintassi degli oracoli, aggiungiamo due ulteriori note: i commenti vengono scritti racchiusi tra (* parentesi e asterischi *); le primitive crittografiche predefinite (espandibili con la dichiarazione expand), sono fornite in input a `CryptoVerif` in un file separato da quello corrente.

5.1.2 Oracoli

Dopo aver stilato la lista delle dichiarazioni, si inserisce la parola chiave *process* e si procede definendo l'oracolo principale. Notiamo che un oracolo non è altro che la descrizione di un processo (non c'entra nulla con la definizione di *oracolo* in crittografia moderna). Attraverso la grammatica BNF, possiamo descrivere in maniera formale la composizione degli oracoli:

$$
\begin{aligned}
\langle odef \rangle \ ::= \ & \langle ident \rangle \\
| \ & (\langle odef \rangle) \\
| \ & 0 \\
| \ & \langle odef \rangle \mid \langle odef \rangle \\
| \ & \text{foreach } \langle ident \rangle \leq \langle ident \rangle \text{ do } \langle odef \rangle \\
| \ & \langle odef \rangle \ (\text{seq}\langle pattern \rangle) := \langle obody \rangle
\end{aligned}
$$

Il corpo di un oracolo, quindi, può contenere 0 (oracolo nullo), la chiamata in parallelo di più oracoli tramite la parola chiave *foreach* oppure con l'operando | di parallelo, la chiamata ad un singolo oracolo, e semplici operazioni. Anche quest'ultime sono descritte formalmente mediante la grammatica BNF, nel modo seguente:

$$
\begin{aligned}
\langle obody \rangle \ ::= \ & \langle ident \rangle \\
| \ & (\langle obody \rangle) \\
| \ & \langle ident \rangle \leftarrow \text{R } \langle ident \rangle [; \langle obody \rangle] \\
| \ & \langle ident \rangle [:\langle ident \rangle] \leftarrow \langle term \rangle [; \langle obody \rangle] \\
| \ & \text{return } (\text{seq}\langle term \rangle) [; \langle odef \rangle] \\
| \ & \text{if } \langle cond \rangle \text{ then } \langle obody \rangle \text{ [else } \langle obody \rangle] \\
| \ & \text{end}
\end{aligned}
$$

Vediamo ora queste operazioni nel dettaglio, aiutandoci con alcuni esempi.

- *Assegnazione di un valore random ad una variabile.* Ad esempio, con il frammento di codice seguente

  ```
  b <-R bool;
  ```

  assegnamo alla variabile b un valore random di tipo bool. Osserviamo che la casualità è segnalata dalla presenza della lettera R di fianco alla freccia di assegnamento.

- *Assegnazione del risultato di una funzione ad una variabile.* Ad esempio, mediante la seguente operazione

```
c <- exp(g,r);
```

  assegnamo alla variabile c il risultato della funzione esponenziale exp, che calcola g^r. Notiamo che la variabile c avrà lo stesso tipo del valore di ritorno della funzione exp (definito nella lista delle dichiarazioni).

- *Ritorno di uno o più valori.* Ad esempio, con questo frammento

```
return(a);processA
```

  ritorniamo la variabile a all'oracolo definito in un altro processo (chiamato, in questo caso, processA), che verrà lanciato dopo l'esecuzione della return stessa.

- *Verifica di una condizione.* Ad esempio, nel frammento mostrato di seguito

```
if check(bool,a,b) then return(a);
  else return(b);
```

  la condizione all'interno dell'if consente di verificare il valore booleano della variabile bool: se la funzione check vale 0, viene ritornata la variabile a; se vale 1, viene ritornata la variabile b.

- *Terminazione di processo.* Il frammento seguente

```
end.
```

  consente di terminare l'esecuzione in maniera anormale, a seguito di un particolare evento o di una condizione non verificata.

Sottolineiamo nuovamente il fatto che la sintassi e le operazioni eseguibili in `CryptoVerif` sono molte di più rispetto a quelle qui presentate: in questo lavoro viene descritto soltanto il sottoinsieme utilizzato.

5.1.3 Piccolo Esempio Pratico

In questa breve sezione viene presentato, come anticipato dal titolo, un piccolo esempio pratico, per familiarizzare meglio con l'uso e la logica di `CryptoVerif`. Consideriamo il seguente protocollo, che utilizza crittografia asimmetrica e firma digitale (per un ulteriore approfondimento riguardo a quest'ultima primitiva si veda [KL07]). In breve, Alice (il mittente) manda un messaggio firmato a Bob (il destinatario), che verifica l'autenticità della firma ricevuta. Per la versione del protocollo con channels frontend si veda [BC13]. Per quella con oracles frontend si faccia riferimento al codice seguente:

```
process
        OracleStart() :=
        rk <-R keyseed;
        pk <- pkgen(rk);
        sk <- skgen(rk);
        return(pk);
        (foreach i1 <= n1 do processA |
                foreach i2 <= n2 do processB)

let processA =
        OracleAlice() :=
        m <-R nonce;
        s <-R seed;
        return(m, sign(m,sk,s))

let processB =
        OracleBob(m':nonce,s':signature) :=
        if check(m',pk,s') then return() else end
```

All'inizio dell'esecuzione viene lanciato OracleStart, che genera la coppia di chiavi (pk, sk) applicando al seme casuale rk gli algoritmi di generazione delle chiavi pkgen ed skgen. Ritorna poi la chiave pubblica, in modo che anche l'avversario possa conoscerla. Infine, OracleStart chiama parallelamente gli oracoli OracleAlice e OracleBob, rispettivamente n1 e n2 volte. OracleAlice genera un messaggio casuale m e un seme random s; ritorna poi il messaggio m e la firma dello stesso, eseguita con la chiave privata sk ed il seme s. OracleBob prende in input m' e s', elementi ritornati dalla corrispondente chiamata parallela di OracleAlice. Tramite la funzione check controlla se la firma s' ricevuta è valida per il messaggio m', usando la chiave pubblica pk. Se la firma è corretta, l'oracolo ritorna normalmente. Altrimenti il processo viene terminato con end.

5.2 Minivoting Semplice in CryptoVerif

Lo scopo della prima parte del lavoro svolto è di dimostrare la sicurezza di *Minivoting Semplice*, con l'ausilio del dimostratore automatico `CryptoVerif` descritto nel capitolo precedente. Il protocollo *Minivoting Semplice*, invece, è descritto nella sezione 4.1. Da questo momento in poi verrà illustrato e commentato *step-by-step* l'intero protocollo fornito in input al dimostratore, seguito dalla descrizione dei risultati ottenuti.

5.2.1 Descrizione del Protocollo

L'input file realizzato inizia, come di consueto, con una lista di dichiarazioni.

```
1 type plain [fixed].
2 type crypt [fixed].
3 type G [bounded, large].
4 type Z [bounded, large].
```

Nella prima parte inseriamo l'insieme dei tipi. Alla linea 1 definiamo il tipo plain, che indica il tipo dei messaggi in chiaro; al contrario, la linea 2 definisce il tipo crypt, ossia quello dei messaggi cifrati. Entrambi fixed. Le linee 3 e 4 dichiarano i tipi G e Z, entrambi sia bounded che large, utilizzati in un secondo momento come input per la primitiva crittografica relativa al problema decisionale di *Diffie-Hellman* (spiegata in seguito); G rappresenta il tipo di un gruppo di elementi, mentre Z il tipo degli eventuali esponenti.

```
6 param ns.
7 proba pDDH.
```

Con queste due linee di codice dichiariamo i parametri e le probabilità. La linea 6 mostra il parametro ns, che corrisponde al parametro di sicurezza sul quale verrà parametrizzato l'intero protocollo. Verrà utilizzato principalmente per indicare il numero polinomiale di oracoli che si vogliono eseguire. La linea successiva dichiara la probabilità pDDH che, insieme ai tipi G e Z descritti sopra, costituirà l'input della primitiva relativa al problema di *Diffie-Hellman*.

```
9 expand DDH(G, Z, g, exp, mult, pDDH).
```

Le librerie di `CryptoVerif` possiedono diverse primitive crittografiche predefinite, caricate automaticamente all'inizio dell'esecuzione e utilizzabili dagli utenti stessi. Questa linea di codice consente di sfruttare la funzione DDH, che definisce un gruppo di elementi soddisfacenti l'assunzione decisionale di

Diffie-Hellman. Essa prende in input la probabilità pDDH e i tipi G, Z; ritorna in output la variabile g (generatore del gruppo di elementi di tipo G), la funzione exp (che calcola l'esponenziale) e la funzione mult (che calcola il prodotto fra esponenti).

```
11 fun choose(bool,plain,plain): plain.
12 fun mul(plain,G): G.
```

La parte solitamente più corposa della lista delle dichiarazioni è quella relativa alla descrizione delle funzioni, che ci accingiamo a spiegare. Nel nostro caso, definiamo soltanto due funzioni. La linea 11 illustra la funzione choose che, presi in input un valore booleano e due di tipo plain, ritorna una variabile di tipo plain. La linea seguente descrive una semplice funzione matematica, chiamata mul, che corrisponde in questo caso alla moltiplicazione tra un valore di tipo plain e uno di tipo G.

```
19 equiv
20      foreach i<=ns do x <-R G;
21          o(y:plain) := return(mul(y,x)) [all]
22 <=(0)=>
23      foreach i<=ns do x <-R G; o(y:plain) := return(x).
```

In queste quattro linee di codice viene definita un'equivalenza che riguarda la distribuzione di probabilità della funzione mul vista precedentemente. Questa equivalenza indica che la funzione mul mappa una distribuzione uniforme su una distribuzione uniforme. In parole povere, si ha la stessa probabilità che mul(y,x) sia uguale ad x (e quindi le due return abbiano lo stesso valore) poiché, come vedremo in seguito, y può avere solo due valori (0 e 1).

```
24 query secret b.
```

Questo frammento rappresenta l'ultima dichiarazione. Si vuole provare che la variabile b sia indistinguibile da una qualsiasi stringa casuale. Questo perchè il valore b sarà quello che l'avversario dovrà indovinare per riuscire a forzare lo schema. Passiamo ora alla descrizione dei due oracoli che costituiscono il protocollo: Oracle e OracleV.

```
37 process
38          Oracle() :=
39          b <-R bool;
40          sk <-R Z;
41          vk <- exp(g,sk);
42          return(vk);
43          (foreach n1 <= ns do processV)
```

Oracle è l'oracolo che viene lanciato all'inizio dell'esecuzione; viene chiamato una sola volta e rappresenta le prime azioni compiute dall'autorità per far partire le elezioni. Genera casualmente il bit booleano b, la coppia di chiavi (sk, vk) e ritorna la chiave pubblica, che sarà quindi nota anche all'avversario. La return alla linea 42, infatti, simboleggia proprio la pubblicazione della chiave pubblica dell'autorità sul *bulletin board*, a cui l'attaccante ha accesso. Infine, l'oracolo lancia in parallelo n1 istanze di OracleV.

```
27 let processV =
28         OracleV(v0:plain,v1:plain) :=
29         v <- choose(b,v0,v1);
30         r <-R Z;
31         c <- exp(g,r);
32         d' <- exp(vk,r);
33         d <- mul(v,d');
34         return(c,d).
```

OracleV rappresenta l'elettore. Prende in input due valori di tipo plain, ossia i voti che gli elettori possono esprimere. Il voto viene scelto in base al valore del bit b: se è true, v conterrà il voto v0; altrimenti conterrà v1. Successivamente, il voto viene cifrato e ritornato. Come in precedenza, anche la return presente alla linea 34 corrisponde ad una pubblicazione sul *bulletin board*, in questo caso del voto cifrato.

Note.

1. Dopo la cifratura del voto, il crittogramma ottenuto è restituito all'avversario da OracleV (linea 34). Questo modella la parte in cui il voto cifrato viene pubblicato sul *bulletin board* a cui l'avversario può avere accesso dopo tale pubblicazione. In parole povere, l'avversario può accedere al canale tra l'espressione di un voto e l'altro. Tutto ciò formalizza correttamente il punto dello schema in cui *l'avversario può consultare il bulletin board in qualsiasi momento.*

2. Ci si accorge subito che l'ultima parte dello schema *Minivoting Semplice* (ovvero la somma dei crittogrammi, la decifratura della stessa e il calcolo del risultato finale effettuati dall'autorità) non viene descritta nel protocollo appena presentato. Non è un refuso. Analizzando lo schema di partenza vediamo che l'avversario ha a disposizione tutti i voti v_0 espressi (poiché è proprio lui a decidere il valore dei voti v_0 e v_1 per ogni elettore); è in grado quindi di calcolare autonomamente il risultato delle elezioni (che dovrebbe corrispondere all'output del gioco). Pertanto, l'ultima parte dello schema è stata omessa poiché delegata all'avversario stesso.

3. In questo esperimento non sono stati volutamente considerati gli attacchi *reply* che, invece, il protocollo in [BW14] prende in esame. Nel nostro caso, l'avversario può solamente ottenere informazioni dal *bulletin board*: verranno quindi gestiti soltanto attacchi passivi e non attivi.

Eseguendo il file appena descritto in ambiente `CryptoVerif` si ottiene una sequenza di giochi, che illustreremo nella prossima sezione.

5.2.2 Giochi della Dimostrazione

I *game* che stiamo per presentare sono stati tutti generati in automatico dal dimostratore, tranne il Game 0 che rappresenta il protocollo stesso, e che qui non vedremo poiché è già stato dettagliatamente illustrato nella precedente sezione. Procediamo adesso con la descrizione dei vari *game*.

```
Game 1 is
      Oracle() :=
      b <-R bool;
      sk <-R Z;
      vk: G <- exp(g,sk);
      return(vk);
      foreach n1_89 <= ns do
      OracleV(v0: plain,v1: plain) :=
      v: plain <- choose(b,v0,v1);
      r <-R Z;
      c: G <- exp(g,r);
      d': G <- {28}exp(vk,r);
      d: G <- mul(v,d');
      return(c,d)
```

Il Game 1 è sostanzialmente una scrittura maggiormente compatta del *game* iniziale. Notiamo che è stato cambiato il nome della variabile n1 in n1_-89. Questa ridenominazione ha luogo per evitare situazioni in cui si abbia uno stesso nome per oggetti diversi. Inoltre, è stato inserita la dicitura {28} a metà della terzultima riga, che servirà per l'applicazione di alcune equivalenze nella costruzione del Game 2, mostrato qui di seguito.

```
Game 2 is
      Oracle() :=
      b <-R bool;
      sk <-R Z;
      vk: G <- exp(g, sk);
      return(vk);
```

```
foreach n1_89 <= ns do
OracleV(v0: plain, v1: plain) :=
v: plain <- choose(b, v0, v1);
r <-R Z;
c: G <- exp(g, r);
d': G <- exp(g, mult(sk, r));
d: G <- mul(v, d');
return(c, d)
```

Otteniamo il Game 2 tramite la semplificazione di {28}: exp(vk,r) diventa exp(g, mult(sk, r)) sfruttando la funzione mult (funzione che consente la moltiplicazione tra esponenti) presente nella primitiva crittografica predefinita DDH, espansa nella definizione del protocollo.

```
Game 3 is
    Oracle() :=
    b <-R bool;
    sk_100 <-R Z;
    vk: G <- @5_exp'(g, sk_100);
    return(vk);
    foreach n1_89 <= ns do
    OracleV(v0: plain, v1: plain) :=
    v: plain <- choose(b, v0, v1);
    r_99 <-R Z;
    c: G <- @5_exp'(g, r_99);
    @5_b'_98: Z <- sk_100;
    @5_ka'_97: bitstring <- @5_mark;
    if defined(@5_b'_98, @5_ca_96) then
    (
      d': G <- @5_ca_96;
      d: G <- mul(v, d');
      return(c, d)
    )
    else
      @5_ca_96 <-R G;
      d': G <- @5_ca_96;
      d: G <- mul(v, d');
      return(c, d)
```

Il Game 3 è frutto dell'applicazione dell'equivalenza ddh(exp) data proprio dalla primitiva DDH, con probabilità pari a [probability ns * ns * pDDH(time(context for game 2) + time + (2. * ns + 1.) * time(exp))]. Questa applicazione comporta anche il cambio di nome di alcune variabili.

```
Game 4 is
      Oracle() :=
      b <-R bool;
      sk_100 <-R Z;
      vk: G <- @5_exp'(g, sk_100);
      return(vk);
      foreach n1_89 <= ns do
      OracleV(v0: plain, v1: plain) :=
      v: plain <- choose(b, v0, v1);
      r_99 <-R Z;
      c: G <- @5_exp'(g, r_99);
 {27} if defined(v, @5_ca_96) then
      (
        d: G <- mul(v, @5_ca_96);
        return(c, d)
      )
      else
        @5_ca_96 <-R G;
        d: G <- mul(v, @5_ca_96);
        return(c, d)

Game 5 is
      Oracle() :=
      b <-R bool;
      sk_100 <-R Z;
      vk: G <- @5_exp'(g, sk_100);
      return(vk);
      foreach n1_89 <= ns do
      OracleV(v0: plain, v1: plain) :=
      v: plain <- choose(b, v0, v1);
      r_99 <-R Z;
      c: G <- @5_exp'(g, r_99);
      @5_ca_96 <-R G;
      d: G <- mul(v, @5_ca_96);
      return(c, d)
```

I Game 4 e 5 sono frutto di semplici semplificazioni. Il Game 5, in particolare, si differenzia dal 4 solamente per la rimozione dell'*if-then-else* presente in {27}, la cui *if-condition* è risultata essere *false*.

```
Game 6 is
      Oracle() :=
      b <-R bool;
      sk_100 <-R Z;
```

```
        vk: G <- @5_exp'(g, sk_100);
        return(vk);
        foreach n1_89 <= ns do
        OracleV(v0: plain, v1: plain) :=
        v: plain <- choose(b, v0, v1);
        r_99 <-R Z;
        c: G <- @5_exp'(g, r_99);
        @5_ca_103 <-R G;
        y_102: plain <- v;
        d: G <- @5_ca_103;
        return(c, d)
```

Questo *game* è stato ottenuto applicando l'equivalenza definita nella descrizione del protocollo iniziale tramite il costrutto equiv (si faccia riferimento alla linea 19 nella sottosezione *Descrizione del Protocollo*).

```
Game 7 is
        Oracle() :=
        b <-R bool;
        sk_100 <-R Z;
        vk: G <- @5_exp'(g, sk_100);
        return(vk);
        foreach n1_89 <= ns do
        OracleV(v0: plain, v1: plain) :=
        r_99 <-R Z;
        c: G <- @5_exp'(g, r_99);
        @5_ca_103 <-R G;
        return(c, @5_ca_103)
```

Riconosciamo il Game 7 come ultimo gioco della dimostrazione, ottenuto rimuovendo gli ultimi assegnamenti superflui (ad esempio quello sulla variabile y_102). Notiamo che, dopo le applicazioni delle equivalenze, la variabile @5_ca_103 corrisponde alla variabile d' dei primi giochi.

5.2.3 Risultati

Il risultato restituito in output da `CryptoVerif`, insieme alla sequenza di giochi appena descritta, è il seguente:

```
Proved secrecy of b in game 7

Adv[Game 1: secrecy of b] <= 2. * ns * ns *
pDDH(time(context for game 2) + time +
(2. * ns + 1.) * time(exp)) + Adv[Game 7: secrecy of b]
```

```
Adv[Game 7: secrecy of b] <= 0

RESULT Proved secrecy of b
up to probability 2. * ns * ns *
pDDH(time(context for game 2) + time +
(2. * ns + 1.) * time(exp))

RESULT time(context for game 2) = ns * time(mul) +
ns * time(choose)

All queries proved.
```

Proved secrecy of b in game 7: il dimostratore è riuscito a provare la segretezza del bit b (e quindi la sicurezza del protocollo) costruendo sette giochi. Il secondo frammento indica la probabilità dell'avversario di indovinare il valore del bit b nel Game 1, mentre quello ancora dopo mostra che la probabilità che l'avversario lo indovini nel Game 7 è pari a 0. Notiamo che, ovviamente, queste due probabilità sono diverse. Di conseguenza il risultato riportato dal dimostratore è *all queries proved*: `CryptoVerif` è riuscito a dimostrare la sicurezza del protocollo *Minivoting Semplice*, provando la query richiesta.

Capitolo 6

EasyCrypt

`EasyCrypt` è un framework interattivo per la costruzione di prove dimostranti proprietà di sicurezza crittografiche [BD+16]. Viene presentato per la prima volta nel 2009 da Barthe, Zanella e altri [Zan10], con il nome di `CertiCrypt`. Inizialmente era un tool costruito su CoQ, in grado di mettere a disposizione principi di ragionamento e tecniche crittografiche per dimostrare prove game-based. Negli anni viene perfezionato, e da `CertiCrypt` diventa `EasyCrypt`. Attualmente, questo framework basa le sue potenzialità su diverse *features* quali CoQ, pWhile e pRHL, nonché sulle prove game-based. La parte del capitolo che segue sarà interamente dedicata alla descrizione dettagliata di queste tre *features*. Per le dimostrazioni basate su giochi si faccia riferimento alla sezione 3.2 di questo lavoro.

6.1 CoQ

CoQ, *Calculus of Constructions*, è un proof assistant ancora in fase di sviluppo, inventato da Thierry Coquand nel 1983. Questo tool fornisce un ricco ambiente di sviluppo interattivo per la dimostrazione di teoremi e altre prove formali, strumenti di alto livello per la loro costruzione, tra cui tattiche efficaci ed una vasta libreria di definizioni e lemmi. Inizialmente, CoQ nasce per sviluppare programmi in cui fosse richiesta una fiducia assoluta: per esempio nella telecomunicazione, nel trasporto e nelle operazioni bancarie. In questi casi, il forte bisogno che i programmi si attenessero strettamente alle specifiche giustificava lo sforzo richiesto per verificare che le proprietà di sicurezza definite venissero rispettate. CoQ nasce proprio per rendere questo sforzo il più leggero possibile. È anche un sistema mediante il quale poter sviluppare dimostrazioni basate sulla logica del prim'ordine. CoQ è implementato con il linguaggio OCaml e, per semplificarne l'utilizzo, è stato creato un apposito ambiente di sviluppo: CoQIDE. Il linguaggio di CoQ è Gallina, che consente di rappresentare tipi e termini, tipici dei linguaggi di programmazione. Con questo linguaggio, infatti, è possibile definire

λ-termini e tipi appartenenti al λ-calcolo, nonché effettuare computazioni derivanti da regole di riduzione (come la β-riduzione). Questo sistema di tipi, come vedremo, è stato ereditato da `EasyCrypt`, che possiede espressioni e comandi rigorosamente ben tipati. Nonostante assomigli molto al λ-calcolo tipato semplice, il potere espressivo di CoQ è ben superiore: infatti è anche possibile tipare proposizioni rappresentanti formule di vario genere. Queste possono essere dimostrate solo mediante assunzioni di altre proposizioni, che possono essere definite come ipotesi, assiomi o teoremi. Infine, CoQ possiede una proprietà fondamentale: è un formalismo fortemente normalizzante, quindi qualsiasi computazione giungerà sicuramente al termine. `EasyCrypt` sfrutta CoQ per la formalizzazione della teoria matematica, probabilistica e computazionale, di cui `EasyCrypt` stesso si occuperà di dimostrare determinate proprietà di sicurezza. Maggiori approfondimenti su questo framework possono essere trovati in [AC16] e [BC04].
In molti si chiedono da dove derivi l'acronimo CoQ. Il sito ufficiale del framework [2] dice così: *some French computer scientists have a tradition of naming their software as animal species: Caml, Elan, Foc or Phox are examples of this tacit convention. In French, coq means rooster, and it sounds like the initials of the Calculus of Constructions (CoC) on which it is based.* Il gallo è anche il simbolo nazionale della Francia e c-o-q sono le prime tre lettere del cognome del primo sviluppatore di CoQ, Thierry Coquand.

6.2 pRHL

A differenza di `CryptoVerif`, che genera i giochi automaticamente, `EasyCrypt` si appoggia, tra le altre, sulla *probabilistic Relational Hoare Logic* (pRHL) per dimostrare le transizioni tra giochi. pHRL, illustrata nel dettaglio in [BG+12], è un'estensione probabilistica della logica di Hoare basata sull'espressione di giudizi e su formule relazionali. Quest'ultime sono esprimibili in questa grammatica:

$$\Psi, \Phi ::= e \mid \neg\Phi \mid \Psi \wedge \Phi \mid \Psi \vee \Phi \mid \Psi \Rightarrow \Phi \mid \forall x.\Phi \mid \exists x.\Phi$$

dove e rappresenta un'espressione booleana contenente variabili logiche legate oppure variabili appartenenti a programmi, taggate con $\langle 1 \rangle$ o $\langle 2 \rangle$ per indicare il programma di riferimento . Un esempio di variabile booleana di programma è la keyword *res*, usata per indicare il valore booleano di ritorno di una procedura. In pRHL un giudizio ha la forma seguente:

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$$

dove c_1 e c_2 sono programmi descritti nel linguaggio pWhile (approfondito nella sezione successiva) e Ψ e Φ sono relazioni sulla memoria, chiamate rispettivamente *pre-condition* e *post-condition*. Poiché la valutazione di

un programma pWhile rispetto ad una memoria iniziale produce una sotto-distribuzione su memorie, la verifica di un giudizio pRHL richiede di interpretare le *post-conditions* come relazioni su tali sotto-distribuzioni. A tal fine, pRHL si affida ad un operatore di *lifting* ℓ che trasforma una relazione binaria in una relazione binaria sullo spazio delle sotto-distribuzioni. Possiamo quindi affermare che un giudizio è valido se e solo se

$$\forall m_1,m_2.m_1 \Psi m_2 \Rightarrow (\llbracket c_1 \rrbracket\ m_1)\ \ell(\Phi)\ (\llbracket c_2 \rrbracket\ m_2)$$

ossia per ogni coppia di memorie m_1, m_2 che soddisfa la *pre-condition* Ψ, le due sotto-distribuzioni $(\llbracket c_1 \rrbracket\ m_1)$ e $(\llbracket c_2 \rrbracket\ m_2)$ soddisfano $\ell(\Phi)$. La capacità di derivare affermazioni probabilistiche da giudizi pRHL è essenziale per giustificare il suo utilizzo nella dimostrazione di proprietà di sicurezza nei sistemi crittografici. Osserviamo quindi la seguente proprietà formale:

se

$$\models \mathrm{c}_1 \sim \mathrm{c}_2 : \Psi \Rightarrow \Phi$$

e

$$\Phi \Rightarrow (\langle A \rangle \Leftrightarrow \langle B \rangle)$$

allora

$$\forall m_1,m_2,A,B.m_1 \Psi m_2,\Phi \Rightarrow (A\langle 1 \rangle \Leftrightarrow B\langle 2 \rangle) \Rightarrow Pr[c_1,m_1{:}A] = Pr[c_2,m_2{:}B]$$

cioè, per ogni coppia di memorie m_1,m_2 di giochi differenti e per ogni coppia di eventi A,B tali che se vale Φ tutte le memorie m_1',m_2' che soddisfano A soddisfano anche B e viceversa, la probabilità che accadano i due eventi in relazione alle sotto-distribuzioni indotte è la stessa. Analogamente, se valgono

$$\models \mathrm{c}_1 \sim \mathrm{c}_2 : \Psi \Rightarrow \Phi$$

e

$$\Phi \Rightarrow (\langle A \rangle \Rightarrow \langle B \rangle)$$

allora

$$\forall m_1,m_2,A,B.m_1 \Psi m_2,\Phi \Rightarrow (A\langle 1 \rangle \Rightarrow B\langle 2 \rangle) \Rightarrow Pr[c_1,m_1{:}A] \leq Pr[c_2,m_2{:}B]$$

in cui la seconda proprietà formale indica che tutte le coppie di memorie m_1',m_2' che soddisfano A, soddisfano anche B. Tutte queste proprietà che abbiamo appena analizzato consentono a `EasyCrypt` di ragionare in modo probabilistico sugli eventi riguardanti i giochi della dimostrazione partendo proprio dalla validità e correttezza dei giudizi espressi.

6.3 pWhile

Il linguaggio pWhile è un linguaggio di programmazione imperativo con estensioni probabilistiche, che consente di definire gli schemi e i giochi del protocollo crittografico in esame in `EasyCrypt`. Questo linguaggio consente all'utente di definire tipi, variabili, strutture dati, funzioni, assegnamenti probabilistici, assiomi, lemmi ma anche operatori e tipi specifici per una determinata dimostrazione. Consente anche di modellare schemi e avversari del protocollo, nonché i giochi della dimostrazione stessa. Vediamo nel dettaglio la sintassi del linguaggio: esamineremo soltanto gli elementi usati nella descrizione di *Minivoting Semplice* e nei giochi della dimostrazione (per un ulteriore approfondimento si faccia riferimento a [BC+17]). La sintassi è rappresentata nel modo seguente:

$$\mathcal{C} ::= \text{skip} \mid \mathcal{V} \leftarrow \mathcal{E} \mid \mathcal{V} \xleftarrow{\$} \mathcal{DE} \mid \text{if } \mathcal{E} \text{ then } \mathcal{C} \text{ else } \mathcal{C} \mid \text{while } \mathcal{E} \text{ do } \mathcal{C} \\ \mid \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, ..., \mathcal{E}) \mid \mathcal{C}; \mathcal{C}$$

dove $\mathcal{V}$ è un set di identificatori di variabili, $\mathcal{E}$ è un set di espressioni deterministiche, $\mathcal{DE}$ è un set di espressioni probabilistiche e $\mathcal{P}$ è un set di identificatori di procedura. Con questi insiemi è possibile definire comandi $\mathcal{C}$ che rappresentano, nell'ordine della sintassi appena descritta, assegnamenti deterministici e probabilistici, controlli condizionali, cicli, chiamate a procedura, comandi in sequenza. Definita la sintassi, andiamo ad analizzare gli elementi rappresentabili con il linguaggio pWhile.

- Tipi.
  I tipi di base supportati da `EasyCrypt` sono *unit*, *bool*, *int*, *real*, *bitstring*, *list*, *map*, *sum* e *option*, utilizzabili previa importazione della relativa libreria. Inoltre, il linguaggio pWhile consente di definire nuovi tipi a partire dai nativi, e tipi di dato astratti.

  ```
  type t = int -> bool
  type group
  ```

  In questo esempio vengono definiti un tipo t (che consente di tipare risultati di operazioni tra interi e booleani) e un tipo astratto group (riferito al gruppo ciclico moltiplicativo $\mathbb{Z}^*_p$).

- Operatori.
  Questo linguaggio consente di usufruire di tutti gli operatori aritmetici e logici, e di definirne di nuovi. Un operatore può essere inoltre dichiarato specificando il suo tipo insieme al valore con cui valutarlo. Il comportamento dei nuovi operatori è solitamente descritto tramite assiomi.

```
op x : int = 3
op ( ** ): int -> real -> int
```

Definiamo due nuovi operatori: il primo, x, è di tipo intero e verrà valutato con il valore 3. Il secondo, **, consentirà di calcolare il prodotto tra una variabile intera e una reale, restituendo un valore intero.

- Variabili.
  La dichiarazione di nuove variabili è simile a qualsiasi altro linguaggio di programmazione conosciuto. Le variabili possono anche essere dichiarate senza specificarne subito il tipo, come si può vedere nell'esempio che segue.

```
var num
var count:int
```

  Unica nota: l'identificatore di una variabile è un nome che non può iniziare con la lettera maiuscola.

- Assegnamenti.
  Il linguaggio pWhile consente di effettuare assegnamenti sia deterministici che probabilistici. Gli assegnamenti deterministici sono i classici assegnamenti di espressioni deterministiche a variabili. Quelli probabilistici vengono valutati come distribuzioni uniformi da cui vengono presi casualmente dei valori.

```
x = y + 3
b =$ {0,1}
```

  Il primo assegnamento è deterministico: la variabile x assume il valore dell'espressione y + 3. Il secondo assegnamento è probabilistico: la variabile booleana b può assumere, con la stessa probabilità, il valore 0 oppure il valore 1. Il valore viene scelto casualmente.

- Procedure.
  Le procedure in `EasyCrypt` sono definite all'interno dei moduli (che descriveremo al prossimo punto). Vengono dichiarate usando la parola chiave *proc* e indicando i parametri in input e il tipo del valore di ritorno. Per fare una chiamata ad una procedura bisogna prima specificare il nome del modulo a cui tale procedura appartiene.

```
proc zero-one(b:bool):int = {
    return (b?1:0);
}
val = Mod.zero-one(b);
```

In questo esempio la procedura zero-one prende in input un valore b booleano e, a seconda se esso sia *true* o *false*, ritorna, rispettivamente, l'intero 1 o l'intero 0. Notiamo, come detto, che nella chiamata della procedura zero-one, il nome della procedura è preceduto dal nome del modulo che la contiene (in questo esempio il modulo Mod).

- Moduli.
  I moduli in questo linguaggio possono descrivere funzioni aritmetiche, primitive crittografiche, giochi di dimostrazioni o avversari. Un modulo può contenere implementazioni di procedure (con le proprie variabili locali) o soltanto la loro dichiarazione, variabili globali (visibili quindi da ogni procedura all'interno del modulo) e chiamate a procedure di un altro modulo. Unica nota, il nome di un modulo non può iniziare con la lettera minuscola.

```
module Simple = {
    var z:int;
    proc inc(x:int):int = {
        return x+1;
    }
    proc dec(sk:skey,c:cipher):plain
}
```

  Queste linee di codice descrivono il modulo Simple, che ha una variabile globale z (di tipo intero) e due procedure: inc, che ritorna il valore preso in input incrementato di 1; dec, dichiarata ma non implementata, che presi una chiave segreta sk e un crittogramma c ne ritorna il corrispondente messaggio decifrato.

- Assiomi.
  Gli assiomi sono proposizioni ritenute vere da `EasyCrypt` (e quindi assunte). Consentono, tra le altre cose, di definire il comportamento di un operatore oppure di specificare alcune proprietà aritmetiche. Essendo assunti come veri, gli assiomi non devono essere dimostrati, ma possono essere utilizzati nelle prove dei lemmi.

```
axiom sym: forall(x y:int), x = y => y = x.
axiom grt: n > 0.
```

  L'assioma sym rappresenta una proprietà matematica, cioè se x = y allora vale anche il viceversa. L'assioma grt, invece, descrive il comportamento dell'operatore n: assumiamo come vero che n sia sempre maggiore di 0.

- Lemmi.
  Al contrario degli assiomi, i lemmi sono proposizioni da dimostrare. Descrivono ciò che deve essere verificato per dimostrare la sicurezza dello schema in analisi. Possono riferirsi a schemi, giochi ed avversari, e vengono dimostrati attraverso una sequenza di tattiche, che descriveremo approfonditamente nel prossimo paragrafo.

  ```
  lemma prob &m:
    Pr[First.main() @ &m : res] =
    Pr[Second.main() @ &m : res].
  ```

  Questo lemma, prob, indica che la probabilità che lo schema First, con memoria m, restituisca in output il valore res è uguale alla probabilità che lo schema Second, con la stessa memoria m, restituisca in output lo stesso valore res.

  I lemmi vengono dimostrati in `EasyCrypt` mediante l'ausilio di tattiche differenti, che consentono di manipolare e gestire tali dimostrazioni.

6.4 Tattiche

Per gestire le transizioni logiche e le dimostrazioni dei lemmi, `EasyCrypt` mette a disposizione dell'utente un potente *tactic language*. Esso consente principalmente di poter ridurre i giudizi espressi in pRHL a verification conditions risolvibili automaticamente dai solvers SMT. Una volta definito un giudizio, quindi, l'interprete di `EasyCrypt` si aspetta una serie di tattiche per guidare e completare la dimostrazione. L'applicazione di ogni tattica ad un giudizio può generare sia dei *subgoals* (che devono essere dimostrati a loro volta mediante l'uso di ulteriori tattiche), che una serie di *verification conditions* espresse nella logica del prim'ordine, risolvibili invocando i solvers SMT. Avremo quindi una dimostrazione nella forma seguente:

lemma.
proof.
sequenza di tattiche.
qed.

Le parole chiave proof e qed consentono, rispettivamente, di iniziare e terminare (e quindi salvare) la dimostrazione. Quando interpretati, i lemmi vengono automaticamente trasformati da `EasyCrypt` in un goal che ha la forma seguente:

pre = pre-condition
(n) schemi e giochi coinvolti
post = post-condition

I goals sono visualizzabili sul terminale al momento dell'interpretazione del file '*.ec': contengono la *pre-condition* e la *post-condition*, insieme alle procedure degli schemi e dei giochi coinvolti nel giudizio espresso nel lemma. Per risolvere questi goals, `EasyCrypt` mette a disposizione un insieme molto disparato di tattiche: in questo elaborato analizzeremo solamente quelle usate per dimostrare la sicurezza di *Minivoting Semplice*. Andiamole ad illustrare.

- `byphoare` (_: $P \Rightarrow Q$).
  Questa tattica viene usata quando la conclusione del goal ha la forma seguente:

  $$\Pr[M.p(a_1,...,a_n) \ @ \ \&m : E] = e.$$

  Riduce il goal iniziale in tre subgoals:

  - uno con conclusione `phoare`$[M.p : P \Rightarrow Q] = e$;
  - uno la cui conclusione afferma che P è verificato, i riferimenti alle variabili sono presi dalla memoria m e i riferimenti ai parametri formali di $M.p$ sono stati sostituiti dagli argomenti in P;
  - uno la cui conclusione afferma che $Q \Leftrightarrow E$.

  Uno o entrambi i parametri P e Q possono essere sostituiti dal carattere '_' in modo da essere entrambi oggetto di verifica (come nel nostro caso).

- `proc`.
  Trasforma un goal la cui conclusione è un giudizio in pRHL, pHL o HL che coinvolge delle procedure concrete, in un giudizio in cui tali procedure vengono sostituite dai loro *bodies*. Asserzioni riguardanti il risultato *res* vengono sostituite con il valore di ritorno delle procedure di riferimento.

- `inline` $M_1.p_1,...,M_n.p_n$.
  Esegue l'*inline* delle procedure concrete selezionate, finchè è possibile. Quando si esegue l'*inline* di una chiamata a procedura, ai parametri formali di tale procedura sono assegnati quelli attuali (per evitare conflitti, se necessario, vengono usate nuove variabili, con nomi diversi). In seguito si ha il corpo della procedura, il cui valore di ritorno viene assegnato alla variabile designata dalla procedura chiamante. Notiamo che, per eseguire l'*inline* di tutte le procedure coinvolte non importa elencarle tutte: si può usare il simbolo *.

- `while` I.
  Questa tattica si basa su un'invariante I, che può riferirsi ad una o più

variabili del game. Se la conclusione del goal è un giudizio in pRHL, pHL o HL che termina con un ciclo *while*, questa tattica riduce tale goal in due subgoals:

  - uno contenente il corpo del ciclo *while* in analisi, in cui la *pre-condition* è la congiunzione di I con la condizione del ciclo *while*, e la *post-condition* è I;
  - uno in cui la *pre-condition* è la stessa del goal di partenza, il corpo è dato dalla rimozione del ciclo *while* nel corpo di partenza e la *post-condition* è la congiunzione di I con l'asserzione che: per ogni variabile modificata nel ciclo *while*, se la condizione del ciclo non è verificata, ma I è verificata, allora la *post-condition* del goal di partenza è verificata.

- `splitwhile` c : e.
  Se la conclusione del goal è un giudizio in cui alla c-esima linea è presente un ciclo *while*, ed e è un'espressione booleana ben tipata contestualizzata nel ciclo, questa tattica inserisce prima del *while* una copia del ciclo stesso, in cui e è aggiunta in congiunzione con l'espressione booleana già presente nella condizione del ciclo.

- `rcondt` n.
  Se la conclusione del goal è un giudizio in pRHL o HL in cui l'n-esima linea contine un ciclo *while*, questa tattica riduce il goal di partenza in due subgoals:

  - uno in cui la conclusione è un giudizio in pRHL o HL, in cui la *pre-condition* è uguale a quella del goal di partenza, il corpo è dato dalle prime *n-1* linee del corpo di partenza, e la *post-condition* è la condizione del ciclo *while*, sotto forma di espressione booleana;
  - uno in cui la conclusione è un giudizio in pRHL o HL e l'unica differenza con il goal di partenza è data dal ciclo *while*, che è stato sostituito dal suo corpo.

- `rcondf` n.
  Questa tattica è molto simile alla precedente. Se la conclusione del goal è un giudizio in pRHL o HL in cui l'n-esima linea contine un ciclo *while*, questa tattica riduce il goal di partenza in due subgoals:

  - uno in cui la conclusione è un giudizio in pRHL o HL, in cui la *pre-condition* è uguale a quella del goal di partenza, il corpo è dato dalle prime *n-1* linee del corpo di partenza, e la *post-condition* è la condizione del ciclo *while*, sotto forma di espressione booleana;
  - uno in cui la conclusione è un giudizio in pRHL o HL e l'unica differenza con il goal di partenza è data dal ciclo *while*, che è stato rimosso completamente.

- `wp`.
  Se la conclusione del goal è un giudizio in pRHL, pHL o HL, questa tattica calcola la *weakest pre-condition*: consente di gestire in automatico gli assegnamenti deterministici, partento dal bottom e risalendo fino al top o, se specificato, fino alla posizione *pos*. Alla *post-condition* di partenza verrà quindi aggiunta anche la *weakest pre-condition* calcolata.

- `sp`.
  Questa tattica è simile alla precedente. Anch'essa vale con giudizi in pRHL, pHL e HL, e consente di calcolare la *strongest post-condition* gestendo in automatico assegnamenti deterministici. Al contrario della tattica `wp`, questa parte dal top e scende fino al bottom (oppure, se specificato, fino alla posizione *pos*). Alla *pre-condition* di partenza verrà quindi aggiunta la *strongest post-condition* calcolata.

- `rnd`.
  L'uso di questa tattica considera soltanto assegnamenti in cui la parte sinistra consiste in un'unica variabile. Se il programma di riferimento termina con un assegnamento random, questa tattica sostituisce la *post-condition* con la *weakest pre-condition* probabilistica generata dall'assegmento casuale. Sottolineiamo che la tattica è applicabile soltanto ad assegnamenti situati alla fine del programma.

- `seq` n *:* R.
  Se n è un numero naturale e la conclusione del goal è un giudizio con *pre-condition* P, *post-condition* Q tale che la lunghezza del programma sia almeno pari ad n, questa tattica riduce il goal di partenza in due subgoals:

  - il primo subgoal ha come *pre-condition* P, come *post-condition* R e il corpo del programma consiste nelle prime n linee di codice del programma di partenza;
  - il secondo subgoal ha come *pre-condition* R, come *post-condition* Q e il corpo del programma consiste nelle restanti linee di codice del programma di partenza.

- call (_:I).
  Se il programma di riferimento termina con una chiamata ad una procedura *concreta*, questa tattica usa l'invariante I come parametro attuale ed automaticamente applica la tattica `proc` (descritta in precedenza) sulla procedura. L'invariante può assumere anche valore *true*, nel caso in cui la funzione non abbia parametri oppure quest'ultimi, ad esempio, siano definiti in maniera casuale. Notiamo che lo stesso procedimento sarà applicato alle chiamate di procedure *astratte*.

- `auto`.
  Questa è una delle tattiche automatizzate. Si usa in presenza di un qualunque giudizio in pRHL, pHL o HL. Sfrutta un insieme di tattiche logiche nel tentativo di ridurre il goal corrente in un goal semplificato, dimostrabile più facilmente. Notiamo che questa tattica non può fallire, ma talvolta può non avere alcun effetto e lasciare il goal inalterato.

- `progress`.
  Anche questa è una tattica che fa parte delle automatizzate, e viene usata per ridurre il goal corrente in uno più semplice. Quando usata, questa applica in maniera ripetitiva le tattiche `split`, `subst` e `move` (queste tre tattiche non verranno descritte: per approfondirle si faccia riferimento al manuale indicato in fondo a questa sezione). Come la tattica `auto`, anche questa non può fallire, ma può non avere effetto lasciando il goal inalterato.

- `smt`.
  Questa tattica automatizzata viene applicata per cercare di risolvere il goal corrente, e lo fa mediante l'uso di SMT solvers. Il goal viene fornito a questi solvers insieme alle ipotesi locali effettuate, agli assiomi ed agli eventuali lemmi dimostrati. Per risolvere il goal, gli SMT solvers usano gli input ricevuti ed i loro algoritmi interni.

Per ulteriori approfondimenti circa le tattiche descritte e non, si faccia riferimento al manuale contenuto in [BC+17].

6.5 Minivoting Semplice in EasyCrypt

Anche la seconda parte di questo lavoro si dedica alla dimostrazione delle proprietà di sicurezza di *Minivoting Semplice* (sezione 4.1). Tale dimostrazione si appoggia sul framework `EasyCrypt` descritto ampiamente nelle sezioni precedenti. Questo capitolo si occupa di illustrare nel dettaglio il protocollo in analisi. Seguiranno poi i giochi della dimostrazione e i lemmi da provare. Infine ci dedicheremo all'analisi dei risultati ottenuti.

6.5.1 Descrizione del Protocollo

Il file che descrive il protocollo inizia con l'importazione delle librerie necessarie alla definizione e manipolazione di schemi, avversari, giochi e lemmi del protocollo stesso.

```
1 require import Int.
2 require import Real.
3 require import Bool.
```

```
 4 require import FMap.
 5 require import FSet.

 7 require PKE.
 8 require DiffieHellman.
 9 clone import DiffieHellman as DH.
10 import DDH.
```

Importiamo quindi le librerie adatte a manipolare, nell'ordine, interi, reali, booleani, mappe e insiemi. Inoltre, le *import* specificate nelle ultime linee consentono di importare la teoria (quindi schemi, avversari e giochi) della primitiva *PKE* (schema di crittografia asimmetrica) e dello schema relativo al problema decisionale di *Diffie-Hellman*.

```
12 type pkey = group.
13 type skey = F.t.
14 type plaintext = group.
15 type ciphertext = group * group.

17 clone import PKE as PKE_ with
18 type pkey <- pkey,
19 type skey <- skey,
20 type plaintext <- plaintext,
21 type ciphertext <- ciphertext.
```

Le prime linee indicano i tipi che ci serviranno nella definizione del nostro schema. Avremo quindi i tipi pkey e plaintext che corrispondono al tipo group; ciphertext corrisponde al prodotto di due elementi di tipo group; skey ha tipo *PrimeField*, cioè un tipo che può ammettere un numero finito di elementi, con distribuzione uniforme. Le linee dalla 17 alla 21, invece, operano una copia dei tipi dello schema di crittografia a chiave pubblica *PKE*, importato precedentemente. Questo consente di poter usare *PKE* come base per la costruzione di *Minivoting Semplice*, che si appoggia, tra le altre, su questa primitiva.

```
23 op n:int.
24 axiom l0n: 0 < n.
```

Sfruttando la possibilità di creare nuovi operatori, definiamo il parametro di sicurezza n previsto da *Minivoting Semplice* e, mediante l'assioma l0n, stabiliamo che esso debba sempre essere superiore a zero.

```
26 module M: Scheme = {

28   proc kg():pkey*skey = {
29     var sk,vk;
30     sk = $FDistr.dt;
31     vk = g^sk;
```

```
32      return(vk,sk);
33    }

35    proc enc(vk:pkey,v:plaintext):ciphertext = {
36      var c,d,r;
37      r = $FDistr.dt;
38      c = g^r;
39      d = (vk^r)*v;
40      return (c,d);
41    }

44    proc dec(sk:skey,s:ciphertext):plaintext option = {
45      var c,d,result;
46      (c,d) = s;
47      result = d*(c^(-sk));
48      return Some (result);
49    }
50  }.
```

Questo blocco descrive la costruzione vera e propria del protocollo *Minivoting Semplice*. Per definire lo schema si usa la parola chiave module, seguita dal nome dello schema (nel nostro caso M). Le procedure kg, enc e dec (precedute nella definizione dalla parola chiave proc), rappresentano le funzioni di generazione delle chiavi, cifratura e decifratura usate in *Minivoting Semplice*. Non le descriviamo nuovamente: si possono trovare nella sezione 4.1. Il blocco di codice appena analizzato è seguito dalla descrizione dello schema *ElGamal*, che verrà sfruttato nella dimostrazione della sicurezza di *Minivoting Semplice* (che vedremo in seguito); qui tale descrizione viene volutamente omessa, poiché la si può trovare nel repository messo a disposizione in [5].

```
72  module type Adversary = {
73    proc choose(pk:pkey) : plaintext * plaintext
74    proc guess_first(f:ciphertext) : plaintext * plaintext
75    proc guess_last(f:ciphertext) : bool
76  }.
```

Dopo aver costruito il protocollo *Minivoting Semplice*, bisogna definire il suo avversario: il module sarà quindi di tipo Adversary, e conterrà le procedure astratte che l'avversario potrà invocare durante i giochi della dimostrazione. Nel nostro caso, tali procedure sono tre: choose, che consente all'avversario di scegliere il valore dei due voti avendo a disposizione la chiave pubblica; guess_first e guess_last che permettono all'avversario di cercare di indovinare, rispettivamente, i due voti reali e il bit b, avendo a disposizione il crittogramma relativo al voto espresso nel round corrente. L'uso pratico

di queste tre procedure verrà descritto nella prossima sezione, quando si parlerà dei giochi della dimostrazione.

6.5.2 Giochi della Dimostrazione

La costruzione del protocollo *Minivoting Semplice* è seguita dalla definizione dei giochi e dei lemmi della dimostrazione. Entrambi fanno parte di una particolare sezione, detta *Security*. Dopo aver aperto tale sezione, e dichiarato l'avversario con l'apposita dicitura *declare module*, si può proseguire con la definizione dei giochi. Nel nostro caso sono due: illustriamo il primo qui di seguito, mentre il secondo viene omesso poiché a disposizione nel repository in [5].

```
82  local module Game = {
83
84      proc main():bool = {
85        var v0,v1:plaintext;
86        var b,b_adv:bool;
87        var vk:pkey;
88        var sk:skey;
89        var c,d:group;
90        var i = 0;

92        (vk,sk) = M.kg();
93        b =$ {0,1};
94        (v0,v1) = A.choose(vk);
95        while (i < n) {
96          (c,d) = M.enc(vk,b?v1:v0);
97          (v0,v1) = A.guess_first(c,d);
98          i = i + 1;
99        }
100       (c,d) = M.enc(vk,b?v1:v0);
101       b_adv = A.guess_last(c,d);
102       return(b_adv = b);
103     }
104 }.
```

Questo gioco descrive l'esperimento relativo alla sicurezza di *Minivoting Semplice* (sezione 4.1). La parola chiave *local* che precede la definizione del modulo indica che il gioco è locale alla sezione *Security* aperta in precedenza. Le prime linee dichiarano ogni variabile assegnando a ciascuna il proprio tipo. Viene poi creata la coppia di chiavi pubblica e privata e scelto il bit b in maniera casuale. L'avversario, poi, sceglie il valore dei due voti iniziali v_0 e v_1, chiamando la funzione choose. Il ciclo *while* rappresenta il concetto del *bulletin board*, che contiene tutti i crittogrammi pubblicati

dagli elettori: in qualsiasi momento l'avversario può consultarlo e scegliere di cambiare il valore dei due voti v_0 e v_1, mediante la funzione guess_first. Infine, dopo che l'ultimo elettore ha espresso il proprio voto, l'avversario indica, con la funzione guess_last, quale sia secondo lui l valore del bit b. Il gioco ritorna *true* se l'avversario lo indovina, *false* altrimenti. Osserviamo che la parte della decifratura e del conteggio finale dei voti è stata omessa: come già visto, l'avversario ha a disposizione tutti i voti v_0 espressi, poiché è proprio lui a decidere i valori di v_0 e v_1 per ogni elettore. Sarà in grado, quindi, di calcolare in autonomia il risultato delle elezioni. Notiamo inoltre che le funzioni kg, enc, dec sono le tre procedure dello schema M definito all'inizio del protocollo; le funzioni choose, guess_first, guess_last sono le tre procedure del modulo avversario. Il secondo esperimento, qui omesso, è relativo alla sicurezza dello schema *ElGamal*. Come detto, la descrizione di questo esperimento si può trovare in [5]. Prima di costruire e dimostrare i lemmi relativi alla sicurezza del protocollo *Minivoting Semplice*, dobbiamo accertarci che lo schema M definito e lo schema EG (*ElGamal*, volutamente omesso) siano corretti.

```
122 hoare Correctness: Correctness(M).main: true ==> res.
123 proof.
124 proc; inline*; auto; progress; algebra. qed.
```

Per definire e dimostrare la correttezza dello schema M esprimiamo un giudizio nella logica di *Hoare*. La linea 122 descrive proprio tale giudizio, che viene trasformato da `EasyCrypt` nel seguente goal:

```
pre = true

(1) (pk, sk) = M.kg()
(2) c = M.enc(pk, m)
(3) m' = M.dec(sk, c)

post = m' = Some m
```

(1), (2) e (3) indicano i valori di ritorno delle tre procedure dello schema M indicate. La parola chiave post indica la *post-condition*: lo schema M sarà corretto solo se il messaggio m' ottenuto con la procedura M.dec sarà uguale al messaggio m cifrato dalla procedura M.enc, sfruttando le chiavi sk e vk generate dalla procedura M.kg. Dimostriamo questo giudizio usando le tattiche proc, inline*, auto, progress e algebra, visibili nel blocco di codice precedente (linea 124). Precisiamo che, per dimostrare la correttezza dello schema EG, si è seguita la stessa metodologia qui descritta: non la riportiamo per evitare ridondanza dei concetti.

6.5.3 Lemmi e Risultati

La parte dimostrativa delle proprietà di sicurezza di *Minivoting Semplice* inizia con la definizione di alcuni assiomi.

```
130 axiom Ac : islossless A.choose.
131 axiom Agf : islossless A.guess_first.
132 axiom Agl : islossless A.guess_last.
133 axiom Mk : islossless M.kg.
134 axiom EGk : islossless EG.kg.
135 axiom Me : islossless M.enc.
136 axiom EGe : islossless EG.enc.
```

Questi assiomi, che saranno sfruttati nelle dimostrazioni descritte in questa sezione, si riferiscono agli algoritmi di generazione delle chiavi e di cifratura degli schemi M ed EG, e alle tre funzioni utilizzabili dall'avversario A. Precisamente, indicano che A.choose, A.guess_first, A.guess_last, M.kg, EG.kg, M.enc ed EG.enc termineranno sempre restituendo il risultato atteso, indicato dalle proprie *return*. Questo concetto è riassunto dalla parola chiave *islossless*, usata nella definizione di tali assiomi. Vediamo ora tre lemmi, che serviranno nella dimostrazione della sicurezza di *Minivoting Semplice*. Non descriveremo nuovamente il funzionamento di ogni tattica usata: tale descrizione si può trovare nella sezione 6.4.

```
138 local lemma EG_half &m:
139 Pr[GameEG.main() @ &m : res] = 1%r/2%r.
```

Il primo lemma, denominato EG_half, vuole dimostrare che nessun avversario $\mathcal{A}$ PPT possa vincere il gioco GameEG (relativo allo schema *ElGamal* e descritto nel repository in [5]) con probabilità superiore a $\frac{1}{2}$. Le tattiche `byphoare` (indicante che stiamo dando un giudizio nella logica HL) e `proc`, consentono di impostare la dimostrazione, e ottenere il seguente goal iniziale:

```
pre = true

(1) z =$ FDistr.dt
(2) y =$ FDistr.dt
(3) (pk, sk) = EG.kg()
(4) (m0, m1) = A.choose(pk)
(5) (gz, gy) = EG.enc(g^z, g^y)
(6) b' = A.guess_last(gz, gy)
(7) b =$ {0,1}

post = b' = b
```

L'assegnamento random in (7) verrà gestito con la tattica `rnd`, mentre le tattiche `call` Agl, `call` EGe, `call` Ac e `call` EGk consentiranno di indicare

al dimostratore che le funzioni da (3) a (6) termineranno tutte con il risultato atteso. Infine, le tattiche `auto` e `progress` gestiranno gli assegnamenti deterministici rimasti, producendo questo subgoal:

```
mu FDistr.dt (fun (x : t) => true) = 1%r
```

Questo si verifica dimostrabile con la tattica `smt`, che invocherà gli SMT solvers e genererà a sua volta il goal seguente:

```
mu {0,1} (fun (x : bool) => result = x) = 1%r/2%r
```

Ammetteremo quest'ultimo goal con `admit` per concludere la prima dimostrazione. Vediamo ora la seconda delle tre.

```
155 local lemma M_EG &m:
156 Pr[Game.main() @ &m : res] =
157           Pr[GameEG.main() @ &m : res].
```

Con questo secondo lemma, denominato M_EG, vogliamo dimostrare l'equivalenza tra Game (il gioco relativo al protocollo *Minivoting Semplice*) e GameEG (il gioco relativo allo schema *ElGamal*). Per impostare la dimostrazione usiamo le tattiche `byequiv` (indicante che il giudizio espresso rappresenta un'equivalenza) e `proc`. Viene quindi generato il seguente goal:

```
pre = (glob A){2} = (glob A){m} /\
      (glob A){1} = (glob A){m}
```

```
(1) i = 0                                  z =$ FDistr.dt
(2) (vk, sk) = M.kg()                      y =$ FDistr.dt
(3) b =$ {0,1}                             (pk,sk) = EG.kg()
(4) (v0, v1) = A.choose(vk)                (m0,m1) = A.choose()
(5) while (i < n){                         (gz,gy) = EG.enc(g^z,g^y)
      (c,d) = M.enc(vk,b?v1:v0)             b' = A.guess_last(gz,gy)
      (v0,v1) = A.guess_first(c,d)          b =$ {0,1}
                i = i + 1
    }
(6) (c, d) = M.enc(vk, b?v1:v0)
(7) b_adv = A.guess_last(c, d)
```

```
post = (b_adv{1} = b{1}) = (b'{2} = b{2})
```

Le etichette {1} e {2} che troviamo nella *pre-condition* e nella *post-condition* indicano, rispettivamente, la memoria di Game e la memoria di GameEG. La *pre-condition* illustra che le variabili globali dei due game devono essere equivalenti; notiamo anche che la *post-condition* indica il risultato da ottenere alla fine della dimostrazione, cioè che il bit b_adv ritornato in Game, deve essere uguale al bit b' ritornato dall'avversario in GameEG (altrimenti i due giochi non potrebbero considerarsi equivalenti). Il blocco di codice

situato tra la *pre-condition* e la *post-condition* descrive i due giochi coinvolti nella dimostrazione: la parte destra si riferisce a Game (ossia quello relativo al protocollo *Minivoting Semplice*), mentre quella a sinistra a GameEG (relativo allo schema *ElGamal*). Iniziamo la dimostrazione analizzando il ciclo *while* in (5): con la sequenza di tattiche `rcondf{1} 5`, `progress`, `call(_:true)`, `rnd`, `call(_:true)`, `auto`, `auto`, `progress`, `admit`, prendiamo in esame il corpo del ciclo. In particolare, assegniamo indirettamente il valore 1 alla variabile n, in modo da analizzare il corpo del ciclo una sola volta, ed eliminare successivamente il ciclo *while* dalla dimostrazione. Siccome nel nostro caso il corpo del ciclo non contiene costrutti condizionali (come l'*if-then-else*), possiamo esaminarlo una sola volta (in caso contrario, basterebbe ripetere la sequenza di tattiche di cui sopra un numero n di volte). In seguito, utilizziamo la tattica `swap{2} 7 -3`, quindi su GameEG, per spostare l'assegnamento random da (7) a (4), in modo da riuscire poi a gestirlo insieme a quello presente in Game. L'uso in sequenza delle tattiche `call{1}` Agl, `call{2}` Agl, `call{1}` Me, `call{2}` EGe, `auto`, `call{1}` Ac, `call{2}` Ac, consente di gestire le funzioni restanti, comunicando al dimostratore che esse termineranno con il risultato atteso. Viene quindi generato questo blocco di codice:

```
(1) i = 0                       z =$ FDistr.dt
(2) (vk, sk) = M.kg()           y =$ FDistr.dt
(3) b =$ {0,1}                  (pk,sk) = EG.kg()
(4)                             b =$ {0,1}
```

I due assegnamenti random presenti vengono gestiti (obbligatoriamente) insieme, tramite l'utilizzo della tattica `rnd`. `call{1}` Mk e `call{2}` EGk gestiscono le due funzioni di generazione delle chiavi, come nelle chiamate viste in precedenza. Infine, la sequenza `auto` (che consente di eliminare gli assegnamenti deterministici rimasti), `progress` e `smt` produce quest'ultimo goal, che ammetteremo veritiero con la tattica `admit`:

```
(result0 = bL) = (result = bL)
```

Notiamo che bL è una variabile booleana, result0 è l'output di Game e result è l'output di GameEG. Chiudiamo la dimostrazione della sicurezza di *Minivoting Semplice* con un ultimo lemma, presentato qui di seguito.

```
180 local lemma M_security &m:
181 Pr[Game.main() @ &m : res] = 1%r/2%r.
```

Con questo lemma, denominato M_security, vogliamo dimostrare che nessun avversario $\mathcal{A}$ PPT possa vincere il gioco Game (relativo allo schema *Minivoting Semplice*) con probabilità superiore a $\frac{1}{2}$. Lo dimostriamo con la tattica `by rewrite` (M_EG &m) (EG_half &m), che sfrutta i due lemmi appena provati per concludere la dimostrazione di M_security, in analisi.

I risultati ottenuti sono quelli desiderati. Siamo riusciti a dare una prova formale e rigorosa della sicurezza del protocollo di voto elettronico *Minivoting Semplice* mediante l'ausilio del dimostratore semiautomatico `EasyCrypt`. Tuttavia, dobbiamo considerare la possibilità di uno sviluppo futuro per sfruttare meglio il *tactic language* fornito dal dimostratore e rendere questa prova ancora più precisa. In particolare, si dovrà cercare di sostituire la tattica `admit`, laddove usata, con la tattica `smt`, che consente di sfruttare gli SMT solvers e rendere la dimostrazione maggiormente corretta.

Capitolo 7

Conclusioni

L'obiettivo di questo lavoro di tesi era quello di dimostrare, prima in maniera automatica e poi in modo semiautomatico, la sicurezza del protocollo di voto elettronico *Minivoting* nella sua versione più semplice, non prima di averlo formalmente descritto. Si è voluta dare una dimostrazione rigorosa e formale della sicurezza del protocollo *Minivoting*, non presente nell'articolo di Bernhard e Warinschi [BW14], che descrive proprio questo protocollo ma in maniera più informale e meno dettagliata.

Innanzitutto, si è cercato di fornire una base teorica sufficientemente solida al fine di riuscire a comprendere a pieno il funzionamento di *Minivoting*; si è parlato quindi di crittografia moderna e del modello computazionale da essa introdotto, di crittografia a chiave pubblica omeomorfa e di schemi di codifica *threshold* (in particolare, anche degli schemi *ElGamal* e *Threshold ElGamal*). La descrizione formale del protocollo elettronico *Minivoting Semplice* è seguita dalla dimostrazione della sua sicurezza, prima in chiave teorica e poi pratica (sfruttando i dimostratori automatico e semiautomatico). Prima di dare la dimostrazione pratica, si sono analizzati `CryptoVerif` e `EasyCrypt`, due *tool* adatti per provare, a livello sperimentale, proprietà di sicurezza di un qualsiasi protocollo crittografico. In questa tesi è stato trattato soltanto un sottoinsieme della teoria legata a questi due *tool*, in modo da non dilungarsi in argomenti non utili alla comprensione del lavoro svolto. In seguito, si è data la dimostrazione della sicurezza di *Minivoting* sfruttando questi due strumenti.

I risultati ottenuti possono considerarsi soddisfacenti. Teoricamente, abbiamo provato che *Minivoting* nella versione semplice è CPA-sicuro contro avversari PPT, ossia che nessun avversario riesce a forzare lo schema proposto in tempo polinomiale, con probabilità superiore a $\frac{1}{2} + \varepsilon(n)$. Questa proprietà di sicurezza è stata dimostrata in maniera più solida parametrizzando *Minivoting Semplice* sullo schema omeomorfo *ElGamal*. In particolare, questo schema specifico è risultato essere sicuro poiché la sua sicurezza si appoggia sulla complessità del problema decisionale di *Diffie-Hellman*. An-

che a livello sperimentale i risultati ottenuti sono quelli desiderati. Il primo strumento utilizzato, `CryptoVerif`, non assume un input particolarmente dettagliato, ma è comunque riuscito a dimostrare la sicurezza del protocollo richiesta. Tuttavia, siccome `CryptoVerif` ha basso potere espressivo in quanto si comporta come una *black box*, per dimostrare in maniera più solida la sicurezza di *Minivoting Semplice* si è pensato di utilizzare un secondo strumento, `EasyCrypt`. `EasyCrypt` è un dimostratore potente, che richiede un input molto dettagliato e preciso: anche con questo *tool* siamo riusciti a dimostrare la proprietà di sicurezza richiesta. Notiamo che, nella costruzione del *game* in entrambi gli strumenti è stato delegato all'avversario il calcolo del risultato finale delle elezioni, essendo l'avversario stesso a decidere il valore dei voti v_0 e v_1 con cui far votare ogni elettore. Specifichiamo inoltre che questo lavoro considera avversari passivi, perciò non gestisce attacchi attivi di tipo *replay*.

In conclusione, la riuscita dimostrazione della sicurezza di *Minivoting* nella sua versione più semplice consente di considerarlo un buon punto di partenza sul quale costruire il protocollo di voto elettronico *Helios*, in fase di sperimentazione in alcune città Europee (si veda, ad esempio, l'uso di *Helios* per le elezioni del Presidente dell'Università di Lovaino, in Belgio [AD+09]). Tuttavia, il grande punto debole di *Minivoting Semplice* è che la sua sicurezza si basa anche sull'onestà dell'unica autorità che prende parte al processo delle elezioni: in disonestà questa potrebbe decifrare i voti degli elettori violandone la privacy o anche alterare il risultato delle elezioni poiché non controllato da nessun'altra entità. È per questo motivo che, nelle versioni *Minivoting Threshold* e *Minivoting Zero-Knowledge* (più solide in termini di sicurezza), vengono introdotte nuove primitive crittografiche che consentono di arginare queste problematiche. Lavori futuri verteranno sicuramente sulla dimostrazione della sicurezza di queste due versioni di *Minivoting* (qui trattate solo teoricamente), per poi giungere alla dimostrazione della sicurezza del protocollo di voto elettronico "finale" *Helios*.

Ringraziamenti

Penso che la parola "grazie" sia una delle più belle che si possano pronunciare: ripaga dell'aiuto, dei sorrisi, degli incoraggiamenti e di tutto il sostegno ricevuto: io mi ritengo una persona fortunata, poiché mi è stato donato tutto questo.
Il mio grazie, quindi, lo voglio dedicare prima di tutto alla mia preziosa famiglia, che ha saputo incoraggiarmi a non mollare quando le cose non andavano per il verso giusto, ma anche spronarmi a dare il massimo in ogni occasione: non potevo desiderare sostegno e famiglia migliore di quella che ho. Grazie per la pazienza, per avermi sopportata quando ero nervosa o in ansia, e per la grande fiducia che riponete in me e che io, ogni giorno, cerco di meritare.
Un grazie immenso va al mio relatore, dott. Ugo Dal Lago, che con la sua pazienza e, soprattutto, i suoi ottimi consigli ha saputo guidarmi nella giusta direzione, aiutandomi a raggiungere i miei obiettivi. Grazie per avermi dedicato il suo tempo, e avermi fatto da guida nella conclusione del mio percorso di studi.
Un grazie doveroso anche alle mie amiche di sempre, per avermi ascoltata ed aiutata quando ero in difficoltà, e per essere le perfette compagne di risate e divertimento. Grazie per le serate trascorse insieme lontane dai libri e vicine alle risate e alla spensieratezza.
Infine, sembrerò presuntuosa ma voglio ringraziare me stessa: per le volte in cui ho stretto i denti a fatica quando pensavo di mollare tutto, e per quelle in cui me la sono cavata da sola, senza l'aiuto di nessuno. Grazie a chi mi ha remato contro, per aver fatto aumentare la fiducia in me stessa.
Grazie a tutti voi, perché l'unione fa la forza e probabilmente, senza il vostro sostegno non avrei tagliato questo importante, agognato e sudato traguardo.

Bibliografia

[1] Helios Voting Website. *https://heliosvoting.org*

[2] CoQ Website. *https://coq.inria.fr*

[3] Finland Elections Website. *http://www.vaalit.fi*

[4] Switzerland Elections Website. *http://ge.ch/vote-electronique*

[5] EasyCrypt Website. *https://www.easycrypt.info*

[AB+05] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, L. Vigneron. *The Avispa Tool for the Automated Validation of Internet Security Protocols and Applications*, 2005.

[AC16] A. Azevedo de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, B. C. Pierce, V. Sjöberg, B. Yorgey. *Software Foundations*, 2016.

[Adi08] B. Adida. *Helios: Web-based Open-Audit Voting*, 2008.

[AD+09] B. Adida, O. De Marneffe, O. Pereira, J. J. Quisquater. *Electing a University President using Open-Audit Voting: Analysis of real-world use of Helios*, 2009.

[AR00] M. Abadi, P. Rogaway. *Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)*, 2000.

[AS16] T. M. N. U. Akhund, M. M. Sarker. *The Roadmap to the Electronic Voting System Development: A Literature Review*, 2016.

[BC04] Y. Bertot, P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.*, 2004.

[BC+11] D. Bernhard, V. Cortier, O. Pereira, B. Smyth, B. Warinschi. *Adapting Helios for Provable Ballot Privacy*, 2011.

[BC+17] G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, S. Zanella Béguelin.*EasyCrypt Reference Manual*, 2017.

[BC13] B. Blanchet, D. Cadé. *Proved Generation of Implementations from Computationally Secure Protocol Specifications*, 2013.

[BC16] B. Blanchet, D. Cadé. *CryptoVerif Computationally Sound, Automatic Cryptographic Protocol Verifier User Manual*, 2016.

[BD+16] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, P. Y. Strub. *EasyCrypt: A Tutorial*, 2016.

[BE+07] M. D. Byrne, S. P. Everett, K. K. Greene. *Usability of Voting Systems: Baseline Data for Paper, Punch Cards, and Lever Machines*, 2007.

[BG+12] G. Barthe, B. Grégoire, S. Zanella Béguelin. *Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs*, 2012.

[BP+12] D. Bernhard, O. Pereira, B. Warinschi. *How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios*, 2012.

[BW14] D. Bernhard, B. Warinschi. *Cryptography Voting - A Gentle Introduction*, 2014.

[Cap08] L. Caporusso. *Elezioni come Procedura: Forma, Osservazione e Automatizzazione del Voto*, 2008.

[CC03] A. Cerioni, F. Cenacchi. *Aspetti di Sicurezza nelle Elezioni Elettroniche*, 2003.

[Che14] V. Cheval. *APTE: An Algorithm for Proving Trace Equivalence*, 2014.

[Cor00] J. S. Coron. *On the Exact Security of Full Domain Hash*, 2000.

[Cor15] V. Cortier. *Formal Verification of E-Voting: Solutions and Challenges*, 2015.

[Cre08] C. J. F. Cremers. *The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols*, 2008.

[DF+14] Z. Durumeric, T. Finkenauer, J. A. Halderman, J. Hursti, J. Kitcat, M. MacAlpine, D. Springall. *Security Analysis of the Estonian Internet Voting System*, 2014.

[DH76] W. Diffie, M. Hellman. *New Directions in Cryptography*, 1976.

[DT96] J. M. Davis, S. Thomas. *Direct Recording Electronic Voting Machine and Voting Process*, 1996.

[DY83] D. Dolev, A. C. Yao. *On the Security of Public Key Protocols*, 1983.

[Edi69] T. A. Edison. *Improvement in Electrographic Vote-Recorder*, 1869.

[ElG85] T. ElGamal. *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, 1985.

[FO+92] A. Fujioka, K. Ohta, T. Okamoto. *A Pratical Secret Voting Scheme for Large Scale Elections*, 1992.

[FP01] P. A. Fouque, D. Pointcheval. *Threshold Cryptosystems Secure Against Chosen-Ciphertext Attacks.*, 2001.

[Gjo11] K. Gjøsteen. *The Norwegian Internet Voting Protocol.*, 2011.

[GM84] S. Goldwasser, S. Micali. *Probabilistic Encryption.*, 1984.

[GO13] O. Goldreich, Y. Oren. *Definitions And Properties Of Zero-Knowledge Proof Systems.*, 2013.

[Gol01] O. Goldreich. *Foundations of Cryptography, Basic Tools.*, 2001.

[Ker83] A. Kerckhoffs. *La Cryptographie Militaire*, 1883.

[KK05] D. C. Kimball, M. Kropf. *Ballot Design and Unrecorded Votes on Paper-Based Ballots*, 2005.

[KL07] J. Katz, Y. Lindell. *Introduction to Modern Cryptography*, 2007.

[Kri14] R. Krimmer. *Identifying Building Blocks of Internet Voting: Preliminary Findings*, 2014.

[LY11] B. Libert, M. Yung. *Adaptively Secure Non-Interactive Threshold Cryptosystems*, 2011.

[Ra80] M. O. Rabin. *Probabilistic Algorithm for Testing Primality*, 1980.

[Roy88] R. G. Saltman. *Accuracy, Integrity and Security in Computerized Vote-Talling*, 1988.

[Rub01] A. Rubin. *Security Considerations for Remote Electronic Voting over the Internet*, 2001.

[SW01] D. Sangiorgi, D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*, 2001.

[Sch96] B. Schneier. *Applied Cryptography. Protocols, Algorithms and Source Code in C*, 1996.

[Sho06] V. Shoup. *Sequences of Games: A Tool for Taming Complexity in Security Proofs*, 2006.

[Ven12] D. Venturi. *Crittografia nel Paese delle Meraviglie*, 2012.

[Zan10] S. Zanella Béguelin. *Formal Certification of Game-Based Cryptographic Proofs*, 2010.