Università degli Studi di Salerno

Dipartimento di Ingegneria dell'Informazione ed
Elettrica e Matematica Applicata

Progetto di Algoritmi e Protocolli per la Sicurezza

Gruppo: ABCD
Amendola Miriam (Resp. WP4)
Battipaglia Valerio (Resp. WP2)
Caso Antonio (Resp. WP3)
Dell'Orto Maria Giuseppe (Resp. WP1)

Docente: Prof. Ivan Visconti

2022/2023

Indice

1 Threat Model

1.1 Generazione di stringhe casuali

La funzionalità da realizzare è quella della generazione di stringhe binarie casuali, che devono essere utilizzate per l'esecuzione dei giochi della sala Bingo virtuale.

Per questa funzionalità possiamo avere un numero variabile di giocatori $P_1, \ldots, P_n$, che chiameremo *players*. L'attore che invece si occupa di generare continuamente numeri casuali è il server della sala Bingo, che indichiamo con S. La generazione dovrà avvenire sulla base di informazioni legate al contesto della partita, che indicheremo genericamente con il termine *seed*.

Possiamo assumere che, per questa funzionalità, sia il server che i player potrebbero comportarsi, in momenti separati, in maniera disonesta. I player avrebbero interesse a vincere conoscendo in anticipo la sequenza di numeri casuali, mentre il server avrebbe interesse a inquinare la sequenza generata o per consentire la vincita a giocatori lui vicini, oppure per trarre un proprio guadagno personale impedendo la vincita dei giocatori onesti.

Definizioni Siano $P_1, \ldots, P_n$ dei giocatori che intendono giocare ad una partita nella lobby della sala Bingo e sia S il server. Al tempo T_0, $P_1, \ldots, P_n$ (ed eventualmente il banco) fanno una scommessa. Al tempo T_1, i *player* inviano ad S dei *seed* $s_1, \ldots, s_n$ e, contemporaneamente, S produce esso stesso un *seed* s_0. Il server inizializza un generatore G, il quale, dall'istante T_2, può essere impiegato per produrre in uscita una sequenza $r = r_0, r_1, \ldots$ di numeri pseudocasuali, nell'ambito della puntata.

Threat models Per questa funzionalità, possono essere individuati i seguenti tipi di avversari:

- **giocatore passivo**: si limita ad osservare la sequenza di numeri generata dal server ed i seed inviati dagli altri giocatori. Un avversario di questo tipo, che potenzialmente può essere anche composto da un insieme di player, oltre a registrare i numeri generati dal server osserva i seed che esso stesso invia durante la partita. Questo macro-avversario si potrebbe trovare in essere quando uno o più player si mettono d'accordo durante la partita per cercare di prevedere il prossimo numero casuale.
- **forgiatore**: al posto di inviare il seed come richiesto dal protocollo, invia delle informazioni a sua scelta. Anche questo avversario può essere composto da più avversari che si mettono d'accordo a scapito del server, che invece agirebbe in maniera onesta. Anche il server, in alternativa, potrebbe agire da forgiatore, impiegando un seed non aderente al protocollo, agendo, in tal caso, a danno dei giocatori.

- **bluff**: sia il server che i player possono ripudiare i seed se l'esito della partita non è a loro favore, nella speranza che la partita venga annullata.
- **collaborazione tra server e giocatori**: in questo scenario, vi è una collaborazione tra server e $n-1$ giocatori disonesti; ciò sarebbe mirato a estorcere indebitamente denaro alla parte onesta, a beneficio e dei giocatori e del server.
- **server disonesto**: il server non segue il protocollo per la generazione dei numeri pseudocasuali e produce esso stesso una sequenza numerica che consenta di far vincere il banco evitando l'utilizzo del generatore.

Correctness Se i player $P_1, \ldots, P_n$ ed S producono contributi casuali $s_0, \ldots, s_n$, allora S è in grado di generare delle stringhe pseudocasuali che dipendono da tutti i contributi.

Proprietà

- *non-predicibilità*: consiste nell'impossibilità di prevedere una prossima stringa randomica prodotta sulla base delle precedenti stringhe emesse;
- *non-falsificabilità*: la presenza di almeno un seed prodotto in maniera onesta, impedisce gli avversari di forgiare la sequenza randomica;
- *verificabilità*: tutte le parti coinvolte devono poter verificare che la sequenza di numeri casuali sia stata ottenuta tramite una completa aderenza al protocollo descritto.

1.2 Green Pass 2.0

La funzionalità da realizzare consiste nel consentire ad un possessore di Green Pass 2.0 di trasmettere alla sala Bingo solo alcune delle informazioni presenti all'interno del documento, ad esempio la data dell'ultima vaccinazione. Lo stesso Green Pass 2.0, nel contesto della sala Bingo, dovrebbe poter essere usato come mezzo di autenticazione per accedere all'area riservata del giocatore.

Per questa funzionalità è necessario coinvolgere tre attori: il Ministero della Salute (abbr. MS), il giocatore P possessore del GP 2.0 e un attore V, che indica un generico *verifier*, che ha bisogno di informazioni contenute all'interno del GP. Poiché la trattazione di questa funzionalità non è limitata alla sala Bingo ma dev'essere proposta ad una competizione, l'attore V rappresenta, in generale, qualsiasi persona o entità che vuole verificare informazioni riguardanti il soggetto, come l'avvenuta vaccinazione oppure la data dell'ultimo tampone. Nel caso dell'identificazione, si considererà il server S come controparte di P in quanto tale funzionalità è legata alla sala Bingo.

Il Ministero della Salute si assume essere una parte fidata che emette il GP ed essere coinvolta solo in fase di rilascio. Si può invece dubitare dell'onestà dei giocatori e dei *verifier*. Un giocatore, infatti, vorrebbe tentare di contraffare

il GP perché potrebbe non essere vaccinato e quindi desiderare di modificare i propri dati sanitari oppure di usare il GP di un'altra persona per accedere alla sala Bingo.

I *verifier*, invece, potrebbero essere disonesti perché sulla base del GP ricevuto cercano di estrarre ulteriori informazioni personali sui giocatori senza la loro autorizzazione.

Definizioni Supponiamo che al tempo $T0$, MS abbia rilasciato a P il proprio Green Pass. Al tempo $T1$, P vuole accedere per la prima volta alla sala Bingo, per cui contatta S e si registra, inviandogli il GP.

Per consentire la registrazione (e successivamente l'accesso), è necessario che S verifichi che l'utente rispetti i requisiti stabiliti dalla Sala Bingo. I requisiti devono coinvolgere esclusivamente i dati $m_j, \ldots, m_k$ imposti dal Garante della Privacy. Si suppone che questi dati siano pubblicamente disponibili e l'applicativo usato da P per gestire il Green Pass sia sempre aggiornato con le nuove policy.

Se la procedura di verifica è andata a buon fine, al tempo $T2$, S registra P nel suo database, consentendo l'accesso alla sala Bingo.

Threat models

- **Eavesdropper**: resta in ascolto sul canale durante lo svolgimento del protocollo di identificazione e cerca di ottenere informazioni personali dei giocatori, o di carpire informazioni utili successivamente per indurre tale conoscenza;
- **Man-in-the-middle**: avversario attivo con la volontà di impedire ad una persona l'accesso al servizio modificando i dati scambiati dalle due parti;
- **Forgiatore**: un *player* potrebbe forgiare un Green Pass per accedere al servizio senza avere l'autorizzazione del Ministero della Salute. Ad esempio, qualcuno di non vaccinato avrebbe interesse a fingersi tale in modo da avere comunque accesso alle sale di gioco;
- **Ladro di identità**: avversario attivo in grado di utilizzare il Green Pass di un'altra persona per accedere al servizio;
- **Verifier ficcanaso**: il server stesso potrebbe essere intenzionato ad avere accesso a maggiori quantità di informazioni di quante consentitogli dalle policy di gioco, questo sia a fini di compravendita di informazioni personali degli utenti sia al fine di impersonarsi come altro utente in altri contesti;
- **Server disonesto** il server, con l'interesse a non svolgere le verifiche sulla natura e contenuto del Green Pass i maniera onesta; questo al fine di impedire a un utente l'accesso alle sale gioco.

Correctness Se P è onesto e dispone di un Green Pass rilasciato da MS, allora qualunque siano i dati $m_j, \ldots, m_k$ imposti dalla policy, la verifica del Green Pass effettuata da un verifier V onesto dovrà dare esito positivo e dovrà consentire l'accesso al profilo utente, senza che siano rivelate informazioni diverse da quelle proposte dalla policy.

Proprietà

1. *Confidenzialità*:
   (a) Le informazioni personali scambiate durante questa comunicazione dovranno essere non intercettabili e comprensibili da terze parti disoneste.
   (b) Ulteriori informazioni rispetto a quelle dichiarate dal server come necessarie per l'accesso al servizio non possono essere estratte dal Green Pass.
2. *Integrità*:
   (a) I dati contenuti all'interno del Green Pass non sono forgiabili nè dal player stesso nè da parti terze durante il loro transito sul canale di comunicazione.
3. *Autenticità*:
   (a) I dati contenuti all'interno del Green Pass devono essere verificabili come emessi dall'autorità competente, vale a dire il Ministero della Salute.
   (b) Chi verifica il Green Pass è in grado di accertarsi che il *player* sia il vero possessore del documento.

2 Soluzione

Premessa Tutte le comunicazioni tra i vari attori, sia per la generazione di numeri casuali che per la verifica del Green Pass si appoggiano su TLS. Si presuppone quindi che il server sia dotato di un certificato digitale rilasciato da un'autorità di certificazione *trusted*, il quale certifica che la chiave pubblica del Ministero della Salute è Pk_{MS} (chiaramente associata ad una chiave segreta Sk_{MS}).

2.1 Green Pass 2.0

Formato Sia Π = (`Gen, Sgn, Ver`) uno schema di firma digitale e sia Π' = (`Gen', Com', Ver'`) uno schema di commitment. Siano $m_1, \ldots, m_n$ le informazioni che devono essere presenti sul Green Pass.

Formalmente, il Green Pass 2.0 è un dato strutturato nella maniera seguente:

$$GP = (m := (id, c_1, c_2, \ldots, c_n, d, Pk_p), \sigma)$$

Dove:

- id è un identificativo alfanumerico associato al possessore del GP;
- $c_i = \texttt{Com'}(m_i, r_i)$, $\forall i = 1, \ldots, n$ sono i commitment delle informazioni che sono previste sul GP;
- d è la data di scadenza del GP;
- Pk_P è la chiave pubblica di P;
- $\sigma = \texttt{Sgn}(Sk_{MS}, m)$, dove Sk_{MS} è la chiave segreta di MS.

Procedura di rilascio Il Ministero della Salute dovrà:

1. generare una coppia di chiavi asimmetriche (Pk_P, Sk_P) legate a P;
2. produrre i commitment dei dati personali di P;
3. firmare digitalmente il GP usando Sk_{MS};
4. inviare all'utente il Green Pass, Sk_P, i dati sanitari e le randomness usate per produrre i commitment.

Interazione Server - Player
In primo luogo, V deve accertarsi che il Green Pass inviatogli da P sia effettivamente di sua proprietà e non sia stato inviato da qualcuno che lo ha ottenuto in maniera illecita.

Per fare ciò, quando P contatta V per richiedere l'accesso al servizio, V invia a P una *challenge* c, ossia una stringa casuale.

Il *player* risponde inviando il Green Pass seguito da $(m_j, r_j), \ldots, (m_k, r_k)$ e una firma digitale così calcolata:

$$\sigma_{GP} = Sgn(SK_P, GP||(m_j, r_j)|| \ldots ||(m_k, r_k)||c)$$

A questo punto, il server inizia il processo di verifica.

Verifica del Green Pass Il verifier esegue l'algoritmo `CheckGP(args)` composto dalle seguenti procedure:

1. **verifica l'identità** di P assicurandosi che questo abbia superato la *challenge* tramite verifica della firma digitale σ_{GP} avvalendosi della chiave pubblica dell'utente Pk_P;
2. **verifica l'autenticità** del GP tramite verifica della firma digitale σ_{MS} avvalendosi della chiave pubblica del Ministero della Salute Pk_{MS};

3. **verifica la validità** del GP controllandone la data di scadenza;
4. **verifica l'autenticità** dei dati di P aprendo il relativo commitment presente su GP impiegando i dati in chiaro e le randomness inviategli da P;
5. **verifica i requisiti** di P, con funzioni che dipendono dalle sue policy di accesso, ad esempio, verifica che l'utente sia maggiorenne o che abbia fatto la terza dose.

```
CheckGP(args):
    b = Ver(σ_GP, Pk_P, GP||(m_j, r_j)|| ... ||(m_k, r_k)||c)
    if b == 0:
        return

    b = Ver(σ_MS, Pk_MS, GP) and check(d)
    if b == 0:
        return

    for i = j to k:
        b = Ver(c_i, m_i, r_i) and check(m_i)
        if b == 0:
            return
```

Autenticazione Affinché il server possa riconoscere il GP come strumento di accesso al profilo è necessario che venga memorizzata, in fase di registrazione, un'informazione legata a quest'ultimo, che non cambi ad ogni emissione del documento; a tale scopo viene impiegato il campo *id* del certificato.

Quando l'utente si registra, se la procedura di verifica `CheckGP` è andata a buon fine, il server crea una tupla all'interno del proprio database legata al profilo dell'utente e inserisce l'hash del campo *id* usando `SHA256`.

In questo modo, ogni volta che l'utente vuole accedere al servizio, il server cercherà l'hash dell'*id* all'interno del database. Se trova un riscontro, farà accedere l'utente al servizio, in caso contrario negherà l'accesso.

2.2 Generazione di numeri casuali

Preparazione dei seed I giocatori $P_1, \ldots, P_n$ entrano nella lobby di gioco e avviano una partita. In primo luogo, effettuano le scommesse per la partita e solo a questo punto, essi producono i contributi $s_1, \ldots, s_n$ composti da stringhe casuali di taglia n.

Ciascun giocatore P_i produce una firma digitale σ_i di $s_i||t_i$, dove s_i è il contributo del giocatore e t_i è un timestamp. P_i manda ad S la tripla $(s_i, t_i, \sigma_i = Sgn(Sk_P, s_i||t_i))$, dove Sk_P è la *secret key* di P_i accoppiata alla chiave pubblica Pk_P presente sul Green Pass 2.0.

Il server esegue la stessa procedura dei giocatori ottenendo la tripla $(s_0, t_0, \sigma_0 = Sgn(Sk_S, s_0||t_0))$, dove Sk_S è la *secret key* del server legata al suo certificato TLS.

Il server calcola il *seed* s del generatore attraverso un Merkle Tree in cui le foglie sono i contributi $s_0, \ldots, s_n$.

$$s = \mathcal{MT}(s_0, \ldots, s_n)$$

Generatore Successivamente, il server ottiene una stringa binaria di numeri pseudocasuali r_0 tramite utilizzo della funzione SHA256.

$$r_0 = H(s)$$

Questa stringa potrà essere impiegata come stringa casuale per partecipare alla sala Bingo. Successivamente, quando verranno esauriti i bit di r_0, il server procederà a calcolare una ulteriore stringa random nel modo seguente:

$$r_1 = H(s||r_0)$$

Questo processo si ripeterà ulteriore finché vi sarà necessità di ulteriori stringhe, tramite la ricorrenza:

$$r_n = H(s||r_{n-1})$$

Quando si saranno esaurite le scommesse fatte prima dell'invio dei contributi, sarà necessario ripetere il calcolo del seed utilizzando nuovi contributi inviati dai *player*.

Verifica Al termine di ogni mano della partita, il server invia a tutti i giocatori i *seed* $s_0, \ldots, s_n$.

Ciascun giocatore potrà verificare che la sequenza di numeri casuali sia stata ottenuta onestamente ripetendo il processo di generazione. Nel caso la verifica non andasse a buon fine, un giocatore potrebbe contattare le autorità per denunciare l'accaduto.

Se, in fase di verifica, un *player* dovesse contestare l'utilizzo del suo contributo, ossia dichiarare che il contributo inviatogli dal server non è quello che lui ha mandato, la firma digitale prodotta contestualmente all'invio del *seed* farà fede e potrà essere usata dal server come prova della sua onestà.

2.3 Algoritmi e protocolli utilizzati

2.3.1 SHA256

`SHA256` è una funzione *one-way* collision-resistant. `SHA256` prende in input una stringa di dimensione arbitraria produce in output un hash di taglia 256 bit.

2.3.2 Merkle Tree

Un Merkle Tree $\mathcal{MT}$ è un albero binario in cui ogni nodo interno è l'hash crittografico dei suoi due figli. L'uso di questa struttura garantisce che se anche un singolo bit di dati viene modificato, l'hash radice sarà completamente diverso.

2.3.3 Schema di commitment

Uno schema di commitment consente di "impegnare" dei dati in modo che il destinatario non possa conoscere il valore impegnato fino a quando il mittente non decide di rivelarlo. Uno schema di commitment è una tripla di algoritmi randomizzati ed efficienti (`Gen, Com, Ver`) che soddisfa le seguenti proprietà:

- **hiding**, nessuna informazione sui dati impegnati può essere ricavata a partire dal commit.
- **binding**, è un problema difficile trovare due dati che verifichino lo stesso commit.

Come schema di commitment si può usare:

- `Gen`(1^n): `return` $r \leftarrow \{0,1\}^n$
- `Com(m,r)` : `return` $H(r||m)$
- `Ver(c,m,r)` : `return` $H(r||m) == c$

2.3.4 Firma digitale di Schnorr

La firma digitale è uno schema crittografico che consente di dimostrare **autenticità** e **integrità** dei dati forniti dal mittente, impiegando una coppia di chiavi privata e pubblica.

Come schema di firma digitale si può usare lo schema di firma di *Schnorr*.[1] Lo schema Π = (`Gen, Sgn, Ver`) è così composto: L'algoritmo `Gen` genera una coppia di chiavi pubblica e privata $((G, q, g, y = g^x), x)$.

L'algoritmo `Sgn`(x, m):

`Sgn`(x, m):
$r \leftarrow \{0,1\}^n$
$a = g^r$
$c = H(y||a||m)$
$z = r + cx$
`return` (z, a)

L'algoritmo `Ver`$((G, q, g, y), m, (z, a))$:

[1] **Nota:** L'implementazione della firma è stata realizzata impiegando ECDSA come algoritmo di firma in quanto proposto come standard da OpenSSL. In un contesto reale sarebbe auspicabile utilizzare Schnorr in quanto più veloce e migliore nell'hiding, essendo basato su una (Honest Verifier) Zero-Knowledge Proof.

`Ver`$((G, q, g, y), m, (z, a))$:
    $c = H(y||a||m)$
    `return` $g^z == ay^c$

3 Analisi

3.1 Generazione di stringhe casuali

3.1.1 Non-falsificabilità

A partire dal threat model relativo ad una collaborazione tra il server stesso ed $n - 1$ giocatori, dimostriamo la solidità della proprietà.

Si supponga che il server disponga di un canale di comunicazione con i giocatori disonesti al fine di stabilire un eventuale stringa utile ai propri fini disonesti (potendo inoltre modificare i seed randomici emessi dai giocatori disonesti a proprio piacimento).

Nonostante ciò, sarà sempre presente almeno un contributo dovuto a un attore onesto: questo contributo onesto verrà utilizzato all'interno di una delle foglie del Merkle Tree e dunque verrà sottoposto ad hashing (utilizzando SHA256), e il risultato verrà propagato verso il livello superiore.

Allo stesso modo, anche le componenti disoneste emesse dagli altri giocatori subiranno stesso processo all'interno del Merkle Tree. Da ciò ne deriva che il contributo onesto confluirà necessariamente all'intero della lavorazione di cui saranno oggetti i contributi disonesti; questo vuol dire che per riuscire a calcolare una stringa utile ai propri intenti disonesti, il server dovrebbe essere in grado di:

- Conoscere perfettamente la struttura interna della funzione di hash resistente alle collisioni SHA256, in modo da poter dedurre l'input corretto per ottenere un output utile ai propri fini (e dunque, sapere come modificare i seed ricevuti per ottenere il giusto input).
- Conoscere perfettamente il modo in cui la stringa ottenuta dall'attore onesti si propaghi all'interno della struttura dell'albero.

Conseguentemente, anche essendo in grado di modificare i *seed* ricevuti dei giocatori disonesti (incluso il proprio), non sarà possibile indurre una prolifica sequenza di bit tale per cui il risultato del Merkle Tree (e dunque la radice dello stesso) si dimostri utile ai fini del party disonesto.

La robustezza della proprietà contestualmente a questo threat model implica, di rimando, la robustezza anche nei confronti di un avversario passivo, del resto già implicitamente stabilita avendo introdotto il protocollo TLS durante le comunicazioni tra giocatori e server. Similmente, questa proprietà permane anche nel contesto di un threat model con

- Server disonesto, giocatori onesti: il server potrebbe modificare le stringhe dei giocatori a loro insaputa, ma comunque non avrebbe modo di influenzare (salvo essere dotato di capacità computazionali impensabilmente elevate) la root del Merkle Tree.
- Server onesto, giocatori disonesti: i giocatori, tutti d'accordo, potrebbero inviare i loro $n-1$ contributi in modo da ottenere una certa generazioni di numeri durante la partita. Questo però, come discusso precedentemente, non potrà verificarsi dal momento che, insieme ai contributi dei giocatori, all'interno del Merkle Tree verrà utilizzato anche l'elemento fornito dal server, ripristinando la correttezza e liceità del protocollo.

3.1.2 Non-predicibilità

Si consideri nuovamente il threat model **collaborazione tra server e giocatori**; si dimostra di seguito la resilienza della proprietà in simile contesto.

Al fine di poter predire la successiva stringa, sarebbe necessario essere a conoscenza, come menzionato nella precedente dimostrazione, del funzionamento interno della funzione di SHA256 e, ancor di più, sapere il modo in cui ciascun output convergerebbe in input per il livello superiore. Assumendo ciò computazionalmente troppo complesso, e ancor di più nel tempo utile all'inizio di una partita (per cui non sarebbe pensabile aspettare più di qualche minuto, pena la disconnessione dei giocatori più impazienti), risulta che plasmare le $n-1$ stringhe $s_0, s_1, ..., s_{n-1}$, associate agli $n-1$ attori disonesti, al fine di influenzare la disonesta generazione della root

$$y = H_{2n-1}(H_{2n-2}(...H_0(s_0))||H_{2n-3}(...H_n(s_n)))$$

sia non possibile, garantendo dunque la resistenza della proprietà a un simile avversario.

Considerando ora un avversario composto esclusivamente o da un server disonesto, o da l'interezza dei giocatori disonesti, non è possibile in ogni caso predire la futura y emessa (e dunque predire la generazione futura dei numeri utili al gioco, in quanto il generatore G non lo si suppone segreto).

3.1.3 Verificabilità

Questa proprietà, come discusso, mira a rendere inequivoca la generazione della sequenza di numeri di gioco, ed accertare che essa sia avvenuta in maniera strettamente aderente al protocollo. Dimostriamo la robustezza di questa proprietà.

Successivamente al piazzamento delle quote e alla fase di scommessa, ad ogni giocatore, secondo protocollo, vengono inviati i contributi di tutti gli n attori della sala gioco. In questo modo, ogni singolo attore può verificare autonomamente la bontà della stringa generata dal generatore; prima calcolando da se la

root del Merkle Tree, e in secondo luogo inserendola all'interno del generatore e verificando che la sequenza di numeri prodotta sia effettivamente genuina.

Non è, pertanto, concepibile un comportamento disonesto in merito da parte del server o da un avversario "server+giocatori": se anche il server inviasse dei contributi falsificati al giocatore onesto, egli avrebbe immediata consapevolezza di ciò grazie alla firma $\sigma_i = Sgn(Sk_P, s_i||t_i)$, attraverso la quale (conoscendo la chiave pubblica degli altri attori) è persino in grado di verificare il momento esatto di emissione della stringa, scongiurando qualsiasi manipolazione.

Si consideri ora il threat model "**bluff**": in questo caso, un avversario che abbia perso la proprio scommessa, o che voglia invalidare la vittoria di un altro giocatore, potrebbe decidere di ripudiare il proprio contributo inneggiando ad una scorrettezza da parte del server al fine di ottenere l'annullamento della partita e dunque la restituzione della somma.

In questo caso, non si rende possibile ripudiare la propria stringa dal momento che l'utilizzo stesso della firma scongiura tale dinamica. Infatti, non sarebbe possibile produrre una stringa simile senza essere in possesso della chiave segreta, questo perché

$$\sigma_i = Sgn(Sk_P, s_i||t_i)$$

non potrà essere forgiata da altri che non posseggano la chiave Sk_P (fosse possibile ciò, si avrebbe un avversario in grado di vincere contro la *CDH assumption*), e sarà inoltre possibile verificare l'istante di emissione della stessa attraverso il time stamp.

Vi è, del resto, la possibilità che un giocatore si appelli al fantomatico riproponimento del seed da lui inviatogli contestualmente a una partita precedente: attraverso l'impiego del timestamp facente parte del messaggio firmato, un qualsiasi altro giocatore (anche fosse un singolo attore onesto) sarebbe in grado di decifrare il messaggio utilizzando la chiave pubblica, verificare il timestamp associato, e accertare la veridicità o meno delle parole del querelante.

3.1.4 Correttezza

Dati i contributi $s_0, s_1, ..., s_n$degli n attori onesti, è possibile definire la correttezza dello schema proposto nel seguente modo

$$\forall P_{i\in[0,n]}\ \exists!\ \sigma_i = Sgn(S_{k_i}, s_i||t_i)\ \wedge\ s = y = \mathcal{MT}(s_0, \ldots, s_n)$$

tale che

$$Prob[Ver(s_i, \sigma_i, P_{k_i}) = 1\ \wedge\ H(y) = r] = 1 - \varepsilon$$

Dovesse ogni giocatore produrre il proprio contributo, anche in caso di eventuali avversari previsti dai threat model precedentemente definiti, si sarebbe in grado di

- Giungere alla fine della partita seguendo il protocollo.

- Avere l'abort della partita in seguito a una rilevata anomalia di gioco.

In entrambi i casi, la correttezza dello schema viene rispettata.

3.1.5 Efficienza

Il sistema è progettato per essere il più efficiente possibile: il broadcasting dei contributi (dal server ai giocatori), per quanto oneroso, viene effettuato in quanto necessario alla verificabilità dell'onesto svolgimento della partita. La struttura stessa dei contributi è, inoltre, corredata del timestamp, il quale però si dimostra irrinunciabile per le motivazioni sopra descritte.

Allo stesso modo, l'utilizzo del TLS come strumento sottostante di sicurezza CCA è necessario al fine di garantire protezione contro avversari del tipo Man-in-The-Middle (ciò, include, ovviamente, anche protezione verso avversari passivi).

Una soluzione alternativa avrebbe potuto impiegare strumenti di BlockChain, questo avrebbe certamente reso inequivocabile la verificabilità e avrebbe impedito qualsiasi meccanismo e tentativo di forgiatura delle stringhe anche non ricorrendo a stringenti regole di protocollo come quelle descritte (e.g, il dover aspettare il termine della fase di scommessa prima di poter inviare i contributi al fine della verifica).

Per quanto ciò si sarebbe dimostrato di indubbia utilità, avrebbe certamente comportato due problematiche:

- L'impiego delle meccaniche di BlockChain (sia essa permissionless o permissioned) richiedono in ogni caso un dispendio energetico non trascurabile, che invece è sollevato dal bilancio per mezzo della soluzione proposta.
- Il protocollo risultante, per quanto maggiormente "collaudato", non sarebbe stato minormente complesso e articolato: da ciò ne sarebbero scaturite sicure latenze e costi di transazione dovuti all'utilizzo stesso della BlockChain.

Le computazioni richieste, contestualmente alla soluzione proposta, possono essere ridotte a computazioni di funzioni di hash crittografiche, le quali sono notoriamente piuttosto veloci. Oltretutto, sono presenti anche esponenziazioni modulari al fine di calcolare firme e cifrature, ma gli attori dovranno eseguire un numero costante di queste operazioni.

Infine, è bene notare che l'efficienza del meccanismo proposto è largamente basato sull'efficienza della funzione di hash SHA256 e sulla rapidità di calcolo della stessa.

3.1.6 Trasparenza

Contestualmente al protocollo di generazione delle stringhe pseudo-casuali, l'unico frangente in cui una mancanza di trasparenza potrebbe risultare effettivamente pericolosa è durante il calcolo del *seed* da fornire in ingresso al generatore

G e sul quale si baserebbe la sequenza di numeri e dunque lo svolgimento della partita.

A rimedio di ciò, è stata impiegata la soluzione precedentemente ampiamente discussa, la quale garantisce la **totale** trasparenza in merito alla generazione della stringa e al modo in cui ciascun *seed* partecipa in essa. Per questo motivo, il protocollo può vantare un'elevata trasparenza, anche se, come già discusso nel paragrafo precedente, questo avviene a scapito di un appesantimento delle operazioni di verifiche.

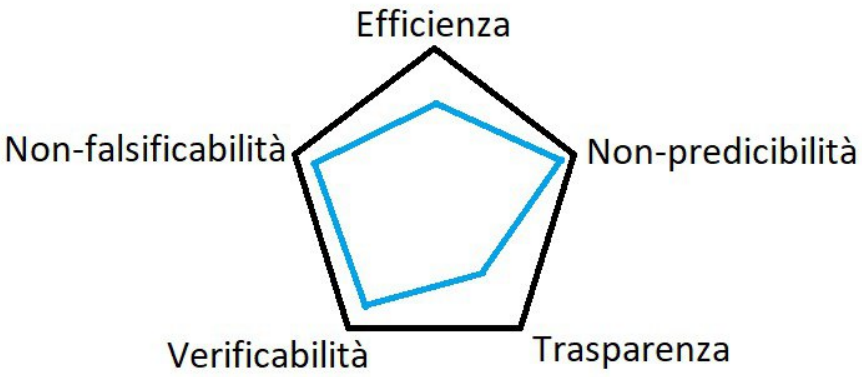

Figura 1: Valutazione proprietà

TOOLS TABLE		
Strumenti del WP2	**Segue da**	**Usato per**
Diffusione della randomicità internamente al Merkle Tree	Random Oracle Property di SHA-256	Non-falsificabilità, non-predicibilità
Non ripudio della firma digitale	Segretezza della chiave privata	Verificabilità
Non forgiabilità della firma digitale	Assunzione su Sign	Non-falsificabilità
Timestamp	n.d.	Verificabilità

3.2 Green Pass 2.0

3.2.1 Confidenzialità

Le proprietà di confidenzialità garantite dallo schema possono essere divise in due macro-sezioni (C.1 e C.2) come precedentemente affermato, e si discuterà ciascuna delle due separatamente.

Le informazioni scambiate durante la comunicazione non saranno in alcun modo intercettabili da una terza parte disonesta (sia essa ascrivibile a un avversario passivo o meno) a causa dell'utilizzo sottostante del protocollo TLS, notoriamente CCA sicuro.

In merito alla seconda particolarizzazione della proprietà di *confidenzialità*, essa viene assicurata dall'adozione dello schema di commitment fornito di seguito.

$$c_i = SHA256(r_i||m_i) \qquad \forall m_i \in GP$$

Al momento dell'invio delle informazioni, il verifier potrà, come espresso nel paragrafo *Soluzione*, accedere esclusivamente a quanto consentito dal player, non disponendo del mezzo per poter decodificare il contenuto del Green Pass; ciò è conseguenza dell'impossibilità da parte del server (qualora disonesto) di poter invertire l'hash del commitment per estrarre le informazioni personali.

Sulla base di quanto enunciato, risulta evidente la intrinseca robustezza della proprietà a qualsiasi *avversario* di tipo *passivo*. Inoltre, tale robustezza è garantita anche contestualmente a tutti gli altri threat model, in quanto:

- Per il threat model *Man-In-The-Middle*, la robustezza è garantita sia dall'utilizzo del TLS che dall'uso della challenge, motivo per cui un eventuale avversario non sarebbe in grado di impersonare un giocatore proprietario di un certo Green Pass.
- Per il threat model *Verifier ficcanaso*, la robustezza è garantita dal momento che il commitment non può essere violato in base a quanto precedentemente descritto.

3.2.2 Trasparenza

Il protocollo risulta, agli occhi di un attore onesto, fortemente trasparente dal momento che egli ha costante consapevolezza dell'ammontare di informazioni che gli viene richiesto (in base alla policy pubblicamente disponibile e consultabile), ed ha coscienza della sua interazione con il verifier in tutte le sue parti, dalla richiesta di connessione alla risoluzione della challenge (essa stessa avviene sotto il pieno controllo del player onesto ed utilizzando la challenge a lui visibile).

Unico elemento tacciabile di poca trasparenza potrebbe essere la generazione del Green Pass, in quanto essa avviene sotto il completo controllo del Ministero della Salute e solo in un secondo momento esso viene ceduto all'attore onesto fornendogli le istruzioni sulle modalità del recupero dei dati contenuti nel documento.

3.2.3 Efficienza

Il protocollo risulta purtroppo non eccessivamente efficiente, sebbene la configurazione proposta miri ad essere la più proficua possibile.

In particolare, gli strumenti crittografici utilizzati possono essere considerati particolarmente efficienti, in quanto sono stati utilizzati

- Funzioni di hash, notoriamente efficienti.
- Schema di commitment, anch'esso efficiente.
- Firma digitale di Schnorr, notoriamente efficiente.

Sebbene ciò sia vero, il protocollo soffre della necessità di un vistoso scambio di messaggi e tra il Ministero e il player (in quanto è necessario l'invio di vari elementi al fine di un corretto utilizzo del Green Pass) e tra il player ed un eventuale verifier: non solo viene prevista una challenge da dover svolgere, ma viene anche previsto il doppio invio del Green Pass al server (una prima volta come "presentazione", ed una seconda a fini di verifica).

Per questi motivi, il protocollo, sebbene impieghi strumenti conclamatamente efficienti, appare verboso seppur di necessaria complicatezza.

3.2.4 Correttezza

Dato il Green Pass:

$$GP = (m := (id, c_1, c_2, \ldots, c_n, d, Pk_p), \sigma)$$

E la firma su di esso definita:

$$\sigma_{GP} = Sgn(SK_P, GP||(m_j, r_j)|| \ldots ||(m_k, r_k)||c)$$

Si ha che:

$$\forall P \ \exists!\ \sigma = Sgn(S_{k_M}, GP) \ \wedge \ \exists! \sigma_G P = Sgn(S_{k_P}, GP||(m_j, r_j)|| \ldots ||(m_k, r_k)||c)$$

tale che

$$Prob[Ver(\sigma_G P, P_{k_P}, GP||(m_j, r_j)|| \ldots ||(m_k, r_k)||c) = 1 \wedge Ver(sigma, P_{k_M}, GP)$$

$$= 1 \ \wedge Ver(c_i, m_i, r_i) = 1] = 1 - \varepsilon$$

3.2.5 Integrità

La proprietà è facilmente dimostrabile in base a quanto descritto nel capitolo precedente.

I dati contenuti all'interno del Green Pass sono firmati dal Ministero: per poterli modificare, bisognerebbe essere in grado di forgiare la firma stessa contenuta nel documento, cosa che richiederebbe il possesso della chiave privata del Ministero stesso.

Non potendo, ragionevolmente, accedere ad essa, nessun avversario potrà essere in grado di creare un Green Pass contenente informazioni forgiate: ciò rende la proprietà robusta rispetto ad un avversario *Forgiatore*.

Similmente, un avversario *Man-In-The-Middle* non sarà in grado di modificare il Green Pass in maniera utile, dal momento che qualsiasi modifica dello stesso verrebbe immediatamente rilevata grazie alla non corrispondenza della firma presente sul documento.

3.2.6 Autenticità

Si discute ora il primo segmento della proprietà (A.1). Come descritto all'interno del secondo capitolo, il Green Pass è prodotto dal Ministero come segue:

$$GP = (m := (id, c_1, c_2, \ldots, c_n, d, Pk_p), \sigma)$$

Il Green Pass contiene, dunque, la firma apposta dal Ministero sul contenuto del Green Pass: la firma inequivocabilmente attribuisce la produzione del Green Pass all'autorità, e sarà certamente identificabile come appartenente ad essa.

Contestualmente al secondo segmento, il verifier è certamente in grado di accertare l'identità di chi invia il Green Pass grazie all'utilizzo della challenge, in base a quanto precedentemente detto.

Il commitment, inoltre, consente al verifier di accertare che i dati inviatigli siano effettivamente i dati contenuti nel Green Pass.

Sulla base di quanto enunciato, risulta evidente la intrinseca robustezza della proprietà a qualsiasi *avversario* di tipo *passivo*. Inoltre, tale robustezza è garantita anche contestualmente a tutti gli altri threat model, in quanto:

- Per il threat model *Forgiatore*, la robustezza è garantita dalla presenza della firma al documento, per cui per poter forgiare un Green Pass un avversario dovrebbe essere a conoscenza della chiave segreta del Ministero (cosa che si spera non possa accadere...).
- Per il threat model *Ladro di identità*, la robustezza è garantita per mezzo della challenge, nella misura in cui l'unica persona che potrebbe firmare il documento ed allegare la challenge è esclusivamente il possessore della chiave segreta del player onesto: per tale motivo è lecito assumere che ciò non

sarebbe possibile per un avversario disonesto. Se non ci fosse la challenge, il sistema sarebbe vulnerabile ad eventuali *replay attack*, dal momento che un avversario potrebbe memorizzare il contenuto dei messaggi inviati dal giocatore onesto e riproporli successivamente.

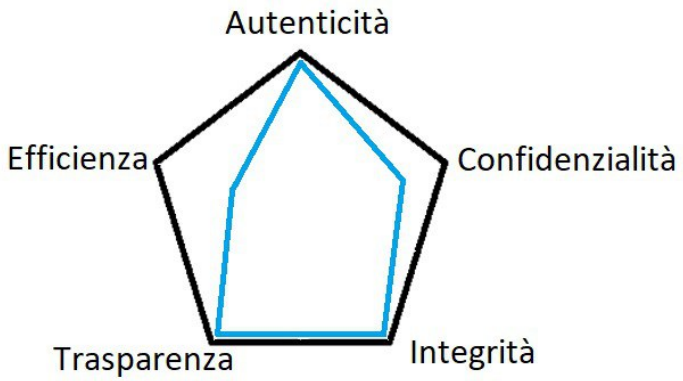

Figura 2: Valutazione proprietà

TOOLS TABLE		
Strumenti del WP2	**Segue da**	**Usato per**
TLS	Accordo di chiavi DH, etc.	C.1
Non forgiabilità della firma digitale	Assunzione su Sign	A.1, A.2
Proprietà di hiding dello schema di commitment	Random Oracle Property di SHA256	B.2
Proprietà di binding dello schema di commitment	Random Oracle Property di SHA256	I.1

4 Implementazione

4.1 Architettura

Il software presentato viene sviluppato proponendosi di simulare una Sala Bingo online che sia in grado di simulare gli attori descritti nella sezione 1 rispettando i protocolli presentati nella sezione 2.

Di seguito verranno descritte le implementazioni degli attori coinvolti.

Ministero della Salute Il Ministero della Salute è stato modellato come un server web che consente di generare e richiedere certificati. Il codice, scritto in Python 3.11 impiegando `flask` come framework per il server, è riportato all'interno della cartella `medical_authority`. Questa entità mantiene un database, implementato usando `sqlite3`, al cui interno sono contenute le informazioni personali e sanitarie degli utenti.[1]

Il web server consente di scaricare una policy di esempio e di scaricare degli archivi contenenti i Green Pass 2.0, indicando l'*id* dell'utente corrispondente al campo *id* del Green Pass descritto nella sezione 2.1.

Client I *player* possono partecipare ai giochi della Sala Bingo tramite un software che si occupa sia di gestire l'autenticazione tramite Green Pass 2.0 sia di inviare i contributi per la generazione di numeri casuali. Il software è stato sviluppato in Python 3.11 ed stato inserito nella cartella `client`. È dotato di interfaccia grafica che consente di simulare i protocolli di autenticazione, registrazione, generazione e verifica dei numeri casuali.

L'interfaccia viene costruita in maniera programmatica usando il framework `tkinter`. Le funzionalità che richiedono la comunicazione con il server, come la scommessa o l'autenticazione, sono eseguite da thread separati per prevenire il blocco l'interfaccia.

All'avvio, il client cerca di connettersi tramite *socket* al server. Se il server è raggiungibile, il client può autenticarsi o registrarsi. In ogni caso, verrà prima eseguita la procedura di autenticazione tramite Green Pass descritta nella sezione 2.1.

Se il controllo del Green Pass non è andato a buon fine oppure si è verificato un errore durante l'autenticazione, verrà visualizzato un messaggio di errore che informa l'utente del problema.

Se, invece, l'autenticazione è andata a buon fine, l'utente viene inserito dal server in una *lobby*. Il *client* produce, firma e invia al server una stringa binaria presa da `/dev/random` da usare come contributo.

[1] È stato inserito un metodo per inizializzare il database con dei dati arbitrari. Per poter aggiungere, visualizzare o modificare il database si consiglia di usare un client sqlite3, come *DB Browser for SQLITE*.

A questo punto, l'utente può piazzare una scommessa usando il bottone *Bet*. Per semplicità è stato assunto che ogni mano del gioco consista nell'indovinare un numero da 1 a 90. Al termine dello svolgimento di una mano del gioco, il server invierà anche i contributi usati per inizializzare il generatore e il client potrà verificare l'onestà del server replicando l'estrazione usando il bottone *Verify*.

Per funzionare correttamente, il software necessita dei seguenti file:

- *all_user_data.json*, contenente tutte le coppie (m_i, r_i)
- *green_pass.json*, contenente il Green Pass emesso dal Ministero della Salute
- *public_key.pem* e *private_key.pem*, corrispondenti alla coppia (Pk_P, Sk_P)
- *signature.bin*, contenente la firma digitale del Green Pass da parte del Ministero della Salute

Questi file possono essere caricati tramite il bottone *Upload Green Pass Data*, che copia il contenuto di una cartella all'interno della directory di lavoro del client.

Server Il server della Sala Bingo è stato modellato come un software che comunica tramite socket con i client dei *player*. Il software è stato sviluppato in Python 3.11 ed stato inserito nella cartella `server`. Il codice è diviso in più classi:

- La classe `Server` si occupa di gestire la logica di funzionamento dell'applicativo
- La classe `ServerSocket` si occupa di avviare il *socket server* e gestire le connessioni con i client.
- La classe `Lobby` modella una lobby di gioco, memorizzando i giocatori presenti e le loro scommesse

All'avvio dell'applicativo, viene avviato il server *socket*. Quando un client si connette, il server accetta la connessione e avvia un thread `handle_client` che gestisce la singola connessione con un client e il thread `handle_lobby` che implementa la logica di gioco.

Il thread `handle_lobby` controlla periodicamente che tutti i giocatori abbiano piazzato una scommessa. Quando una lobby è pronta, il thread inizializza il generatore di numeri casuali e invia in *broadcast* il risultato ai client.

Il thread `handle_client` è stato progettato come una macchina a stati.

- Nello stato `WAITING`, il server è in attesa di una richiesta di autenticazione (o registrazione) del client. Quando il server riceve la richiesta, genera e invia una *challenge* ed il thread passa nello stato `WAITING_FILES`.
- Nello stato `WAITING_FILES`, il server riceve i file necessari per l'autenticazione, ossia *user_data.json*, *green_pass.json*, *signature.bin*, *public_key.pem*

e *challenge_signature.bin*; quest'ultimo contenente la *challenge* firmata assieme ai dati di contesto. Terminata la ricezione dei file, il server controlla i dati tramite la funzione `CheckGP`, il cui algoritmo è stato descritto nella sezione 2.1.

Se l'autenticazione è andata a buon fine, il server passa nello stato `LOGGED`, altrimenti ritorna nello stato `WAITING` inviando la causa dell'errore.

- Nello stato `LOGGED`, il server attende la ricezione del contributo per la generazione di numeri casuali da parte del client e lo memorizza all'interno di una directory legata alla lobby. Successivamente attende la ricezione della scommessa piazzata dal *player*, la memorizza all'interno della lobby e passa nello stato `BETTING`.
- Nello stato `BETTING`, il server attende il termine della mano di gioco. Quando la lobby ha decretato un vincitore, informa il client dell'esito della partita.

4.2 Libreria

Le funzionalità di firma, commitment e generazione di numeri casuali necessarie per l'implementazione del sistema sono state inserite all'interno del package `lib`, condiviso da tutti i componenti del sistema.

Le funzionalità di base (generazione di chiavi, firma, SHA256, etc.) sono fornite dalla libreria **OpenSSL**. Nel package `lib` sono state aggiunte delle funzioni wrapper che richiamano gli eseguibili OpenSSL e ne processano il risultato.

Il modulo `sign.py` contiene i wrapper per lo schema di firma **ECDSA** e per l'implementazione della *challenge* descritta nel paragrafo *Interazione Server-Player* della sezione 2.1.

Il modulo `commit.py` contiene i wrapper per lo schema di commitment e per il calcolo di **SHA256**.

Il modulo `random.py` contiene gli algoritmi per la generazione di stringhe casuali. In questo modulo è incluso:

- L'implementazione dell'algoritmo per il calcolo del *Merkle Tree* di stringhe binarie.
- L'implementazione dell'algoritmo per il calcolo del *seed* del generatore `get_initial_seed`.
- L'implementazione del `PRG` in grado di produrre stringhe binarie (`random_string_generator`) e una sua estensione in grado di produrre interi (`random_number_generator`).