UNIVERSITA` DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA

DELL'INFORMAZIONE ED ELETTRICA E

MATEMATICA APPLICATA

Algoritmi e Protocolli per la Sicurezza

Team ABGN-secure

Intonti Giovanni 0622701994 – WP3

Lanzara Nicola 0622702118 – WP4

Squitieri Beniamino 0622702021 – WP1

Vitale Antonio 0622701988 – WP2

WP1

Introduzione

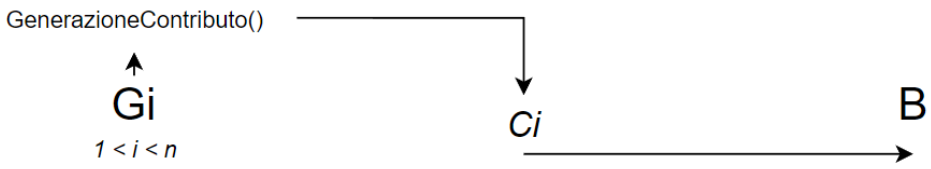
A causa della più recente pandemia dovuta al virus Covid, è stato necessario identificare la popolazione e monitorare il loro stato di salute attraverso la stesura di un documento: il Green Pass. Esso è risultato necessario per partecipare ad eventi sociali, a causa della pericolosità di contagio del virus. Ultimi aggiornamenti da parte del Data Protection Authority hanno portato alla necessità della stesura di un nuovo documento digitale rilasciato dall'autorità giudiziaria per adempiere agli stessi obiettivi del Green Pass originale, ma per eventi sociali on-line. Si vuole quindi definire un formato per questo nuovo Green Pass 2.0 tenendo conto della necessità, espressa dal garante, di poter e dover inviare soltanto le informazioni necessarie sulla base del contesto e non tutte le generalità del soggetto per accedere ad un servizio. Uno degli eventi sociali di maggiore interesse è quello delle sale virtuali, compresa quella di Mister Joker, che richiede che tale documento sia utilizzato per accedere al proprio conto gioco e anche per poter partecipare alle partite. Una funzionalità richiesta per intrattenere i giocatori è quella di generare stringhe casuali attraverso il contributo attivo di tutti i giocatori, compreso il banco o server.

L'obiettivo del project work è la stesura di una struttura per il Green Pass 2.0 e la definizione formale di come effettuare l'accesso alla piattaforma virtuale di Bingo da parte dei giocatori stabilendo il meccanismo di generazione delle stringhe casuali.

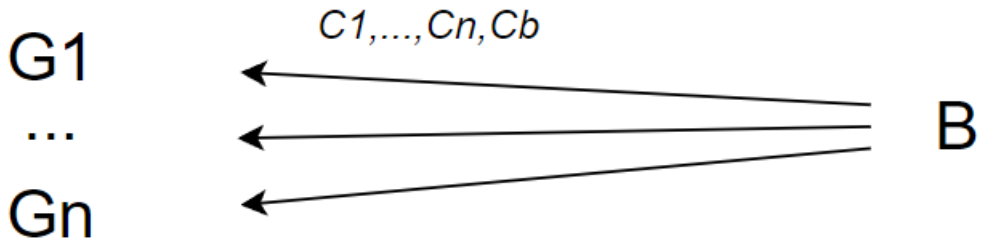
Completezza:

Prima funzionalità:

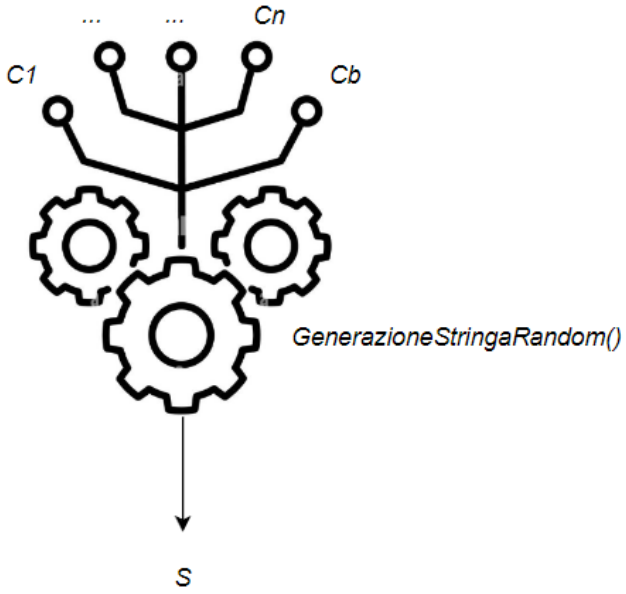
Siano $G_1,\ldots,G_n$ i giocatori della sala virtuale di Mister Joker e B il banco, o più in generale il server, e sia P la generica partita a cui partecipano i vari giocatori. Ogni giocatore al tempo T0 parteciperà al protocollo generando un contributo C_i, $1 < i < n$, così come il banco, che a sua volta parteciperà al protocollo con un suo contributo C_b , calcolati secondo una funzione *GenerazioneContributo()*. Al tempo T1 ogni giocatore invierà il proprio contributo al banco.



Al tempo T2, il banco invierà tutti i contributi, compreso il proprio a ciascuno degli altri giocatori, per poter permettere di effettuare una verifica canonica del risultato dopo la computazione della stringa random:



Una volta ricevuti e inviati tutti i contributi, al tempo T3 il banco effettua una computazione con lo scopo di creare una stringa random tramite la funzione *GenerazioneStringaRandom()* che prenda in input i contributi dei vari giocatori. La funzione in questione deve essere valutata e il risultato deve essere mostrato dal server al tempo T4 a tutti i giocatori, ovvero la stringa random S generata.



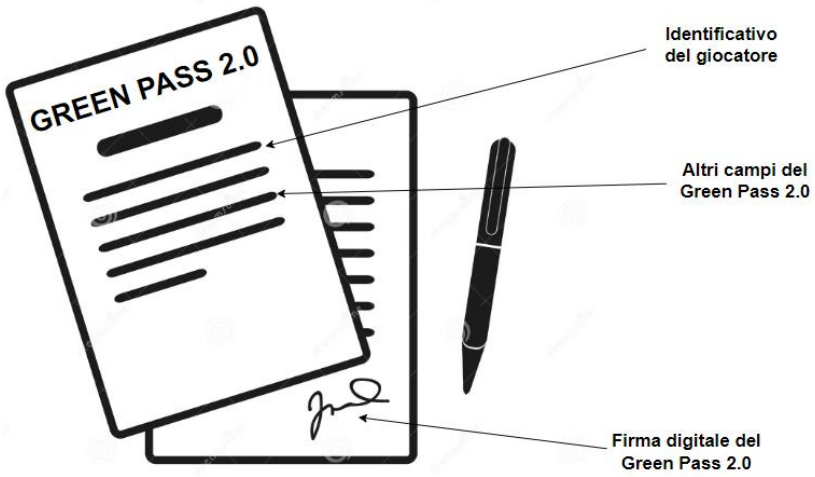
Al tempo T5 ogni giocatore può verificare l'autenticità del risultato e confermare la corretta esecuzione del protocollo, in un caso, o obiettare nel caso in cui il protocollo non sia stato eseguito in maniera onesta. La verifica effettuata da un giocatore è semplicemente una verifica canonica rieseguendo la funzione *GenerazioneStringaRandom()* con input gli stessi contributi utilizzati dal server e inviati da questo ai giocatori. Se tutti i giocatori partecipano in maniera onesta al protocollo, la correctness vale se la probabilità che la funzione *GenerazioneStringaRandom()* dia lo stesso risultato per tutti i partecipanti della sala e questo risultato sia proprio la stringa S sia unitaria:

$$Prob[GenerazioneStringaRandom\_i(C1,...,Cn,Cb)->S]=1 \quad \forall_{1<i<n} i$$

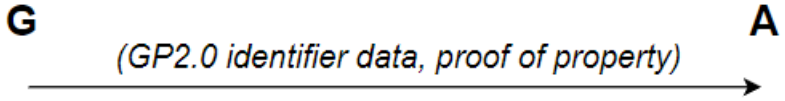
Dove per *GenerazioneStringaRandom_i(),* si intende la valutazione della funzione *GenerazioneStringaRandom()* da parte del giocatore i-esimo.

Seconda funzionalità:

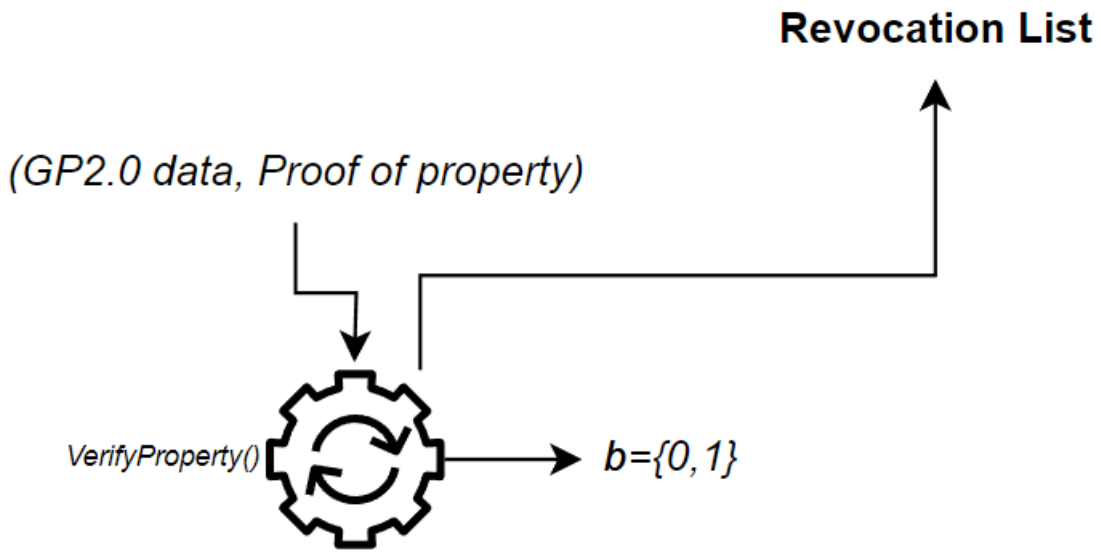
Sia G il generico giocatore e sia A l'applicazione web a cui il giocatore vuole effettuare l'accesso. Per farlo esso dovrà essere dotato di un particolare documento, il Green Pass 2.0, il cui formato dovrà essere definito e rilasciato dall'autorità sanitaria, il documento conterrà le stesse informazioni del Green Pass tradizionale, e in più informazioni aggiuntive richieste dal garante, il tutto associato ad un'unica firma digitale dell'autorità sanitaria. Al tempo T0, quindi, definito il formato del green pass, ogni giocatore può richiedere all'autorità sanitaria il rilascio del proprio documento che sarà associato univocamente a quel giocatore e sarà firmato dal Ministero della Salute:



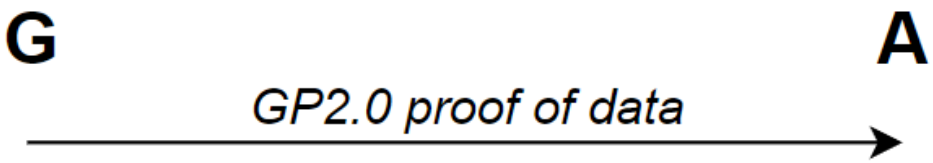
Si può assumere che quest'ultima sia in grado di verificare se le informazioni richieste dagli utenti come dati del documento siano validi, inoltre l'identificativo del giocatore deve rimanere immutato tra un Green Pass 2.0 e un altro dello stesso giocatore così da permettergli di effettuare l'accesso allo stesso profilo di gioco. Ricevuto il Green Pass 2.0, al tempo T1 il giocatore G può identificarsi con l'applicazione A inviando una prova della proprietà del documento e le sole informazioni necessarie all'applicazione per la procedura di autenticazione:



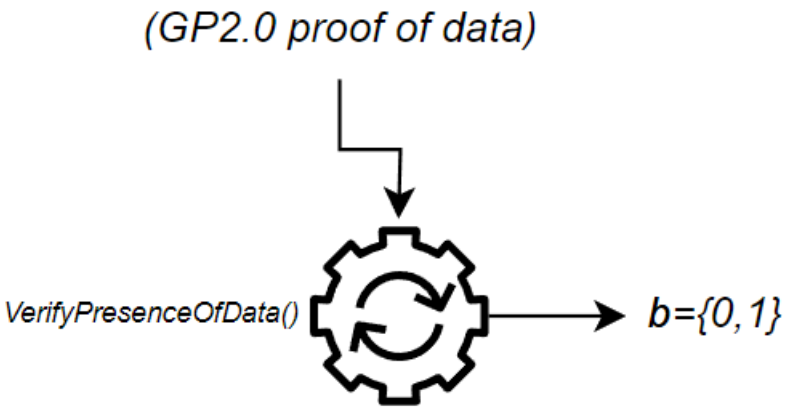
Al tempo T2 l'applicazione può verificare l'autenticità dei dati inviati e permette l'accesso al giocatore, tramite opportuna funzione *VerifyProperty(),* la stessa controllerà anche dati come la scadenza del documento e verificherà che il documento non sia stato inserito nella lista delle revoche del Ministero della Salute:



In caso di verifica andata a buon fine, la funzione restituirà 1, altrimenti 0. Nel caso in cui restituisca 1 si prosegue con il protocollo. Effettuato l'accesso, al tempo T3 il giocatore G può inviare all'applicazione A prova della proprietà di ulteriori dati richiesti dal garante per la partecipazione ai giochi dell'applicazione evitando la diffusione di tutti i dati del documento, ma solo di quelli strettamente necessari:



Al tempo T4 l'applicazione verificherà i dati e permetterà l'accesso ai giochi in caso di verifica andata a buon fine effettuata tramite funzione *VerifyPresenceOfData()*:



Se tutti gli attori si comportano in maniera onesta, e il documento non è scaduto o revocato e contiene tutte le informazioni richieste dal garante, allora la correctness è definita per l'accesso al profilo, se la *VerifyProperty()* valutata dall'applicazione da esito 1 con probabilità unitaria:

Valid GP2.0 data & Valid Proof of Property ⟶ Prob[VerifyProperty(GP2.0data,Proof_of_property) ->1] = 1

Mentre per l'accesso ai giochi, la correctness è definita se l'utente ha già effettuato l'accesso al profilo e se anche la *VerifyPresenceOfData()* valutata dall'applicazione da esito 1 con probabilità unitaria:

Valid GP2.0 proof of data ⟶ Prob[VerifyPresenceOfData(GP2.0_proof_of_data) ->1] = 1

Attori del sistema:

L'accesso alle sale virtuali di Mister-Joker viene effettuato tramite piattaforma online. Qualsiasi cittadino in possesso di Green Pass 2.0 può registrarsi alla piattaforma, identificarsi tramite Green Pass 2.0 e accedere ai giochi mostrando di possedere le informazioni necessarie richieste dal garante. Una figura onesta del sistema è sicuramente l'autorità sanitaria poichè la possibilità che essa agisca in modo malevolo è ridotta al minimo dal rischio elevato di poter perdere credibilità e reputazione.

- **Autorità sanitaria (AS):** istituzione da potersi considerare fidata, ovvero il Ministero della Salute, gestisce le richieste da parte degli utenti dei Green Pass 2.0, ne verifica l'autenticità e ne rilascia il relativo documento, secondo il formato definito. Essa ha inoltre la possibilità, come qualunque certificato digitale, di revocare un qualsiasi Green Pass 2.0, inserendolo in una lista online di revoche a cui ogni utente e applicazione deve fare riferimento;
- **Mister Joker (MJ) :** amministratore della sala virtuale di Bingo, è colui che ha richiesto la progettazione delle funzionalità, gestisce l'applicazione che permette ai giocatori di incontrarsi e giocare nelle sale virtuali;
- **Banco (B)**: figura conducente il gioco nella sala virtuale, partecipa anche lui alla generazione della stringa casuale tramite suo contributo. In generale possiamo assumere che sia sempre presente nella sala virtuale, anche nei giochi in cui non è figura attiva e partecipe, spesso useremo le parole Server e Banco in maniera interscambiabile;

- **Giocatore della Sala Bingo (GSB):** giocatore che, in possesso di Green Pass 2.0 valido, si identifica nella piattaforma di Mister Joker e partecipa alla funzionalità di generazione di stringhe casuali, salvo la verifica da parte dell'applicazione di possesso delle informazioni necessarie;
- **Data Protection Authority (DPA):** garante della privacy che definisce, in maniera dinamica, quali informazioni ogni giocatore deve possedere per partecipare ad un determinato servizio online, in particolare della sala virtuale.

Threat Model

- **Partecipante Manipolatore di Stringhe (PMS):** un giocatore GSB o un banco B è interessato a manipolare la stringa che egli stesso dà come contributo al tempo T0, in modo tale da poterne trarre vantaggio, l'attacco è limitato alla scelta deterministica del suo contributo evitando di usare la funzione *GenerazioneContributo(),* sfruttando eventuali vulnerabilità del meccanismo di generazione di stringhe casuali. Per tale motivo, la sua capacità computazionale è polinomiale (PPT).
- **Avversario Usurpatore (AU):** un avversario è interessato a manipolare i contributi di altri giocatori partecipanti al meccanismo di generazione di stringhe casuali. L'avversario in questione potrebbe essere sia un partecipante della sala, sia un utente esterno, che opera come "Human in the Middle", ovvero che nella comunicazione tra un generico giocatore ed il server al tempo T1, modifica il contributo del giocatore, inviandone uno generato da lui in modo tale da trarne vantaggio o farne trarre ad un suo complice nella sala virtuale. Lo stesso avversario potrebbe avere interesse, al tempo T2, a modificare anche i contributi che il banco invia ai giocatori oppure la stringa random generata, la sua capacità computazionale è considerabile polinomiale (PPT).
- **Partecipanti Cooperanti (PC):** i partecipanti, o avversari in generale, potrebbero cooperare, sia ad un mero livello di informarsi vicendevolmente sul loro contributo per la stringa casuale al tempo T0, o anche potrebbero essere un insieme di avversari precedentemente e successivamente elencati che si uniscono per trarre vantaggio dal meccanismo di generazione delle stringhe. La loro capacità computazionale è polinomiale (PPT).
- **Avversario Coercitivo (AC):** un avversario, esterno alla sala virtuale, o anche un giocatore, potrebbe essere interessato a costringere un giocatore a mandare un determinato contributo al tempo T0, anche intervenendo in maniera violenta, così da trarre vantaggio nel meccanismo di generazione delle stringhe. La sua capacità computazionale è considerabile polinomiale (PPT).
- **Banco Sostitutore (BS):** il banco della sala virtuale potrebbe essere interessato a manipolare la valutazione della *GenerazioneStringaRandom()* o modificare la stringa casuale generata dal meccanismo al tempo T3 e sostituirla con un'altra preparata ad hoc in base alla situazione, in modo tale da trarne vantaggio, magari facendo perdere più giocatori possibili. Il banco potrebbe anche scartare i contributi di alcuni giocatori per ottenere una stringa casuale a lui favorevole. La sua capacità computazionale è considerabile polinomiale (PPT).
- **Giocatore/i Bugiardi (GB):** un giocatore/i potrebbero essere interessati a svolgere il protocollo correttamente, ma mentire sulla corretta esecuzione di tale da parte di altri giocatori o del banco stesso al tempo T5, in modo da screditare quest'ultimo o abortire l'intero processo. Ha/hanno capacità computazionale polinomiale (PPT).

- **Giocatore che Abortisce (GA):** un giocatore potrebbe essere interessato ad abortire il meccanismo di generazione delle stringhe, in qualunque punto del protocollo, poiché riesce a carpire informazioni sul risultato finale, ha capacità computazionale polinomiale (PPT).
- **Possessore Alteratore (PA):** un possessore di Green Pass 2.0 potrebbe essere interessato a manomettere i dati all'interno del documento preservandone la validità secondo l'autorità sanitaria così da avvantaggiarsene per scopi personali di natura fraudolenta, nella finestra temporale [T0,T1], ad esempio potrebbe acquisire dei dati per partecipare ai giochi della sala virtuale che prima non aveva: riuscendo ad eludere i controlli effettuati dalle funzioni *VerifyProperty() e VerifyPresenceOfData()* che daranno risultato 1 nonostante i dati siano stati alterati. La sua capacità computazionale è considerabile polinomiale (PPT).
- **No-fox (NF):** oppositori dell'innovazione che minano l'utilizzo del Green Pass 2.0 a causa della loro natura retrograda e allarmista, vogliono lasciare tutto così com'è, evidenziando le criticità di una nuova tecnologia rimarcando la poca trasparenza di quest'ultima. Hanno capacità computazionali considerabili polinomiali (PPT).
- **Avversario Ladro (AL):** un avversario, che è in grado di reperire il Green Pass 2.0 altrui nella sua interezza o i dati per effettuare l'accesso, ai tempi T1 e T3, potrebbe essere interessato ad identificarsi con quest'ultimo eludendo i controlli della funzione *VerifyProperty()*. Inoltre, potrebbe essere esso stesso a richiederlo all'autorità sanitaria al posto del vero proprietario, al tempo T0, in questo modo potrebbe accedere anche al rispettivo conto virtuale e trarne vantaggi. Ha capacità computazionali polinomiali (PPT).
- **Applicazioni inciucioni (AI)**: degli avversari che, essendo in grado di reperire in modo lecito delle informazioni del Green Pass 2.0, le condividono tra di loro per aumentare la loro conoscenza sul suo possessore dopo l'esecuzione del protocollo. La loro capacità computazionale è considerabile polinomiale (PPT).

Proprietà da soddisfare

Confidenzialità

Generazione Stringa Casuale

- **C1:** Nessun partecipante al protocollo deve essere in grado, al tempo T1, di reperire alcuna informazione riguardante il contributo di altri partecipanti, senza aver prima valutato la funzione GenerazioneContributo().

Formato GP 2.0, identificazione e accesso ai giochi

- **C2**: I dati contenuti nel Green Pass 2.0 che non sono esplicitamente richiesti o inviati dal possessore del documento devono rimanere confidenziali e non devono essere divulgati ad altre entità partecipanti o non al protocollo di identificazione in un'applicazione o per l'accesso ai servizi.
- **C3**: Tutti i dati inviati tra un possessore del Green Pass 2.0 ed il server nella finestra temporale [T0,T3] devono essere trattati come confidenziali e impedirne l'accesso ad entità esterne non autorizzate, non facenti parti del protocollo di identificazione o per l'accesso ai servizi.
- **C4**: Durante il processo di autenticazione o per l'accesso ai servizi, il sistema deve garantire ai partecipanti la possibilità di dimostrare la proprietà delle informazioni desiderate senza doverle rivelare in chiaro alle entità

coinvolte, salvo casi in cui sia strettamente necessario farlo, ovvero quei casi in cui l'applicazione ha bisogno di conoscere necessariamente determinate informazioni in chiaro tra cui, ad esempio, l'identificativo univoco.

- **C5:** Il formato del Green Pass 2.0 deve permettere al possessore del documento di inviare in maniera selettiva le informazioni del documento senza la necessità di dover inviare l'intero Green Pass 2.0.

Integrità

Generazione Stringa Casuale

- **I1**: Nessun'entità al di fuori o all'interno del sistema deve essere in grado di modificare i contributi C_i inviati da un generico giocatore G_i al banco B al tempo T1 e dal banco ai giocatori al tempo T2 durante il protocollo di generazione delle stringhe casuali, lo stesso è valido per la stringa S inviata dal banco ai giocatori al tempo T4.
- **I2**: Più partecipanti al protocollo di generazione di stringhe casuali che si mettono d'accordo sui contributi da inviare al tempo T0 non possono manipolare l'esito della generazione della stringa casuale a loro vantaggio salvo la presenza di almeno un partecipante onesto nel protocollo.
- **I3**: Il banco non deve essere in grado di manipolare l'esito del protocollo di generazione della stringa casuale a suo vantaggio al tempo T3, né deve essere in grado di raggirare il meccanismo per generare una stringa casuale diversa da quella che il protocollo onesto genererebbe e mostrarla in maniera fraudolenta ai giocatori al tempo T4.
- **I4:** Nessun partecipante al protocollo di generazione di stringhe casuali deve essere in grado di interrompere il protocollo per trarne vantaggio, senza rimanere impunito, in qualsiasi momento della generazione.
- **I5:** Tutti e soli i contributi dei partecipanti al protocollo di generazione di stringhe casuali devono essere considerati, senza omissioni, nella generazione della stringa casuale al tempo T3.
- **I6:** Un partecipante che ha contribuito al protocollo di generazione della stringa casuale con un contributo C non deve essere in grado di dimostrare il contrario (ovvero contestare l'esito del gioco giustificandosi negando di aver inviato il contributo C)

Formato GP 2.0, identificazione e accesso ai giochi

- **I7**: Qualsiasi informazione contenuta nel Green Pass 2.0 deve essere immutabile, alterare o manipolare una qualsiasi di queste informazioni provocherà l'immediato rifiuto di tale Green Pass.
- **I8**: Qualsiasi informazione inviata in maniera selettiva dal possessore del Green Pass 2.0 deve poter essere verificabile (ovvero dimostrabile che è stata rilasciata dall'autorità sanitaria).
- **I9**: Un giocatore che intende identificarsi ad una piattaforma tramite Green Pass 2.0 deve essere in grado di fornire una prova di possesso del documento stesso.

Trasparenza

Generazione Stringa Casuale

- **T1:** Ciascun partecipante può verificare l'autenticità e integrità della stringa casuale S mostrata dal banco al tempo T4 in accordo al protocollo di generazione di stringhe casuali, così come l'effettiva considerazione del proprio contributo all'interno del protocollo
- **T2**: Il meccanismo di generazione delle stringhe casuali deve essere pubblico e visibile a tutti i partecipanti al protocollo, così come i dettagli di esso: le varie fasi e l'algoritmo stesso di generazione.

Formato GP 2.0, identificazione e accesso ai giochi

- **T3**: Tutte le operazioni di generazione, gestione, compreso il meccanismo di revoche, e verifica del Green Pass 2.0 devono essere trasparenti e accessibili a tutti gli utenti autorizzati, in modo che possano comprendere il processo e avere fiducia nella correttezza del sistema.
- **T4**: I criteri scelti dal garante per l'accettazione e l'utilizzo del Green Pass 2.0 devono essere chiari, sia per l'identificazione in un'applicazione che per l'accesso ai servizi, e accessibili a tutti gli utenti in maniera pubblica, in modo che essi possano verificare in prima persona quali sono le politiche di accesso agli eventi sociali e le informazioni richieste per parteciparvi.

Efficienza

- **E1:** La generazione di stringhe casuali, così come l'identificazione e l'accesso ai servizi tramite Green Pass 2.0 devono essere efficienti, fornire risposte in tempi rapidi e prevedere un carico computazionale ridotto.

WP2

Prima funzionalità (generazione continua di stringhe casuali).

Il meccanismo di generazione continua di stringhe casuali si basa sulla collaborazione di ogni partecipante che effettua l'accesso ad una sessione di gioco nella sala virtuale, banco compreso. In caso di assenza e/o partecipazione non attiva del banco, sarà comunque il server a partecipare tramite contributo. Le comunicazioni tra ogni giocatore e la piattaforma di gioco vengono effettuate sfruttando TLS 1.3 nella modalità in cui il server ha un certificato digitale rilasciato da un'autorità di certificazione, e il client no. Prima dell'accesso alla sala virtuale, ogni giocatore Gi dovrà innanzitutto accedere al proprio profilo, come definito nella prossima funzionalità, per farlo ogni giocatore dovrà possedere una coppia di chiavi pubblica e privata

(PK,SK) <- ECDH (Curva NIST P-256)

basate sul problema del logaritmo discreto nel setting delle curve ellittiche. La PK e SK, infatti, sono rappresentate da coppie di punti sulla curva ellittica, il relativo schema di firma si basa sulle curve ellittiche e punti su di esse, di conseguenza verrà applicato nel protocollo in formato black box (viene scelta la curva ellittica *P-256* definita nello standard NIST (National Institute of Standards and Technology) con dimensione di chiave pari a 256bit).
Per la generazione delle chiavi, si assume che nessun computer sia corrotto e che quindi nessun avversario possa avere accesso alla SK di nessun attore in gioco. Una volta che l'utente accede nella sala virtuale attende l'ingresso del minimo numero di partecipanti ammessi alla sala virtuale, raggiunto il minimo si procede con il protocollo per la generazione della stringa random.

Si assume, che dato che la seguente funzionalità prevede l'accesso al conto gioco e ai servizi, il server abbia a disposizione l'associazione id univoco giocatore – PK, quindi possiede le PK dei giocatori della sala virtuale.

Al tempo T0 deve essere definito un ordine per ogni partecipante al protocollo, dove per ordine si intende la posizione univoca di ogni partecipante per la concatenazione dei contributi generati dagli stessi, la generazione della stringa random, infatti, come descritto nel seguito, si basa sulla concatenazione dei contributi dei giocatori; per convenzione il banco B invierà le chiavi pubbliche dei vari partecipanti e l'ordine sarà sancito univocamente da quello lessicografico delle chiavi, unica eccezione fa il contributo del banco stesso che verrà sempre concatenato come ultimo valore, anche questo noto in anticipo a tutti i partecipanti. L'ordine è immutabile ed è definito come quello lessicografico delle PK dei partecipanti.

Presa coscienza dell'ordine, al tempo T1 ogni giocatore Gi, banco compreso, valuta la funzione *GenerazioneContributo()* , ovvero genera una stringa random $C_i \leftarrow \{0,1\}^n$,che rappresenterà il contributo del partecipante (per il banco il contributo verrà nominato C_b), dove n è un parametro di sicurezza impostato a 256:

$n = 256,\ C_i \leftarrow \{0,1\}^n,\ C_b \leftarrow \{0,1\}^n$

e genera un commitment del contributo, calcolato in questo modo:

$r_i \leftarrow \{0,1\}^n\ ,\ commit_i = SHA256(C_i \,||\, r_i)\ ,\ commit_b = SHA256(C_b \,||\, r_b)$

sfruttando SHA-256 per la costruzione dello schema di commitment. Dopo aver creato il commitment, esso viene firmato tramite ECDSA:

$sign_i = ECDSA\_signature(commit_i)$

Infine, ogni giocatore, manda sia il commitment che la firma al banco B. Una volta che il banco ha ricevuto il commitment e la firma di ogni giocatore, al tempo T2, verifica la correttezza della firma ed in seguito invia ad ogni giocatore tutti i commitment ricevuti, compreso il suo, e le relative firme digitali, firmando l'intero messaggio ovvero l'insieme di tutti i commitment inviati dal server al banco, suo compreso.

$sign_b = ECDSA\_signature(all\_commits)$

Al tempo T3, a questo punto, ogni giocatore G_i, previa verifica dell'autenticità della firma del banco e dei commitment ricevuti, apre il suo commitment inviando la coppia:

$(C_i\ ,\ r_i) = opening_i$

al banco B. Inoltre, anche il banco apre il suo commitment:

$(C_b\ ,\ r_b) = opening_b$

ed invia, al tempo T4, ad ogni giocatore le aperture dei rispettivi commitment svelando i vari contributi dei giocatori ed il proprio, in modo tale che ogni giocatore possa verificare che i commitment inviati precedentemente siano validi, sia dei giocatori che del banco stesso:

$Vrfy(opening_i, commit_i) => SHA256(C_i , r_i) == commit_i$

$Vrfy(opening_b, commit_b) => SHA256(C_b , r_b) == commit_b$

Al tempo T5 il banco, ricevuti i contributi C_i di ogni giocatore, valuta la funzione *GenerazioneStringaRandom()*, ovvero, concatena, secondo l'ordine stabilito in precedenza, i contributi C_i di ogni giocatore ed il suo C_b, dati n giocatori:

$SHA256(C_1 || C_2 || ... || C_n || C_b) = randomString$

Al tempo T6 il banco invia la stringa casuale a ogni giocatore della sala virtuale. Nell'esempio si assume che l'ordine lessicografico coincida con l'ordine dei numeri naturali 1,2,…,n. Al tempo T7 ogni giocatore, avendo ogni contributo degli altri partecipanti, e l'ordine di concatenazione, può verificare che la concatenazione del suo contributo e degli altri secondo l'ordine stabilito equivale alla stringa casuale calcolata ed inviata dal banco, assumendo che sia il giocatore 1 ad effettuare il controllo (ma viene effettuato da tutti):

$SHA256(C_1 || C_{2\_received} || ... || C_{n\_receiverd} || C_{b\_received}) = randomStringCalculatedByOne$

$Vrfy(randomStringCalcualtedByOne == randomString)$

Introduzione Blockchain

Terminata la spiegazione della prima funzionalità, si pone l'accento sulla possibilità di utilizzare la tecnologia blockchain per modificare la realizzazione del protocollo di generazione di stringhe casuali. La funzionalità è gestita attraverso l'utilizzo di uno smart contract, pubblicato dal banco su una blockchain permissionless denominata BingoChain con annessa criptovaluta BingoCoin. La funzionalità è gestita allo stesso modo di quanto avveniva senza blockchain, con la differenza che ora al tempo T0 ogni giocatore invierà espressa richiesta di partecipare alla partita, inviando al server una richiesta firmata della volontà di partecipare con annessa la quantità di criptovaluta da depositare sullo smart contract, sia per pagare le transaction fees eventuali dovute alle transazioni sulla blockchain, sia come garanzia che il partecipante si comporti in maniera onesta. Al tempo T1 quindi, il server pubblicherà sulla blockchain le varie richieste dei partecipanti, compresa la sua, con i relativi depositi sullo smart contract. In questo caso l'ordine lessicografico delle chiavi per la concatenazione è definito espressamente nel codice dello smart contract stesso. A questo punto viene definito un timeout alla fine del quale, se nessun commitment è stato pubblicato sulla blockchain, lo smart contract restituirà i vari soldi ai partecipanti. Al tempo T2, definiti i partecipanti, essi invieranno su un canale secondario, state channel, e non direttamente sulla blockchain, i vari commitment al server, quindi tramite comunicazione TLS 1.3, firmati, il server effettuerà gli stessi controlli della funzionalità senza blockchain al tempo T3, prima di pubblicare sulla blockchain, per evitare computazione da parte dello smart contract che dia esito negativo, e dopo aver effettuato i controlli pubblica i commitment sulla blockchain. A questo punto al tempo T4 ogni giocatore, banco compreso, apre il commitment,

ancora una volta i giocatori comunicheranno con il server con TLS, senza passare per la blockchain, ma in questo caso firmando l'apertura, non per ragioni legate strettamente al protocollo, ma per la necessità della blockchain di attribuire il costo della transazione ad un particolare utente. Il server effettuerà i relativi controlli al tempo T5 e pubblicherà sulla blockchain. Al tempo T6, in accordo ai contributi pubblicati sulla blockchain sia il server che i giocatori possono calcolare la stringa random in accordo al protocollo.
In caso qualche giocatore decida di non aprire al tempo T4, o il banco stesso, chiunque può pubblicare sulla blockchain la propria apertura firmata, costringendo in questo modo ogni altro partecipante ad aprire, pena la perdita della criptovaluta inizialmente depositata.
Schematizzato in pseudocodice lo smart contract è definito nel seguito.

Innanzitutto, il contratto gestisce una hash map partecipants che associa ogni PK dei partecipanti ad una struttura contenente il quantitativo di criptovaluta BingoCoin depositata, il commitment inviato e un valore booleano per verificare se il commitment è stato aperto o meno, inizializza il valore di alcune costanti per definire un commitment non ancora inviato o un apertura non pervenuta, infine gestisce due variabili: un timeout, descritto nella funzionalità in alto, e un targetBlockNumber che definisce il massimo numeri di blocchi entro i quali, nel caso di apertura singola del commitment, tutti gli altri giocatori devono aprire.

```
contract BingoGame{
  HashMap partecipants;
  const int NOCOMMITMENT=-1;
  const bool NOOPENING=false;
  int targetBlockNumber=-1;
  time timeout = null;

  function addPlayers(Players[] players)
  {
    for each player in players
    {
      if(checkSignECDSA(player.sign,player.key,player.amount)==OK){
        if(players.amount<=100){
          clear(partecipants);
          abort("The player %s has not inserted the correct amount!",player.key);}
        partecipants[player.key]=[player.amount,NOCOMMITMENT,NOOPENING];}
      else{
        clear(partecipants);
        abort("The player %s has an invalid sign!",player.key);}
    }
    start(timeout);
    publishOrderedPKOnBlockchain()
  }
```

La funzione addPlayers verrà richiamata dal banco dopo aver ricevuto le richieste di partecipazione e di deposito della criptovaluta sullo smart contract da parte dei giocatori, firmate, la funzione infatti verifica la firma e la quantità di criptovaluta (per semplicità si assume che nessun partecipante depositi più della somma richiesta, ovvero 100 BingoCoin) poiché chiunque potrebbe depositarne meno. Se i controlli vanno a buon fine, lo smart contract pubblica sulla blockchain le PK dei partecipanti e aggiorna la hash map.

La funzione addCommitments viene chiamata dal banco dopo aver ricevuto i commitment firmati dei partecipanti, lo smart contract effettua la verifica della firma e associa nella hash map la PK del partecipante al proprio commitment, inoltre pubblica sulla blockchain se tutte le transazioni sono corrette (anche una sola firma errata fa abortire la funzione). Anche la funzione addOpenings è chiamata dal banco una volta ricevute le aperture dei giocatori, lo smart contract verifica che le firme siano corrette e che anche le aperture lo siano, altrimenti abortisce, e nel caso in cui la verifica va a buon fine pubblica sulla blockchain i contributi dei partecipanti.

```
function addCommitments(Commitments[] commitments)
{
  for each commitment in Commitments
  {
    if(checkSignECDSA(commitment.sign,commitment.content,commitment.player_key)==OK){
      partecipants[commitment.player_key].commitment=commitment;}
    else{
      clear(partecipants.commitment);
      abort("The player %s has an invalid sign on the commitment!",commitment.player_key);}
  }
  if(AllHaveCommitted(partecipants)){
    publishCommitmentOnBlockchain()}
  else{
    clear(partecipants.commitment);
    abort("Not all the player have sent the commitment");}
}
```

```
function addOpenings(Openings[] openings)
{
  for each opening in Openings
  {
    if(checkSignECDSA(openings.sign,openings.salt,openings.contribution,openings.player_key)==OK){
      if(checkCommitment(opening.salt,opening.contribution,partecipants[opening.player_key].commitment)){
        partecipants[opening.player_key].is_opened=true;}
      else{
        clear(partecipants.is_opened);
        abort("The player %s has an invalid opening on his previous commitment!",opening.player_key);}}
    else{
        abort("The player %s has an invalid sign on the opening!",commitment.player_key);}}
  }
  if(AllHaveOpened(partecipants)){
    publishOpeningsOnBlockchain()}
  else{
    clear(partecipants.is_opened);
    abort("Not all the player have opened the commitment");}
}
```

La funzione addSingleOpening, invece, è chiamata da chiunque, rispecchia lo scenario, in cui qualunque partecipante decida di non aprire, descritto in precedenza, lo smart contract verificherà la firma e l'apertura singola, inoltre definisce un numero di blocchi entro il quale tutti i partecipanti devono aprire il commitment.

```
function addSingleOpening(Openings opening)
{
  if(checkSignECDSA(openings.sign,openings.salt,openings.contribution,openings.player_key)==OK){
    if(targetBlockNumber==-1){
      targetBlockNumber=getCurrentBlockNumber()+len(partecipants)+10;}
    else if(targetBlockNumber<getCurrentBlockNumber()){
      abort("You have not opened in time, funds has gone");}
    if(checkCommitment(opening.salt,opening.contribution,partecipants[opening.player_key].commitment)){
      partecipants[opening.player_key].is_opened=true;}
    else{
      abort("You have not opened right");}
    }
    else{
        abort("You have has an invalid sign on the opening!",commitment.player_key);}

}


function checkBlockNumberAndTimeout()
{
  if(targetBlockNumber!=-1 && targetBlockNumber<getCurrentBlockNumber()){
    int disonestAmount=getDisonestAmount(partecipants);
    int onestAmount=getOnestAmount(partecipants);
    returnFunds(getOnestPlayers(partecipants),disonestAmount+disonestAmount);
    targetBlockNumber=-1;}
  if(elapsed(timeout) && partecipants.commitment.isEmpty()){
    returnFunds(getPlayers(partecipants),getAmount(partecipants));}
}
```

La funzione checkBlockNumberAndTimeout viene valutata dallo smart contract stesso in maniera iterativa per controllare se non si sia raggiunto il limite massimo per l'apertura dei commitment, in quel caso chi non ha aperto perderà la criptovaluta depositata, mentre i giocatori onesti riceveranno un rimborso ed inoltre anche una parte della criptovaluta dei giocatori disonesti. Si fa notare che nel caso in cui tutti aprano entro il blocco target la funzione semplicemente restituirà tutti i soldi agli onesti (tranne quelli utilizzati per le transaction fees). Nello pseudocodice proposto è stata rilassata la proprietà dello smart contract di accedere a dati associati alla transazione come ad esempio quanti soldi sta trasferendo una transazione, o qual è la PK di chi ha invocato questa transazione e anche informazioni sullo stato della blockchain, e per semplicità queste informazioni sono passate in input alla funzione stessa.

Seconda funzionalità (Green pass 2.0 e autenticazione)

Come stabilito nella precedente funzionalità, la comunicazione fra giocatore e server viene effettuata tramite TLS 1.3, nella modalità in cui il server ha un certificato digitale e il client no. Inoltre, ogni attore dovrà possedere una coppia di chiavi pubblica e privata:

(PK,SK) <- ECDH (Curva NIST P-256)

basate sul problema del logaritmo discreto nel setting delle curve ellittiche (ECDSA). Anche l'autorità sanitaria avrà una coppia di chiavi:

(PK_a,SK_a) <- ECDSA (Curva NIST P-256)

Innanzitutto, viene definito il formato del Green Pass 2.0, la sua modalità di stesura e la modalità di firma dell'autorità sanitaria. Il Green Pass 2.0 sarà un Merkle Tree, una costruzione ad albero binario che sfrutta le CRHF, in cui ogni foglia rappresenta un'informazione semplice del documento. Per informazione semplice si intende un'informazione non scindibile in ulteriori informazioni. Prima di inserire l'informazione nella relativa foglia, si calcolano le randomness r_i e s_i:

$$r_i \leftarrow \{0,1\}^n, \; s_i \leftarrow \{0,1\}^n, \; n = 256$$

e data l'informazione Info, la foglia avrà come valore:

$$SHA256(SHA256(r_i \,||\, Info_i) \,||\, s_i)$$

ovvero il commitment del commitment dell'informazione stessa. In caso di mancata informazione di un particolare campo, verrà conservato un valore marker:

$$SHA256(SHA256(0) \,||\, s_i)$$

ovvero il commitment del marker SHA256(0). Inoltre, viene stabilito un ordine univoco per ogni Green Pass 2.0 nel quale compariranno le varie informazioni appartenenti sia al vecchio Green Pass, sia le nuove informazioni da aggiungere; informazioni importanti da includere nel Green Pass sono la PK del richiedente e un ulteriore identificativo univoco, utilizzato per permettere ad un possessore di Green Pass 2.0 scaduto o revocato di richiederne un altro e poter comunque accedere agli stessi profili con cui si identificava con il vecchio documento. L'identificativo è un contatore che l'autorità sanitaria gestisce, ogni nuova richiesta incrementa il contatore e salva il nuovo identificativo. Nel caso in cui la richiesta sia accompagnata dal precedente Green Pass, di cui si deve dimostrare la proprietà, il contatore non sarà incrementato, ma il vecchio identificativo verrà riciclato nel nuovo Green Pass 2.0. Quindi il formato del Green Pass sarà qualcosa di questo tipo:

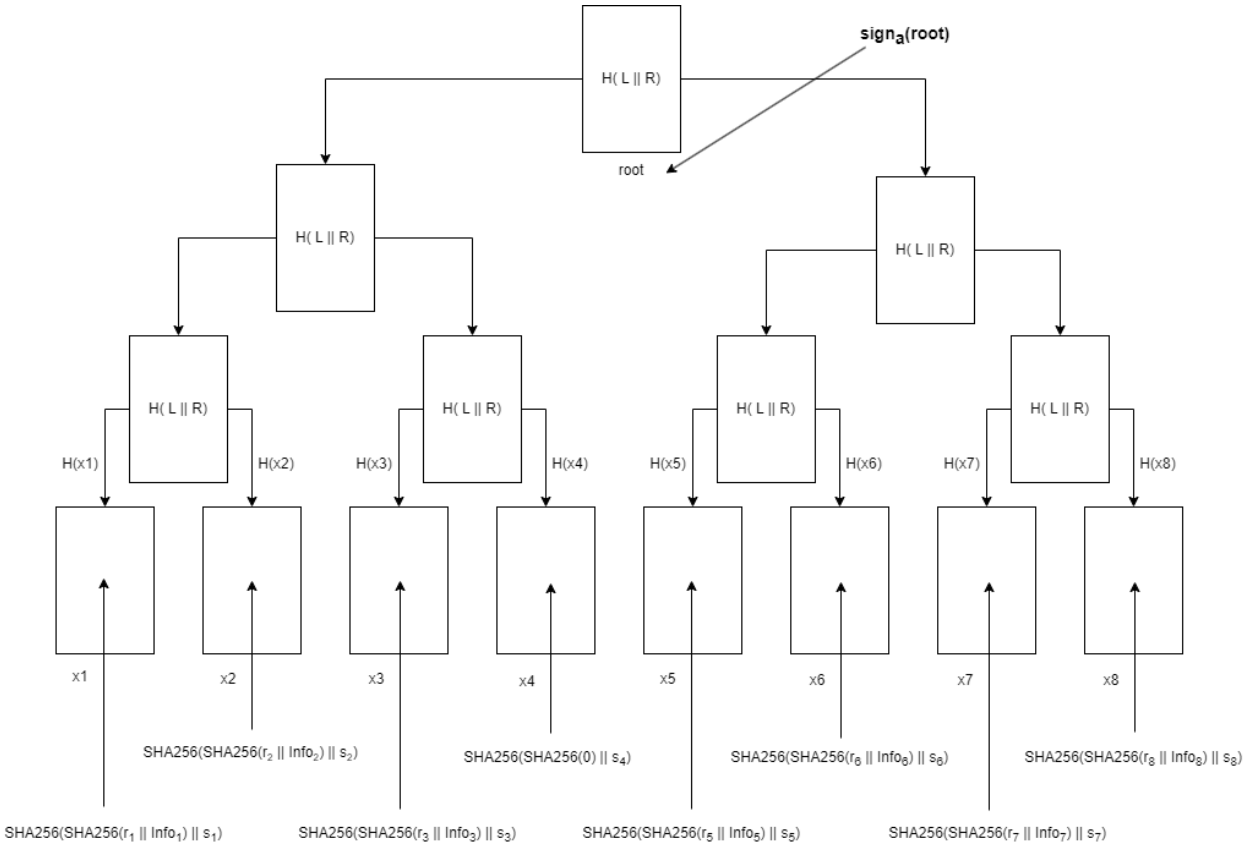


Figure 1: Green Pass 2.0 struttura: Merkle Tree

in cui nelle foglie intermedie si effettua l'hash della concatenazione del risultato di sinistra (L) con quello di destra(R) con SHA256. Una volta formato il Merkle Tree, l'autorità sanitaria pone una firma unica alla radice del medesimo, effettuata tramite ECDSA:

$sign_a = ECDSA\_signature(root)$

A questo punto il formato del Green Pass 2.0 è fissato, ovvero Merkle Tree con annessa firma dell'autorità sanitaria, e si può procedere con la funzionalità prevista.

Innanzitutto, al tempo T0 chiunque voglia ottenere il proprio Green Pass 2.0 deve effettuare una richiesta firmata, secondo ECDSA, da inviare all'autorità sanitaria, includendo i valori che i campi del Green Pass 2.0 devono assumere e che l'autorità sanitaria stessa sarà in grado di verificarne l'autenticità:

request -> GP2.0 data, signRequest=ECDSA_signature(request)

Inoltre, se si è possessori di un precedente Green Pass 2.0 scaduto o revocato, e si volesse mantenere lo stesso identificativo univoco per i servizi online, bisognerà inviare anche le seguenti informazioni all'autorità sanitaria: foglia contenente la PK, foglia contenente la scadenza e foglia contenente l'identificativo univoco, e le relative aperture di commitment inviando PK, scadenza e identificativo con i rispettivi r_i ed s_i, oltre che la firma dell'autorità sanitaria sulla radice del vecchio Green Pass 2.0. Inoltre, dovrà inviare anche il percorso per risalire l'albero.

Considerando che, nell'ordine stabilito, queste informazioni si trovano rispettivamente al sesto, settimo e ottavo posto, dovrà inviare:

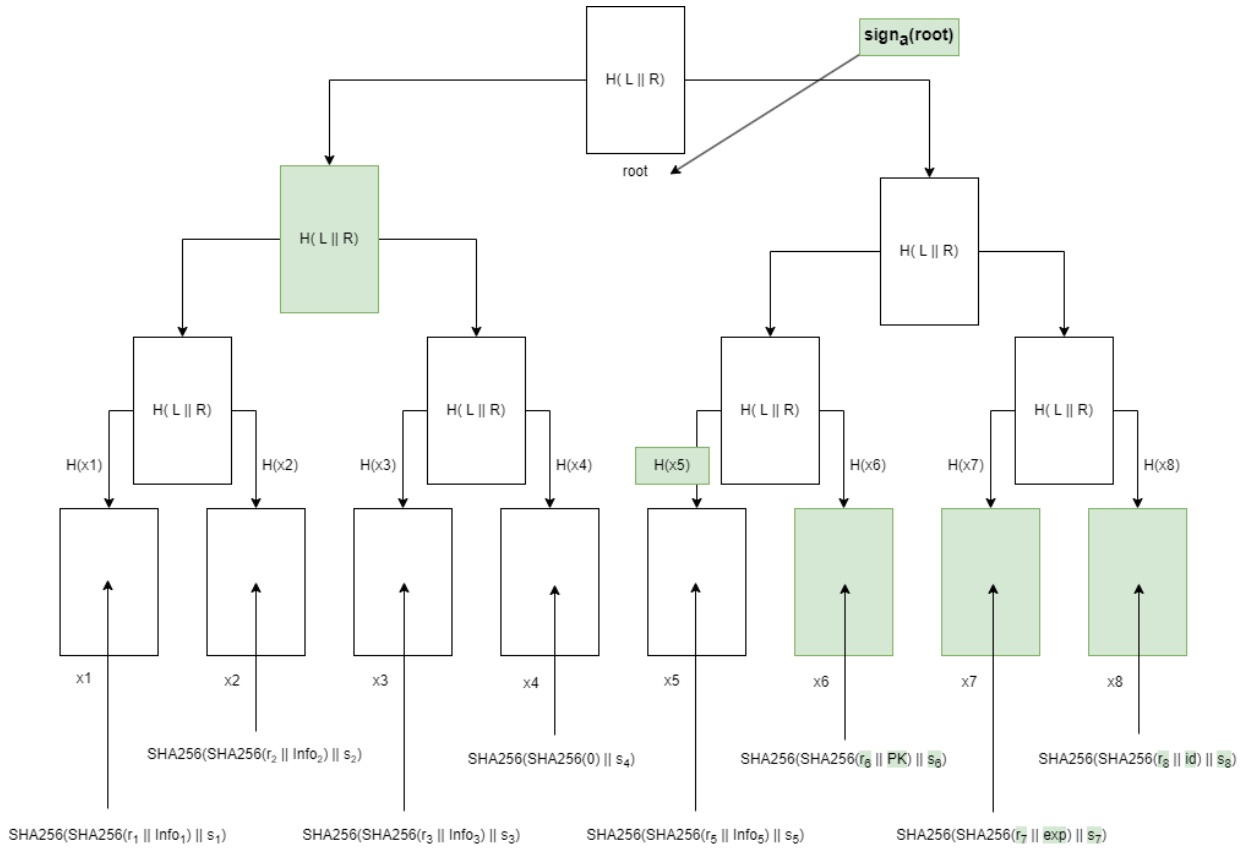


Figure 2: Informazioni per accesso alla piattaforma o richiesta ulteriore di Green Pass 2.0

In caso di richiesta effettuata in questo modo l'autorità sanitaria dovrà innanzitutto verificare che le informazioni siano corrette, aprendo i commitment e risalendo l'albero fino ad arrivare alla radice, verificando la validità della firma. A questo punto l'autorità verifica se il Green Pass 2.0 è scaduto, controllando il dato exp, o revocato, facendo affidamento alla lista di revoche, oltre a verificare se un ulteriore Green Pass 2.0 è stato già rilasciato. Se la verifica va a buon fine l'autorità sanitaria può confrontare la firma del richiedente sulla richiesta rispetto alla PK inviata, se la firma è valida, può confermare la proprietà del documento e rilasciare il nuovo Green Pass 2.0, verificando la validità dei dati richiesti, con identificativo unico l'id presente nel documento. In caso di nuova richiesta, invece, l'autorità sanitaria, rilascerà direttamente il Green Pass 2.0 incrementando il contatore ed inserendo un nuovo identificatore univoco ed inserendo come PK la stessa da cui ha ricevuto la richiesta. Il rilascio del Green Pass 2.0 prevede la creazione del Merkle Tree come descritto, e la firma dell'autorità sulla radice.

Ottenuto il Green Pass 2.0, ogni giocatore può effettuare l'accesso al proprio conto gioco collegandosi al server. Al tempo T1, stabilita la connessione tra server e client**,** tramite TLS 1.3, il server richiede al client di inviargli un Green Pass 2.0 valido e le foglie del documento che riguardano la PK, la data di scadenza del medesimo e l'identificativo univoco, inoltre genera dei random bits e li invia al client:

$random\_bits \leftarrow \{0,1\}^n$, $n=256$

Al tempo T2 il giocatore invierà le foglie corrispondenti alla propria PK, alla data di scadenza del Green Pass 2.0 e l'identificativo univoco, in chiaro, ovvero inviando anche i valori stessi di PK, scadenza e id, inoltre apre i commitment inviando i relativi r_i e s_i oltre che al percorso per salire l'albero, oltre che la firma della radice. inoltre firma tramite PK i random bits inviati dall'applicazione in modo tale che l'applicazione possa verificarne l'appartenenza, tramite funzione *VerifyProperty()*, ovvero verificare che la firma dei bit random sia congrua alla PK presente nell'albero, come prova della proprietà del documento. Come descritto in precedenza per l'invio del Green Pass scaduto/revocato all'autorità sanitaria, il giocatore invierà:

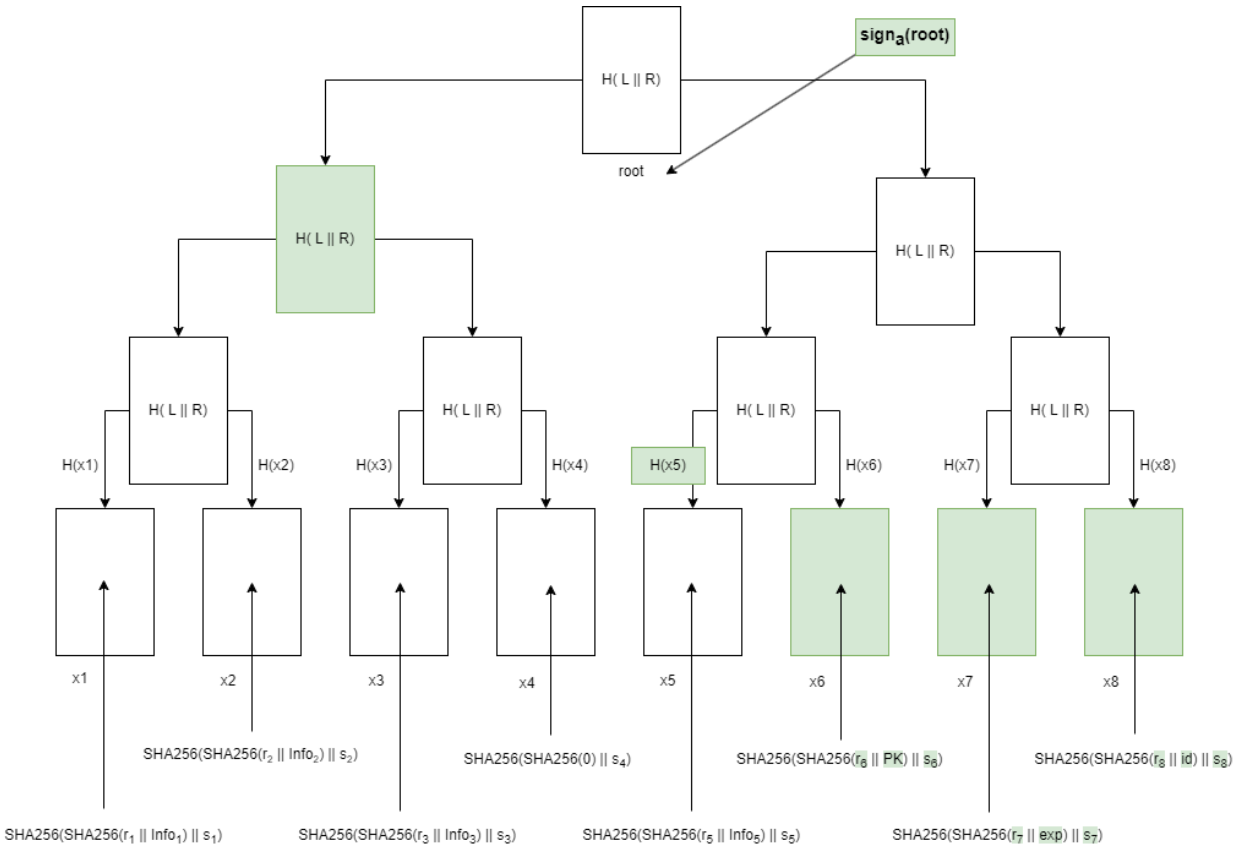


Figure 2: Informazioni per accesso alla piattaforma o richiesta ulteriore di Green Pass 2.0

Al tempo T3, l'applicazione, ricevute le varie informazioni, ricostruisce l'albero, conoscendone il formato, e ottiene l'hash della radice, successivamente verifica, tramite PK_a dell'autorità sanitaria, che la firma della radice calcolata costruendo l'albero corrisponda alla firma inviata del giocatore. Inoltre, verifica che il Green Pass 2.0 inviato non sia registrato come revocato, oltre che valutare la *VerifyProperty()*. Se la verifica da parte dell'applicazione va a buon fine, l'utente ottiene l'accesso al suo profilo di gioco personalizzato in base all'identificativo univoco.

Ora per accedere effettivamente alle sale virtuali il giocatore dovrà inviare all'applicazione ulteriori informazioni richieste dal garante per partecipare attivamente ai giochi. Le informazioni cambiano di giorno in giorno, a discrezione del garante; quindi, l'applicazione dovrà allinearsi alle direttive e richiedere eventuali informazioni al giocatore. Al tempo T4, quindi, il server le richiede al giocatore.

Il giocatore, al tempo T5, verifica che le informazioni richieste dall'applicazione siano in linea con le direttive del garante, pubblicamente visionabili, ed invierà le informazioni richieste, inviando la foglia corrispondente e l'apertura del commitment più esterno, inviando s_i e SHA256(r_i || $Info_i$). Ad esempio, se l'informazione da possedere è il vaccino (denominato Vacc nell'esempio), manderà le seguenti informazioni:

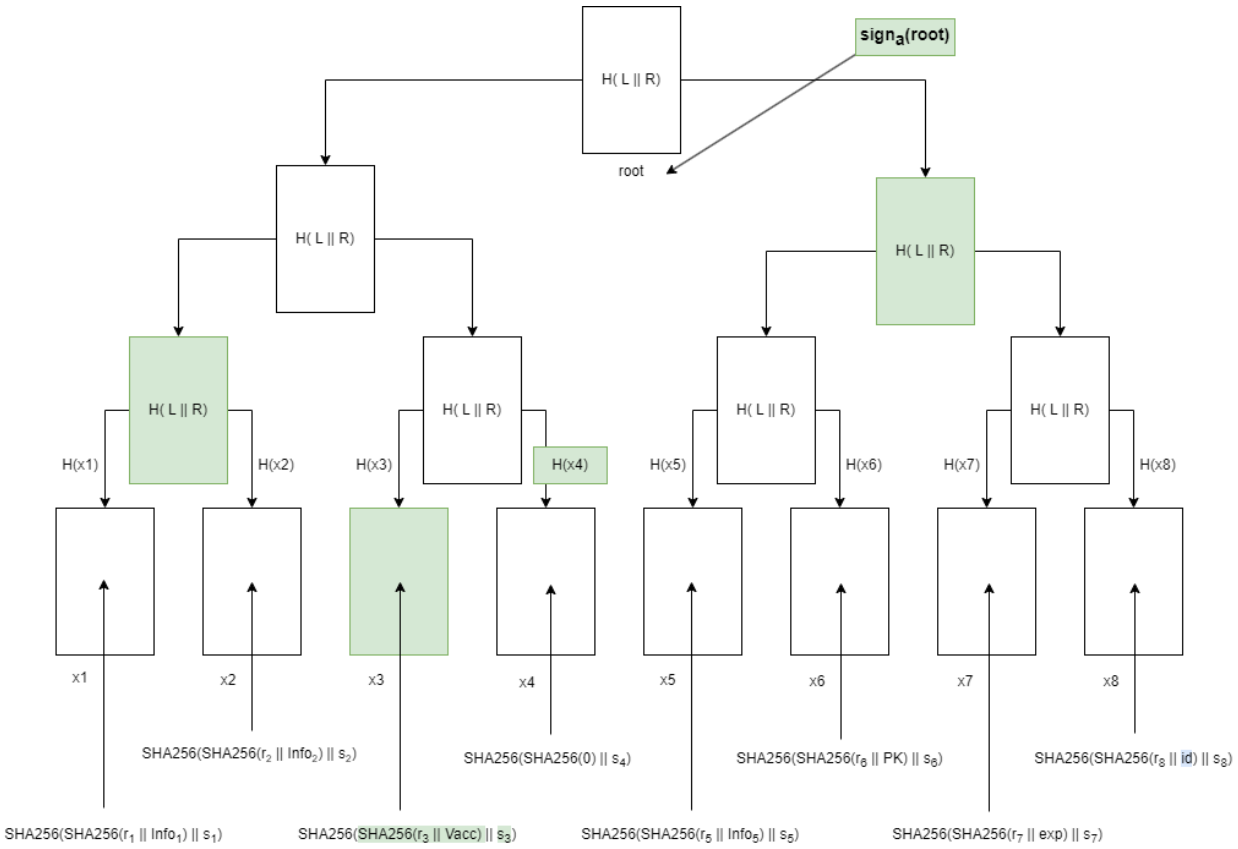


Figure 3: Accesso al gioco tramite invio dell'informazione Vacc

Inoltre, dato che l'applicazione ha già identificato l'utente tramite Green Pass 2.0, può verificare che i dati appartengono allo stesso confrontando la root calcolata nella fase di identificazione rispetto a quella calcolata in questo step, se sono uguali sicuramente si tratterà dello stesso Green Pass 2.0 e accetterà i dati.

Al tempo T6 l'applicazione valuta la funzione *VerifyPresenceOfData(),* ovvero effettua il seguente controllo: per ogni informazione inviata, verifica che l'hash inviato non corrisponda al valore marker, hash di 0:

SHA256(r_i || $Info_i$) != SHA256(0)

in questo modo è sicura che l'informazione richiesta sia effettivamente presente nel campo, ad esempio se un giocatore è vaccinato con uno tra dieci vaccini diversi, l'applicazione potrà sapere che egli è vaccinato ma non con quale vaccino. A questo punto, se la verifica va a buon fine, il giocatore può partecipare al gioco.

WP3

Completezza

Si fa notare che se tutti i partecipanti seguono il protocollo descritto in maniera onesta, sia la generazione della stringa casuale, che l'accesso al/ai sito/giochi vanno a buon fine. Si fa notare, inoltre, che anche nel caso in cui si adotti la soluzione con la blockchain, nessuna autorità giudiziaria sarà interpellata in caso di frode data l'automatizzazione di sanzioni dello smart contract. Ovviamente l'analisi seguente prevederà l'impossibilità di poter mitigare attacchi attuati nel mondo reale.

Analisi della prima funzionalità (generazione stringa casuale)

Nella prima funzionalità viene stabilito uno schema di commitment per nascondere i contributi dei vari partecipanti in una fase iniziale del meccanismo. Questa scelta è dovuta al fatto che, chiunque riesca ad intercettare il valore del contributo di un altro giocatore, potrebbe carpire informazioni sull'esito finale del meccanismo di generazione della stringa, in contrasto con la proprietà **C1**. Lo schema di commitment è definito tramite SHA256 che essendo euristicamente una collision resistent hash function, ma soprattutto un random oracle, permette la costruzione di uno schema di commitment efficiente che gode delle proprietà di binding e hiding. Binding vale poiché chiunque invii il commitment di un contributo, non potrà mai trovare un altro contributo che aperto dia esito positivo nella verifica, perché ciò significherebbe trovare una collisione in SHA256 stessa, inoltre vale hiding perché dato che SHA256 è euristicamente un random oracle ogni avversario è riducibile ad un avversario che fa brute force, dati due contributi C0 e C1 e il commitment commit di uno di loro due esisteranno due randomness r0 e r1 che produrranno commit, quindi l'avversario non è in grado di capire cosa si nasconde dietro al commitment. La scelta di uno schema di commitment con SHA256, al posto dello schema di commitment basato su DLog, ricade sul fatto che non è definito chi sceglie il sampling della funzione CR, se fosse il server ad inviare la quadrupla (p,q,g,h) conoscendo h potrebbe aprire a piacere il proprio commitment, così come collaborare con altri giocatori per permettere a loro di aprire a piacere, cosa non possibile con SHA256, nascerebbe quindi il problema di chi è a scegliere questa quadrupla. Il tutto vale anche se il contributo non è intercettato, ma semplicemente inviato al banco, poiché se fosse inviato in chiaro, allora a questo punto anche il banco potrebbe, avendo i contributi dei giocatori in precedenza, generarne uno a lui vantaggioso e viceversa, nel caso in cui sia lui il primo a mandare il contributo, coalizioni tra i giocatori potrebbero portare alla generazione di una stringa a loro vantaggio dato che il meccanismo è pubblico e visibile, rispettiamo in questo modo **C1**, infatti nessun giocatore riesce a reperire informazioni sulla valutazione della funzione GenerazioneContributo() di altri partecipanti senza aver prima valutato la propria. Dato che ogni giocatore comunica con il server tramite TLS 1.3, rispettiamo la proprietà **I1**; infatti, TLS crea tra client e server un tunnel inattaccabile e fornisce dati sia confidenziali che autenticati.

Grazie a TLS 1.3 l'avversario usurpatore non riesce ad operare come "Human in the Middle", poiché TLS permette di evitare tali attacchi, inoltre egli non riesce ad avere accesso né al commitment di un giocatore né alla sua apertura,

né tantomeno alla stringa random inviata dal server ai giocatori. Anche nel caso di attuazione del protocollo tramite blockchain, la comunicazione è protetta tra client e server, ma nel caso di attuazione del meccanismo di apertura singola, quando qualche partecipante non si sta comportando in maniera onesta, pubblicherà semplicemente una transazione sulla blockchain, neutralizzando direttamente questo tipo di avversario, poiché grazie alla presenza della firma con chiave privata nella transazione un avversario non può manometterla.

Il commitment, inoltre, è firmato secondo lo schema ECDSA. Grazie alla firma rispettiamo anche la proprietà **I6**, infatti dato che il commitment è firmato, nessun giocatore può negare di aver inviato un particolare contributo, poiché è dimostrabile tramite firma, nessun altro, infatti, conosce il logaritmo discreto, ovvero la chiave segreta ECDH di un partecipante, di conseguenza nessuno può forgiare una firma al posto di un'altra persona, poiché significherebbe rompere il logaritmo discreto. A questo punto, il server invia a tutti i giocatori i commitment e le relative firme, compreso il proprio, il tutto firmato ulteriormente dal server, che a questo punto non può negare di aver ricevuto quella serie di commitment specifica. In questo modo ogni giocatore può verificare la veridicità dei commitment, ma soprattutto, essendo i commitment firmati, il banco non potrà, dopo aver ricevuto le aperture, scambiare commitment tra i vari giocatori, magari anche alleati, poiché ciò significherebbe invalidare la loro firma e l'ordine stabilito in precedenza. Infatti, uno scambio del genere, dato che il commitment è firmato, è irrealizzabile poiché anche la firma sarebbe giudicata invalida da tutti i partecipanti onesti del protocollo. Ogni giocatore, in seguito, effettua l'apertura, banco compreso, che invierà a tutti i partecipanti le aperture ricevute e provvederà subito dopo a concatenare i contributi tramite SHA256 secondo l'ordine stabilito in precedenza. La scelta di SHA256, in questo caso, è dovuta alla sua caratteristica di essere euristicamente assimilabile ad un random oracle, in questo modo qualsiasi modifica anche ad un solo bit in input provoca un output totalmente diverso. Grazie alla concatenazione dei contributi in ordine prestabilito tramite SHA256 preserviamo le proprietà **I2** , **I3 , I5, T1 e T2**. Rispettivamente partecipanti che si mettono d'accordo sul contributo da inviare non possono manipolare il meccanismo, poiché essendo SHA256 un random oracle basta un solo input casuale e non deterministico definito da loro, per garantire randomicità nella stringa finale calcolata e ciò è garantito dalla presenza di almeno un partecipante onesto nel protocollo. Il banco, inoltre, per lo stesso motivo non può manomettere il meccanismo senza essere scoperto, poiché essendo verificabile dai giocatori, chiunque scoprirebbe la manomissione, così come sostituire la stringa con una casuale sarebbe identificato subito dai partecipanti alla sala. Grazie alla verificabilità del protocollo chiunque può verificare che il proprio contributo è stato considerato, poiché avendo a disposizione i contributi di tutti gli altri giocatori, può effettuare una verifica canonica della stringa random calcolata conoscendo il protocollo che è pubblico e visibile per tutti. Per le stesse motivazioni preserviamo anche la trasparenza poiché il meccanismo è pubblico e visibile a tutti così come i vari dettagli, come l'ordine univoco, l'invio dei commitment, le aperture e la modalità di generazione tramite concatenazione, proprio per questo motivo ciascun partecipate può verificare l'autenticità della stringa random calcolata.

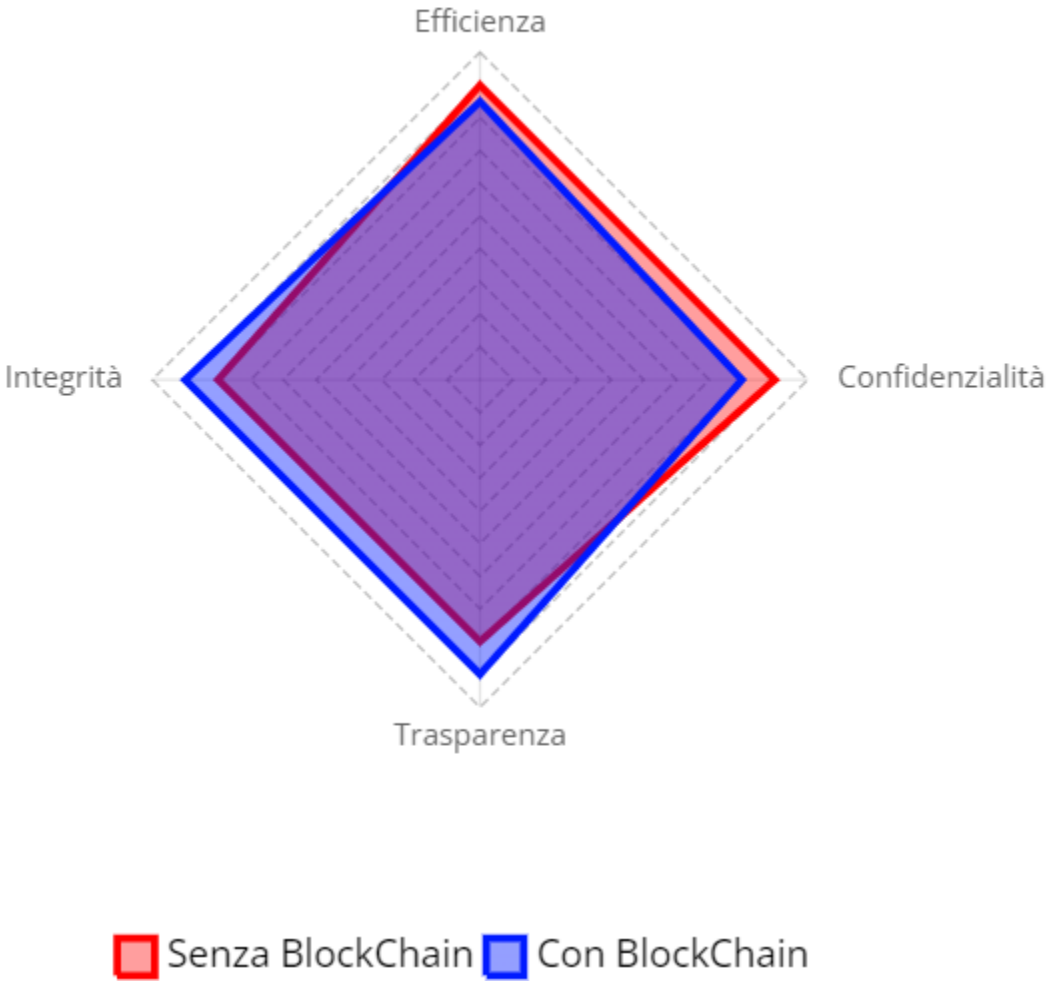
Grazie alla concatenazione ordinata dei contributi dei vari giocatori, riusciamo a neutralizzare gli attacchi di un partecipante manipolatore di stringhe, poiché nonostante non segua in maniera onesta il protocollo, non compromette la generazione della stringa casuale, poiché utilizzando SHA256 un contributo deterministico non fa

differenza, il risultato sarà sempre una stringa casuale (considerando che esista almeno un partecipante che agisca in maniera onesta). Inoltre, riusciamo a contrastare anche i partecipanti cooperanti, per lo stesso motivo, essere a conoscenza del meccanismo e poterne controllare l'input non implica prevederne l'output a meno che non si conosca l'intero input, ciò significherebbe una collaborazione totale, server compreso, inutile poiché non genera vantaggi. Inoltre, in generale, questo tipo di collaborazioni sono difficilmente attuabili, poiché essendo una sala virtuale, i vari avversari dovrebbero fidarsi gli uni gli altri e un solo disonesto fra di loro potrebbe far saltare la cooperazione. Un altro avversario che viene neutralizzato è il banco sostitutore, infatti definito l'ordine univoco di concatenazione, egli non potrà in alcun modo alterare l'ordine, anche grazie al fatto che la firma dei vari giocatori sul commitment lega univocamente quel commitment alla propria PK, utilizzata per definire l'ordine lessicografico, così come non può alterare la stringa casuale finale, poichè essendo il meccanismo noto ai partecipanti essi possono verificarne il corretto svolgimento e denunciare in caso di manomissione. Inoltre egli non può neanche negare di aver ricevuto dei contributi grazie alla firma effettuata nell'invio dei commitment che attesta la loro ricezione da parte del server. Il giocatore bugiardo, invece, non riesce a screditare nè il server nè gli altri giocatori poiché grazie alla firma del commitment non può negare di aver inviato un determinato contributo, così come non può negare che il meccanismo sia stato svolto correttamente poichè pubblicamente noto e verificabile. L' avversario coercitivo, in generale, agisce nel mondo reale, di conseguenza è difficile per noi combattere un tale tipo di avversario. Nonostante ciò, il nostro meccanismo di generazione di stringhe casuali, è comunque una buona arma di difesa contro questo avversario, poichè, a meno che non costringa tutti gli altri partecipanti al meccanismo, ci sarà almeno un giocatore onesto non costretto che, grazie al suo contributo casuale renderà il risultato finale aleatorio, grazie all'utilizzo di SHA256 per concatenare i contributi dei partecipanti.

Infine, ultima analisi, merita il giocatore che abortisce, inteso anche come il server stesso, questo tipo di avversario è abbastanza fastidioso. Esso, infatti, nel caso di funzionalità senza blockchain riesce ad effettuare questo tipo di attacco: in seguito all'apertura degli altri partecipanti, avendo quindi a disposizione i loro contributi, può verificare il risultato finale e, in base ad esso, decidere se aprire o meno, causando, nel secondo caso, un abort nel protocollo. Questo avversario non è mitigato dal protocollo, mentre in caso di implementazione tramite blockchain, lo smart contract provvederà ad elargire una sanzione finanziaria in termini di criptovaluta BingoCoin nel caso in cui l'avversario decida di non aprire il proprio commitment. In questo modo rispettiamo anche la proprietà **I4**. Si fa notare che in generale un avversario che abortisca in altre fasi del protocollo è poco rilevante nell'analisi, abortire in fase di invio del commitment non genera alcun tipo di vantaggio, per lo stesso motivo lo smart contract non prevede una funzionalità per permettere ad un giocatore di inviare il proprio commitment in caso di comportamenti disonesti, proprio perché essere disonesti in questa fase non provoca vantaggi. Per semplicità si potrebbe definire che chi non invia il commitment viene considerato AFK e cacciato dalla sala virtuale così da far continuare il protocollo. Inoltre, la firma è effettuata solo sul commitment, nel caso di funzionalità senza blockchain, poiché firmare l'apertura non genera alcun tipo di sicurezza in più, un giocatore non può contestare di aver aperto in quel modo, perché sarebbe stato l'unico modo di aprire quel determinato commitment.
Si fa notare che non è previso che il server firmi le aperture dei giocatori, questo poiché la ricezione delle aperture è attestata dall'invio della stringa random. Se il server volesse barare, ricevute le aperture dei giocatori, attestando

di non aver mai ricevuto i contributi potrebbe farlo, ma ciò è mitigato dall'introduzione della blockchain e non dalla firma del server sulle aperture, poiché in quello step del protocollo la ricezione è confermata dalla stringa random inviata dal server.

Il protocollo è stato progettato in maniera efficiente, infatti prevede pochi calcoli di funzioni hash SHA256 che sono, di natura, molto efficienti. Inoltre, gli unici partecipanti attivi del protocollo sono i giocatori ed il server, quindi nessuna parte terza viene inclusa. Infine, in caso di adozione della blockchain, viene penalizzata l'efficienza, soprattutto nel caso in cui qualche partecipante si comporti in maniera fraudolenta, a fronte di una maggiore trasparenza del protocollo e la migliore attenuazione del giocatore che abortisce, inoltre la blockchain permette di gestire automaticamente frodi senza ricorrere all'intervento di un'autorità giudiziaria.



Analisi della seconda funzionalità (Green Pass 2.0)

Nella seconda funzionalità viene stabilito innanzitutto il formato del Green Pass 2.0: un Merkle Tree, con all'interno le informazioni riguardanti il vecchio Green Pass e alcune informazioni extra tra cui PK ,scadenza e id univoco del giocatore. La scelta di usare tale struttura discende dalla necessità di garantire la proprietà **C5**; infatti, chiunque voglia inviare un'informazione univoca appartenente all'albero binario può semplicemente inviare la foglia corrispondente a tale informazione senza rivelare nulla sulle altre foglie dell'albero, se non il percorso per risalirlo, e quindi gli SHA256 dei fratelli, o dei nodi intermedi per arrivare alla radice. Si fa notare che questo è valido per il Green Pass 2.0, non semplicemente perché è un Merkle Tree ma perché esso ha il ruolo di vector commitment, in cui le foglie non sono hashate in chiaro, ma viene aggiunto un ulteriore layer di sicurezza che è quello del commitment dell'informazione, garantendo copertura selettiva, nel nostro caso un doppio strato di cui la motivazione verrà enunciata in seguito: la scelta di effettuare un doppio commitment deriva dal soddisfare la proprietà **C4**, essa infatti prevede che chiunque voglia autenticarsi o accedere ad un servizio deve solo dimostrare di possedere alcune informazioni, non rivelando nulla in chiaro, se non le informazioni fondamentali per l'accesso ovvero PK, scadenza del documento e identificativo univoco per legare il green pass al profilo conto gioco del giocatore e verificarne la proprietà. Per raggiungere questa caratteristica è stato necessario introdurre un doppio commitment, quello più interno nasconde le informazioni in chiaro, quello più esterno nasconde l'eventuale presenza di marker. L'applicazione riuscirà ad aprire solo il commitment esterno e verificare la presenza o meno di un campo, come il vaccino, senza avere informazioni sul tipo di vaccino specifico. Non mettere questo strato esterno significherebbe che chiunque può verificare se in una certa foglia è presente il campo marker o un valore vero e proprio. Ovviamente questo deve essere effettuabile dall'applicazione che deve verificare quei dati, ma solo nel caso in cui quei dati servano veramente all'applicazione. Ciò significa che se non ci fosse il commitment più esterno, e l'applicazione ricevesse come hash di un fratello dell'informazione richiesta, lo sha del valore marker, può carpire informazioni ulteriori sul green pass dell'interessato, un esempio è riportato di seguito:

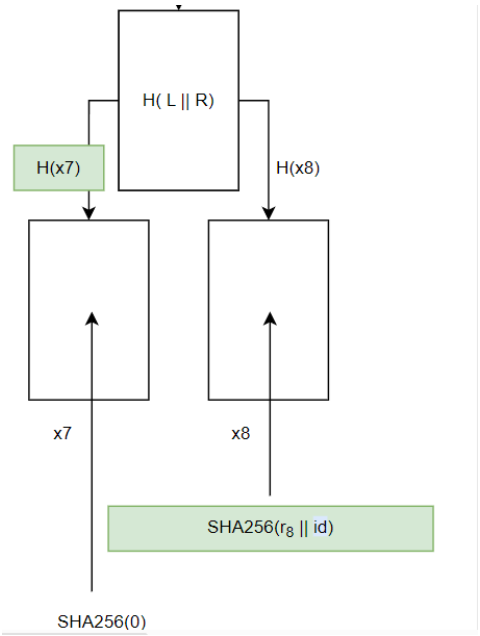


Figure 4: Esempio per dimostrare l'esigenza di un doppio commitment

Se dovessimo inviare solamente l'id all'applicazione, e di conseguenza l'hash della foglia x7, in questo modo l'applicazione, verificando se H(x7) == SHA256(SHA256(0)) può ricavare informazioni sulla presenza di quel campo nonostante non ne avrebbe avuto il diritto. Ragionamento inverso avviene se il commitment viene effettuato anche del valore 0, perché in quel caso l'applicazione, per poter verificare se si tratta di un commitment di 0 o di un valore generico del campo dovrebbe richiedere l'apertura del commitment e visualizzare l'informazione in chiaro. Grazie a questo meccanismo rispettiamo anche la proprietà **C2** poiché oltre a preservare i dati inviati all'applicazione, preserviamo anche gli altri dati che non sono esplicitamente inviati e non devono essere divulgati. Si fa notare che la scelta dell'identificativo univoco come contatore è stata preferita ad un identificativo già in vigore come la tessera sanitaria, per preservare i dati personali del possessore, come data di nascita o luogo di nascita.

Inoltre, viene definita la modalità di autenticazione delle informazioni, ovvero tramite firma dell'autorità sanitaria sulla radice del Merkle Tree. Grazie alla firma apposta dal Ministero della Salute, rispettiamo le proprietà **I7** e **I8**. Rispettivamente una modifica alle informazioni presente nel Green Pass 2.0 implicherebbe la modifica dei bit di una qualsiasi foglia, che comporterebbe la modifica della radice e conseguente rifiuto da parte delle applicazioni della firma effettuata dal Ministero della Salute che sarebbe in questo modo invalidata. Ciò è vero poiché il Merkle Tree, con la costruzione tramite SHA256, è anch'esso una funzione collision resistent poiché trovare una foglia che restituisca la stessa radice nonostante sia diversa da quella precedente implicherebbe una collisione alla funzione SHA256 usata dal Merkle Tree stesso. Inoltre, grazie alla stessa firma, e al suddetto meccanismo, chiunque può verificare la validità delle foglie inviate da un giocatore ricostruendo l'albero e verificando che la firma dell'autorità sanitaria sulla radice, calcolata a partire dalle foglie, sia valida, inoltre grazie all'ordine univoco è verificabile che quella foglia inviata è proprio la foglia dell'informazione richiesta, altrimenti, posizionata in un ordine diverso nell'albero, provocherebbe un cambio della radice.

Il possessore alteratore, infatti, è in forte difficoltà, come specificato in precedenza, non è in grado di alterare alcuna informazione del proprio Green Pass 2.0, poiché una singola modifica porterebbe al cambiamento radicale della radice e ad una firma invalida dell'autorità sanitaria.

Si fa notare che la presenza di informazioni come PK e identificativo univoco, sono necessarie per garantire la proprietà **I9**; infatti, un giocatore che intende identificarsi ad una piattaforma tramite Green Pass 2.0 deve essere in grado di fornire prova della proprietà del documento. Per quanto riguarda l'autenticazione rispetto all'autorità sanitaria, nella richiesta di ricevere un altro Green Pass 2.0 con identificativo univoco precedente, dato che la richiesta và espressamente firmata dal richiedente, l'autorità può effettuare un controllo sulla firma per verificare se la PK del Green Pass 2.0 è la stessa della persona con cui sta comunicando confermandone l'identità. In questo caso è possibile firmare la richiesta, poiché essa avviene una volta tantum, un replay attack di questo tipo sarebbe inutile, poiché una volta che l'autorità ha ricevuto la richiesta e rilasciato il nuovo Green Pass, in caso di ulteriore richiesta semplicemente rifiuterà poiché è già stata rilasciata una versione del documento esatta. Nel caso dell'identificazione al server di gioco, ciò non è valido, poiché tra una comunicazione e l'altra TLS un replay attack è effettuabile, poiché le informazioni da inviare per l'accesso ai giochi sono coerenti tra una comunicazione e l'altra, salvo cambio direzione del garante, proprio per questo motivo sono stati introdotti dei random bits per effettuare il refreshing della sessione, quindi chi desidera accertare la proprietà del Green Pass 2.0 deve firmare questi random bits e l'applicazione verificherà ancora una volta che la firma sia in accordo alla PK del Green Pass 2.0. Si fa notare, che grazie alla presenza di TLS, rispettiamo la proprietà **C3**.

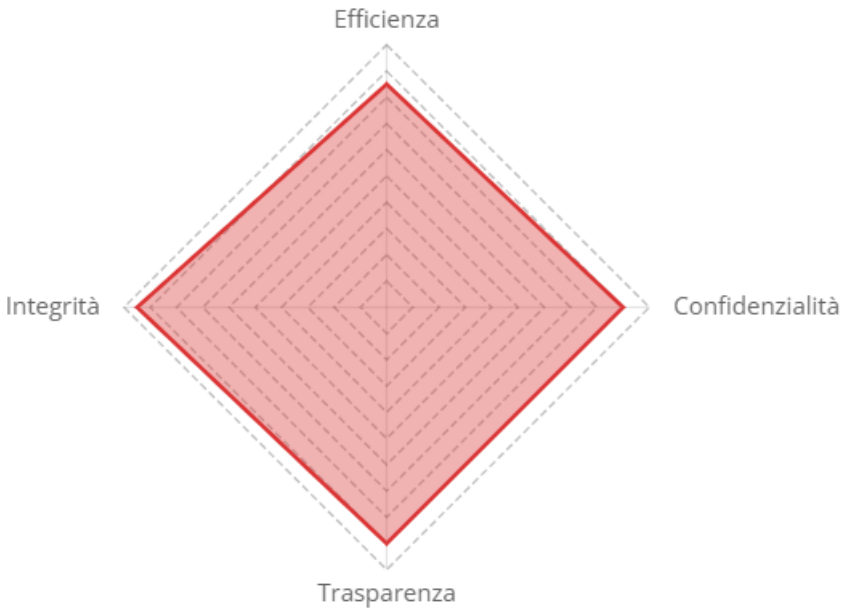
L'avversario ladro è neutralizzato poiché, grazie a TLS, non è in grado di reperire in alcun modo il Green Pass 2.0 altrui, neanche tentando di richiederlo all'autorità sanitaria, sia perché un replay attack provocherebbe il rifiuto dell'autorità sanitaria poiché è presente un documento già rilasciato, sia perché nell'eventualità che una richiesta non sia andata a buon fine, un replay attack su questo tipo di richiesta non modificherebbe l'esito della verifica, a meno che non subisca variazione la politica per il rilascio dei Green Pass 2.0 (a livello di informazioni da possedere), ma in quel caso, considerando questo tipo di avvenimenti legati a provvedimenti abbastanza lenti nel tempo, basterebbe firmare la richiesta || timestamp per rendere l'applicazione sensibile al momento della richiesta. Inoltre in caso di recupero di esso, ad esempio l'applicazione stessa, non potrebbe comunque identificarsi al posto del giocatore grazie alla necessità di firmare i random bits in accordo alla chiave.

Avversari ostici di questo protocollo sono le applicazioni inciucioni, esse infatti non hanno la possibilità, come spiegato in precedenza, di fare inferenza sul contenuto di determinati campi del possessore, ma hanno informazioni sulla presenza o meno di uno o più campi del Green Pass 2.0 dell'utente, se inviato esplicitamente. In questo modo mettendosi d'accordo potrebbero scambiarsi le varie informazioni ricavate, risalendo a quali informazioni un utente possiede, ad esempio se è vaccinato, non riuscendo comunque ad avere però l'informazione in chiaro, ad esempio il tipo di vaccino. Inoltre possono dimostrare effettivamente che un certo utente possiede o meno quell'informazione, poichè essendo presente la PK nell'albero, così come l'identificativo univoco, ed essendo inviata a qualsiasi applicazione per la verifica del possesso del Green Pass e dei dati, possono dimostrare

l'appartenenza di determinate informazioni ad un solo Green Pass 2.0 di un determinato utente. Di conseguenza il nostro protocollo non li neutralizza completamente ma ne limita molto la sfera di azione.

Infine, dato che il formato del Green Pass è pubblicamente noto, così come i vari meccanismi di login nelle applicazioni, e data la figura del garante che definisce in modo unico e imprescindibile quali dati devono essere inviati, riusciamo a sconfiggere quegli avversari ostici come i no-fox che minano all'innovazione facendo leva sulla poca trasparenza di quest'ultima. Nel nostro protocollo, infatti la trasparenza viene privilegiata, infatti rispettiamo anche le proprietà **T3** e **T4**, poiché sia il protocollo completo che i criteri per l'accesso ai servizi sono di dominio pubblico e pubblicamente verificabili.

Ultima analisi riguarda l'efficienza del protocollo, peccando leggermente nella scelta del Merkle Tree nell'efficienza spaziale si raggiunge comunque un buon trade off tra efficienza e sicurezza del protocollo; infatti, il protocollo prevede delle chiamate a SHA256 e meccanismi di comunicazione tramite TLS. Inoltre l'autorità sanitaria viene interpellata solamente per i compiti a lei assegnati, quindi il mantenimento dei Green Pass 2.0.



WP4

Per la realizzazione delle varie funzionalità abbiamo usato un approccio basato sulla bash, creando una serie di script bash che verranno nel seguito analizzati. Si fa notare che nel caso di avvio dei vari file bash, per riprovare l'esperimento si devono eliminare tutti i file intermedi non presenti nella consegna finale. Inoltre si fa notare che le comunicazioni TLS definite nel WP4 tra client e server sono state simulate grazie all'utilizzo dei comandi s_server e s_client. Per facilitare la comunicazione sono stati creati due file bash per riuscire ad inviare un file dal client al server e viceversa:

daClientAServer.sh ***daServerAClient.sh***

Nel quale vengono usati i seguenti comandi openssl:

- openssl s_server -port 8444 -key wwwkey.pem -cert wwwcert.pem -CAfile cacert.pem -ciphersuites 2>&1 | tee temp.bin & cat "$2" | openssl s_client -connect localhost:8444 -crlf -ciphersuites, per l'invio del generico file "$2" dal client al server che conserverà il risultato nel file temp.bin che verrà opportunatamente gestito, ripulendolo dalla comunicazione TLS e recuperando solo il valore del file inviato
- cat "$1" | openssl s_server -port 8444 -key wwwkey.pem -cert wwwcert.pem -CAfile cacert.pem -ciphersuites & openssl s_client -connect localhost:8444 -crlf -ciphersuites > temp.bin, per la comunicazione inversa, dal server verso il client, anche in questo caso il file temp.bin verrà gestito opportunatamente

Si fa notare inoltre, che per la seconda funzionalità è stato previsto il rilascio di due certificati diversi, per quanto riguarda l'autorità sanitaria e il server web, ma comunque entrambi con dominio web localhost. In questo caso i due file sopra citati prevedono la parametrizzazione della chiave e del certificato del server a seconda del tipo di comunicazione che si sta effettuando. Inoltre, si fa notare, che per la prima funzionalità la ricezione dei messaggi dal server è effettuata esclusivamente dal player1 per evitare un enorme quantità di file generati, inoltre il formato dei file non è sempre conservato come quello di partenza, ovvero spesso i file binari sono gestiti come file di testo e viceversa, per facilitare il parsing delle informazioni tra le comunicazioni TLS. Infine, dato che la prima funzionalità è temporalmente successiva alla seconda, si è assunto che il server abbia già le chiavi pubbliche dei vari giocatori ricavate dal loro accesso al sito web.

Prima funzionalità (generazione continua di stringhe casuali).

Per la prima funzionalità abbiamo considerato la presenza di 4 players e un banco. Sono stati realizzati i seguenti script bash:

creazioneCAeCertificatoServer.sh

Nel seguente file gestiamo la creazione della Certification Authority, la sua chiave pubblica e privata, così come il suo certificato autogenerato, per la creazione della chiave privata si è utilizzata la curva prime256v1. In seguito, si crea un server, definendo chiave pubblica e privata e facendo richiesta alla CA di un certificato digitale, la cui configurazione è specificata nel file ***myconfig.cfg***, e viene rilasciato specificando le estensioni del file ***extensions.cnf.*** Per completezza i due file sono forniti nella consegna finale. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl ecparam -name prime256v1 -out prime256v1.pem, per generare i parametri della curva ellittica

- openssl genpkey -paramfile prime256v1.pem -out cakey.pem, per generare la chiave privata della CA a partire dal file dei parametri della curva ellittica
- openssl pkey -in cakey.pem -pubout -out capub_key.pem, per generare la chiave pubblica a partire dalla chiave privata della CA
- openssl req -new -x509 -days 365 -key cakey.pem -out cacert.pem -config /etc/ssl/openssl.cnf, per richiedere un certificate autofirmato nel formato x509 (si fa notare che il percorso del file di configurazione è coerente con la posizione del file nel nostro file system)
- openssl genpkey -paramfile prime256v1.pem -out wwwkey.pem, per generare la chiave privata del server
- openssl pkey -in wwwkey.pem -pubout -out wwwpub_key.pem, per generare la chiave pubblica a partire dalla chiave privata del server
- openssl req -new -key wwwkey.pem -out wwwrequest.pem -config myconfig.cfg, per richiedere il certificate digitale con configurazione specificata
- openssl ca -in wwwrequest.pem -out wwwcert.pem -policy policy_anything -config /etc/ssl/openssl.cnf -extfile extensions.cnf, per rilasciare il certificato digitale richiesto dal server con le estensioni specificate

creazioneChiaviGiocatoriBanco.sh

In questo file, sempre partendo dalla curva prime256v1 sono generate le chiavi private e pubbliche dei 4 giocatori e del banco. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl genpkey -paramfile prime256v1.pem -out Playerxprivatekey.pem, per generare la chiave private del generico giocatore x
- openssl pkey -in Playerxprivatekey.pem -pubout -out Playerxpublickey.pem, per generare la chiave pubblica del generico giocatore x a partire dalla chiave privata
- openssl genpkey -paramfile prime256v1.pem -out Bancoprivatekey.pem, per generare la chiave private del banco
- openssl pkey -in Bancoprivatekey.pem -pubout -out Bancopublickey.pem, per generare la chiave pubblica del banco a partire dalla chiave privata

invioPKDalServer.sh

Il server, con questo file, invia le PK dei giocatori. Per semplicità abbiamo assunto che l'ordine lessicografico nell'esempio implementativo corrisponda con quello dei numeri naturali, di conseguenza il contributo del player 1 sarà concatenato per primo, e quello del player 4 per penultimo, prima di quello del banco.

generazioneContributi.sh

Nel seguente file sono stati generati i contributi casuali dei giocatori e del banco, così come la randomness per effettuare il commitment del contributo, è stato effettuato anche quest'ultimo concatenando il contenuto dei due file e richiamando la funzione SHA256. Inoltre, è stata effettuata la firma sui commitment. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl rand -out ContributoPlayerx.bin 32, per generare il contributo del generico giocatore x
- openssl rand -out RandomnessPlayerx.bin 32, per generare la randomness utilizzata nel commitment del contributo di ogni giocatore x
- openssl rand -out ContributoBanco.bin 32, per generare il contributo del banco
- openssl rand -out RandomnessBanco.bin 32, per generare la randomness utilizzata nel commitment del contributo del banco
- openssl dgst -sha256 concatenatedPlayerx.bin > CommitmentPlayerx.bin, per generare il commitment del contributo del generico giocatore x, considerando la concatenazione dei due file precedentemente creati
- openssl dgst -sha256 concatenatedBanco.bin > CommitmentBanco.bin, per generare il commitment del contributo del banco, considerando la concatenazione dei due file precedentemente creati
- openssl dgst -sign Playerxprivatekey.pem -out signatureCommitmentx.bin CommitmentPlayerx.bin, per firmare il commitment da parte di ciascun giocatore x con la sua chiave privata
- openssl dgst -sign Bancoprivatekey.pem -out signatureCommitmentBanco.bin CommitmentBanco.bin, per firmare il commitment del banco con la sua chiave privata

invioCommitmentAlServer.sh

Nel seguente file, i commitment precedentemente creati, così come le firme, vengono inviate da ogni client al server sfruttando il file di bash precedentemente descritto **daClientAServer.sh**

checkPlayerCommitments.sh

In questo file, il server, ricevuti i commitment dei giocatori e le relative firme, verifica che queste siano valide. Il comando openssl sfruttato è:

- openssl dgst -verify Playerxpublickey.pem -signature signatureCommitmentxOnServer.bin CommitmentPlayerxOnServer.bin, per verificare che la firma del generico giocatore x del suo commitment è valida

invioCommitmentDalServer.sh

Il server, in questo file, invia tutti i commitment ricevuti al player1 (si ricorda la scelta di inviare soltanto ad un player per simulare la funzionalità stessa e per evitare un numero esagerato di file), e le loro firme, compreso il suo e lo firma. Per semplicità la firma qui è stata posta soltanto sul suo commitment e non sull'intera lista, per facilitare il parsing dei file in ricezione dal player 1.

checkAllCommitmentsByPlayer1.sh

Ricevuti tutti i commitment dal player1 esso effettua la verifica che essi siano validi in accordo alle firme ricevute, inoltre verifica la firma del banco sul proprio commitment. Dato che egli è il possessore del commitment1, verifica semplicemente che quello inviato dal banco sia in accordo con quello che lui stesso ha inviato precedentemente. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl dgst -verify PlayerxpublickeyDaServerpem -signature signatureCommitmentxDaServer.bin CommitmentPlayerxDaServer.bin, per verificare che la firma del generico giocatore x del suo commitment è valida
- openssl dgst -verify BancopublickeyDaServer.pem -signature signatureCommitmentBancoDaServer.bin CommitmentBancoDaServer.bin, per verificare che la firma del banco del suo commitment è valida

invioApertureAlServer.sh

Nel seguente file viene aperto il commitment da parte di ogni giocatore inviando al server il contributo e la randomness dei contributi generati in precedenza.

checkPlayersOpenings.sh

In questo file il banco, ricevute le aperture dei giocatori, concatena contributi e randomness e verifica che il commitment sia valido richiamando SHA256 sulla concatenazione di questi due file e verificando l'uguaglianza con il commitment inviato in precedenza. Il comando openssl sfruttato è:

- openssl dgst -sha256 concatenatedPlayerx.bin > OpenedCommitmentPlayerxOnServer.bin, per salvare il commitment del contributo con la randomness inviata e successivamente verificare l'uguaglianza con quello precedentemente inviato, tramite cmp

invioApertureDalServer.sh

In questo file, il server rimanda tutte le aperture dei commitment ricevute ai vari giocatori, in più invia anche la sua apertura. Infine, calcola anche la stringa random generata, concatenando i contributi nell'ordine stabilito, tramite chiamata a SHA256 con input la concatenazione dei file e la invia ai giocatori. Il comando openssl sfruttato è:

- cat ContributoPlayer1OnServer.bin ContributoPlayer2OnServer.bin ContributoPlayer3OnServer.bin ContributoPlayer4OnServer.bin ContributoBanco.bin | openssl dgst -sha256 > StringaRandomCalcolataDalServer.bin, per calcolare la stringa random in accordo al protocollo

checkAllOpeningByPlayer1.sh

Nel seguente file il player1 verifica le aperture degli altri giocatori e del banco, in più concatena il suo contributo, nella posizione a lui assegnata dall'ordinamento, con quelli degli altri partecipanti e produce la stringa random tramite funzione SHA256 della concatenazione di tutti i file ottenendo così la stringa

random calcolata in accordo al protocollo. A questo punto può comparare la sua con quella inviata dal banco e verificare siano uguali. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl dgst -sha256 concatenatedPlayerx.bin > OpenedCommitmentPlayerxOnPlayer1.bin, per salvare il commitment del contributo con la randomness ricevuta dal server e successivamente verificare l'uguaglianza con quello precedentemente inviato dal server, tramite cmp
- openssl dgst -sha256 concatenatedBanco.bin > OpenedCommitmentBancoOnPlayer1.bin, per salvare il commitment del banco con la randomness ricevuta da esso e successivamente verificare l'uguaglianza con quello precedentemente inviato, tramite cmp
- cat ContributoPlayer1DaServer.bin ContributoPlayer2DaServer.bin ContributoPlayer3DaServer.bin ContributoPlayer4DaServer.bin ContributoBancoDaServer.bin | openssl dgst -sha256 > StringaRandomCalcolataDaPlayer1.bin, per calcolare la stringa random e successivamente confrontarla con quella inviata dal banco

Questo termina la discussione sull'implementazione della prima funzionalità senza blockchain, per discutere la scelta implementativa verranno evidenziati i nuovi file bash nel dettaglio, richiamando eventuali funzionalità in comune con l'implementazione senza blockchain. Inizialmente verrà sempre generata la CA e il server, così come le chiavi dei giocatori e del banco.

invioCryptoAlServer.sh

Nel seguente file ogni giocatore genera una richiesta di partecipare alla partita, includendo un quantitativo in criptovaluta e firmando la suddetta richiesta, inviando tutto al server. Il comando openssl sfruttato è:

- openssl dgst -sign Playerxprivatekey.pem -out signatureOfRequestPlayerx.bin requestToPartecipatePlayerx.txt, per firmare la richiesta di partecipare alla partita contenente anche al criptovaluta usata come garanzia

checkPlayerRequests.sh

Il server, ricevute le richieste, controlla che la firma su di esse sia valide, prima di inoltrarle sulla blockchain, inoltre controlla che il quantitativo in criptovaluta sia sufficiente. Il comando openssl sfruttato è:

- openssl dgst -verify Playerxpublickey.pem -signature signatureOfRequestPlayerxOnServer.bin requestToPartecipatePlayerxOnServer.txt, per verificare che la firma del giocatore sulla richiesta sia valida

FirstTransactionBlockchain.sh

Nel seguente file bash è gestita la prima transazione effettuata sulla blockchain, abbiamo anche simulato il comportamento dello smart contract, che verifica innanzitutto che la quantità di criptovaluta sia valida, ma anche che la firma della richiesta sia coerente con la PK del richiedente. Se tutto il blocco di transazioni è valido, viene aggiunto alla blockchain, altrimenti viene interamente scartato. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl dgst -verify Playerxpublickey.pem -signature signatureOfRequestPlayerxOnServer.bin requestToPartecipatePlayerxOnServer.txt, per verificare che la firma del giocatore sulla richiesta sia valida
- openssl dgst -verify Bancopublickey.pem -signature signatureOfRequestBanco.bin requestToPartecipateBanco.txt, per verificare che la firma del banco sulla richiesta sia valida

A questo punto i giocatori genereranno i contributi e invieranno i commitment che verranno verificati dal server, esattamente come nella funzionalità precedente senza blockchain.

SecondTransactionBlockchain.sh

Verificati i commitment dal server, egli pubblica, grazie a questo file, sulla blockchain i commitment dei vari giocatori. Anche in questo caso simuliamo lo smart contract che andrà a verificare che la firma del giocatore sul commitment è valida, inoltre verificherà anche che il giocatore sia presente tra quelli partecipanti alla corrente partita. Se qualsiasi di questi controlli non va a buon fine il blocco viene interamente scartato, altrimenti viene aggiunto alla blockchain. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl dgst -verify Bancopublickey.pem -signature signatureCommitmentBanco.bin CommitmentBanco.bin
- openssl dgst -verify Playerxpublickey.pem -signature signatureCommitmentxOnServer.bin CommitmentPlayerxOnServer.bin, per verificare che la firma del giocatore sulla richiesta sia valida

A questo punto i giocatori inviano le aperture al banco che verifica siano corrette, in accordo alla funzionalità precedente. Inoltre, c'è bisogno di firmare le aperture in questo caso, di conseguenza:

firmaApertureAlServer.sh

Nel seguente file ogni giocatore firma la propria apertura, quindi la concatenazione di contributo e randomness e le invia al server. Il comando openssl sfruttato è:

- openssl dgst -sign Playerxprivatekey.pem -out signatureOpeningx.bin concatenatedPlayerx.bin, per firmare la concatenazione di contributo e randomness del giocatore x

checkFirmeServer.sh

Prima di pubblicare sulla blockchain il server verifica che le firme siano corrette. Il comando openssl sfruttato è:

- openssl dgst -verify Playerxpublickey.pem -signature signatureOpeningxOnServer.bin concatenatedPlayerxOnServer.bin, per verificare che la firma del giocatore x sia valida

ServerFirmaApertura.sh

Con il seguente file anche il server firma la propria apertura, prima di pubblicare il tutto sulla blockchain. Il comando openssl sfruttato è:

- openssl dgst -sign Bancoprivatekey.pem -out signatureOpeningBanco.bin concatenatedBanco.bin, per firmare la concatenazione di contributo e randomness del banco

ThirdTransactionBlockchain.sh

Verificate le aperture dal server, egli pubblica, tramite il seguente file bash, sulla blockchain i contributi dei giocatori, specificando anche la randomness. Lo smart contract controllerà che esista un commitment del giocatore precedentemente inviato e che sia correttamente stato aperto con il contributo e la randomness ricevute oltre a verificare la firma. Se i controlli vanno a buon fine aggiunge il blocco alla

blockchain, altrimenti viene scartato.. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl dgst -sha256 concatenatedPlayerx.bin > OpenedCommitmentPlayerxOnChain.bin, per calcolare il commitment da parte dello smart contract dei vari player, sulla base del contributo e della randomness per poi verificare l'uguaglianza con quelli pubblicati precedentemente sulla blockchain
- openssl dgst -sha256 concatenatedBanco.bin > OpenedCommitmentBancoOnChain.bin, per calcolare il commitment da parte dello smart contract del banco, sulla base del contributo e della randomness per poi verificare l'uguaglianza con quelli pubblicati precedentemente sulla blockchain
- openssl dgst -verify Playerxpublickey.pem -signature signatureOpeningxOnServer.bin concatenatedPlayerxOnServer.bin, per verificare che la firma del giocatore x sia valida
- openssl dgst -verify Bancopublickey.pem -signature signatureOpeningBanco.bin concatenatedBanco.bin, per verificare che la firma del banco sia valida

ApertureSingolaSuBlockchain.sh

Il seguente file è richiamato in caso di frode da parte di qualche giocatore che decidesse di abortire il processo non aprendo il commitment o del server che ricevute le aperture potrebbe decidere di non pubblicarle sulla blockchain. Nel seguente file ogni giocatore può pubblicare singolarmente sulla blockchain la propria apertura obbligando gli altri partecipanti a fare lo stesso, anche in questo caso la blockchain è simulata compiendo gli stessi compiti dello smart contract e verificando che esiste un commitment del giocatore precedentemente inviato e che sia correttamente stato aperto con il contributo e la randomness ricevute, oltre che la verifica sulla firma. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl dgst -sha256 concatenatedPlayerx.bin > OpenedCommitmentPlayerOnChain.bin, per calcolare il commitment da parte dello smart contract del giocatore x, sulla base del contributo e della randomness per poi verificare l'uguaglianza con quello pubblicato precedentemente sulla blockchain
- openssl dgst -verify Playerxpublickey.pem -signature signatureOpeningxOnServer.bin concatenatedPlayerxOnServer.bin, per verificare che la firma del giocatore x sia valida

Un duale di questo file è **AperturaSingolaSuBlockchainBanco.sh**, semplicemente richiamata dal banco, per semplicità nella gestione dei file bash.

Al termine di entrambe le versioni, con o senza frode, viene richiamato il file:

calculateRandomStringFromBlockchain.sh

Ogni partecipante richiama questo file bash per calcolare la stringa random generata dal protocollo ricavandola dai contributi pubblicati sulla blockchain.

Seconda funzionalità (Green pass 2.0 e autenticazione)

Per la seconda funzionalità abbiamo simulato la richiesta di un Green Pass 2.0 da parte di Mario e la ricezione di tale Green Pass dall'autorità sanitaria. Per semplicità il Green Pass 2.0 è rappresentato da vari file che rappresentano foglie e nodi e la radice, in più un file in cui è presente la firma dell'autorità sanitaria.

creazioneCAeCertificatoServer.sh

Per la cui descrizione si rimanda alla funzionalità precedente.

creazioneAutoritàSanitariaServer.sh

Nel seguente file viene creato il server dell'autorità sanitaria da cui rilascerà i Green Pass a chi ne fa richiesta. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl genpkey -paramfile prime256v1.pem -out askey.pem, generazione della chiave privata dell'autorità sanitaria
- openssl pkey -in askey.pem -pubout -out aspub_key.pem, generazione della chiave pubblica dell'autorità sanitaria a partire dalla chiave privata
- openssl req -new -key askey.pem -out asrequest.pem -config myconfigas.cfg, richiesta del certificate alla CA

- openssl ca -in asrequest.pem -out ascert.pem -policy policy_anything -config /etc/ssl/openssl.cnf -extfile extensions.cnf, rilascio del certificate

generazioneInfoGreenPass2.0.sh

Il seguente file stabilisce quelli che sono i dati dell'utente Mario, oltre a creare la sua chiave pubblica e privata, in questo modo è possibile modificare i dati e controllare se si viene accettati o meno dall'applicazione. Nell'esempio in questione per poter partecipare ai servizi della sala virtuale si deve possedere un vaccino, qualsiasi, e non si deve aver commesso alcun tipo di reato. Quindi la richiesta viene formulata con i dati di Mario, che la firmerà e invierà al server dell'autorità sanitaria. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl genpkey -paramfile prime256v1.pem -out Marioprivatekey.pem, per generare la chiave privata di Mario
- openssl pkey -in Marioprivatekey.pem -pubout -out Mariopublickey.pem, per generare la sua chiave pubblica a partire da quella privata
- openssl dgst -sign Marioprivatekey.pem -out signatureRequestOfMario.bin infoGreenPass2.0.txt, per firmare la richiesta di rilascio Green Pass 2.0

controlloERilascioGP2.0.sh

Nel seguente file l'autorità sanitaria verifica innanzitutto che la firma della richiesta sia valida, dopo di che', per semplicità accetta a prescindere i dati presenti, nel mondo reale ovviamente ciò prevederebbe il poter controllare la validità dei dati, e costruisce il Green Pass 2.0 creando le foglie del Merkle Tree fino alla radice. In seguito firma la radice e invia l'albero completo e la firma a Mario. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- openssl dgst -verify checktemp.pem -signature signatureRequestByMario.bin requestByMario.txt, per verificare la firma sulla richiesta di Mario
- openssl rand -out r.bin 32, per la generazione della randomness per il commitment interno dell'informazione
- openssl rand -out s.bin 32, per la generazione della randomness per il commitment esterno

- cat r.bin Info.txt | openssl dgst -sha256 > SHA256rInfo.bin, per creare il commitment più interno
- cat SHA256rInfo.bin s.bin | openssl dgst -sha256 > ValoreSHA.bin, per creare il commitment più esterno
- cat Info.txt Info2.txt | openssl dgst -sha256 > Fogliaxy.txt, per la creazione del generico nodo a partire dalle foglie x e y, per semplicità di notazione, anche i nodi intermedi sono stati chiamati "Fogliaxy" dove x e y rappresentano i figli nell'albero, valido tranne per la radice
- openssl dgst -sign askey.pem -out signatureRadiceMario.bin Radice.txt, per firmare la radice con la chiave privata dell'autorità sanitaria

invioRandomBitsDalServer.sh

Con il seguente file, il server di gioco, crea dei random bits e li invia al giocatore per verificare la proprietà del documento. Il comando openssl sfruttato è:

- openssl rand -out RandomBits.bin 32, per la generazione dei bit casuali

invioIdentificazioneAlServer.sh

Con il seguente file, invece, Mario invia le informazioni richieste per l'accesso al sito, quali foglie riguardanti PK, scadenza e identificativo univoco e aperture dei relativi commitment, oltre che il percorso per risalire l'albero. Inoltre firma i random bits e invia tutto al server. Il comando openssl sfruttato è:

- openssl dgst -sign Marioprivatekey.pem -out signatureRandomBits.bin RandomBitsMario.bin, per la firma dei random bits

checkIdentificazioneAndLogin.sh

In questo file il server di gioco ricostruisce il Merkle Tree verificando che la radice non sia presente nella lista delle revoche, inoltre controlla che la firma dell'autorità sia valida. Inoltre apre i commitment,

sia esterni che interni e ottiene l'accesso a PK, scadenza e identificativo univoco. Dopo aver controllato la validità di tali commitment verifica che la firma dei random bits sia valida in accordo alla PK presente, che il Green Pass 2.0 non sia scaduto e se le verifiche vanno a buon fine permette l'accesso al profilo di gioco legato all'identificativo univoco presente nel documento. I comandi openssl sfruttati sono racchiusi nella seguente lista di comandi:

- cat Valorex.txt Valorey.txt | openssl dgst -sha256 > Valorexy.txt, per ricostruire il percorso nell'albero e ottenere ValoreRadice.txt
- openssl dgst -verify aspub_key.pem -signature signatureRadiceMarioOnServer.bin ValoreRadice.txt, verifica che la firma inviata da Mario sia valida per la radice calcolata dal server
- cat rxOnServer.txt InfoxOnServer.txt | openssl dgst -sha256 > SHA256rInfo.bin & cat SHA256rInfo.bin sxOnServer.txt | openssl dgst -sha256 > ValoreSHAx.bin, per verificare la corretta apertura dei commitment
- openssl dgst -verify Info6OnServer.txt -signature signatureRandomBitsOnServer.bin RandomBits.bin, per verificare che la firma dei random bits sia valida utilizzando la PK del Green Pass 2.0 (nel nostro esempio si trova nella Foglia 6 dell'albero)

invioInfoRequestByServer.sh

File che utilizza il server per richiedere le informazioni ulteriori all'utente, in linea con le direttive del garante. Nel nostro esempio il giocatore riceve la richiesta e la soddisfa, ma nella realtà verificherebbe innanzitutto che la richiesta sia coerente con le direttive del garante.

invioUlterioriInformazioniPerGiocare.sh

Similmente a quanto avviene per l'identificazione, Mario invia le foglie per partecipare al gioco, inviando però solo l'apertura più esterna e non quella interna.

checkUlterioriInformazioni.sh

Con il seguente file l'applicazione ricostruisce l'albero, verifica che la radice sia coerente con quella calcolata nello step precedente e apre i commitment esterni verificando la presenza del campo Vaccino e l'assenza del campo reati. Se la verifica va a buon fine il giocatore ottiene l'accesso ai servizi.

- cat Valorex.txt Valorey.txt | openssl dgst -sha256 > Valorexy.txt, per ricostruire il percorso nell'albero e ottenere ValoreRadice2.txt, da confrontare con ValoreRadice1.txt calcolata al punto precedente
- cat SHA256rxInfoxOnServer.txt sxOnServer.txt | openssl dgst -sha256 > ValoreSHAxOnServer.bin ,da confrontare con il valore nella foglia per accertare la validità del commitment

Questo termina la discussione sull'implementazione della seconda funzionalità.