Project Work

Cyber-Security, A.A. 2020/21, Team 10

Paolo Mansi	0622701542	p.mansi5@studenti.unisa.it	WP1
Alessio Pepe	0622701463	a.pepe108@studenti.unisa.it	WP2
Teresa Tortorella	0622701507	t.tortorella3@studenti.unisa.it	WP3
Luigi Maiese	0622900080	l.maiese1@studenti.unisa.it	WP4

Changelog v.1.4.1

1. È stato rimosso un refuso delle versioni precedenti alla fine del paragrafo 2.2 riguardante una richiesta di chiavi multiple che in realtà non viene effettuata e si confondeva con quella del paragrafo 2.3.1.
2. Al paragrafo 2.5 passo 2 è stato aggiunto un controllo dell'HA sulla presenza dell'ID del richiedente del tampone nella lista degli ID a rischio.
3. Corretti errori grammaticali o di forma che peggioravano la leggibilità ma lasciano il senso inalterato.
4. Sono stati aggiunti WP4 e conclusioni.

1. WP1 – Modello

Negli ultimi cento anni si sono manifestate continuamente nuove epidemie e persino pandemie. Diverse ondate di malattie infettive hanno comprovato quale rischio queste rappresentino per la salute. Le patologie trasmissibili sono state recepite dai mass-media anche nel nuovo millennio: tra il 2002 e il 2004, ad esempio, si è propagato il coronavirus della SARS, fino ai tempi attuali con la diffusione del COVID-19. Ognuno ha il dovere di prepararsi per essere in grado di proteggere sé stesso e gli altri e vivere la quotidianità in caso di pandemia.

Si consideri l'eventualità di una possibile esplosione di un'epidemia difficilmente controllabile. Lo studio della comparsa dell'epidemia si concentra sulla rapida identificazione delle fonti di infezione affinché possano essere adottati provvedimenti immediati. In tal caso vengono prontamente contattate, informate e trattate le persone che hanno avuto contatti con i contagiati. Nella situazione descritta, i provvedimenti impediscono la trasmissione e, di conseguenza, l'insorgenza di malattie. Di norma, dalle informazioni così ottenute vengono ricavate raccomandazioni e misure capaci di ridurre il numero di infezioni nel lungo periodo. La disinformazione provocata dai mass media nonché dai social network fungono da barriera alla distribuzione di applicazioni di questo tipo.

L'obiettivo principale è realizzare un sistema di tracciamento dei contatti funzionante e sicuro sotto le definizioni di confidenzialità, integrità e trasparenza e comunque efficiente secondo le assunzioni e le limitazioni descritte nel prosieguo di questo documento.

La soluzione, così come pensata, introduce delle figure che sono interessate a corrompere il sistema o ad ottenere informazioni riservate mediante questo. Tali figure verranno analizzate nella descrizione del threat model.

Le figure coinvolte sono le seguenti:

- Gli **Utenti** (U) del sistema (cittadini di Campanialandia) che scaricano l'applicazione e la utilizzano sul loro smartphone.
    - L'hardware degli utenti viene assunto privo di software malevolo, che va oltre gli scopi di questa trattazione.
    - Si assume l'utilizzo dell'applicazione di contact tracing (CT) sia facoltativo e che quindi gli utenti possano in ogni momento decidere di installarla o disinstallarla, o comunque di disattivarla in ogni momento.
- L'**Health Authority** (HA) che esegue i tamponi e certifica la positività di essi.
    - Assumiamo che HA effettui correttamente i tamponi e dia sempre informazioni precise sugli esiti.
    - L'HA possiede un sistema già digitalizzato e distribuito, dove la figura di contatto tra utenti ed HA sono i **Medici** ($M_1, M_2, \dots, M_m$).
    - La popolazione di Campanialandia non ha particolari motivi per diffidare di questa istituzione.
- Il **Ministero della Digitalizzazione** (MD) fornisce agli utenti la possibilità di registrarsi al sistema e si assume che possieda o che possa installare dei server di grandi dimensioni (con una spesa non eccessiva) per memorizzare le registrazioni ed altre informazioni utili ai fini del tracciamento.
    - Il MD potrebbe in generale essere interessato a conoscere informazioni che gli utenti non vogliono condividere (es., posizione dei cittadini o contatti di questi).
    - Gli utenti potrebbero essere diffidenti rispetto all'MD.

- La **Polizia** (P) ha l'autorità di vigilare sugli spostamenti degli utenti del sistema certificati positivi, al fine che questi rispettino la quarantena imposta dall'HA, che va oltre gli scopi di questa trattazione.
- L'**Autorità Giudiziaria** (AG) che gestisce i casi di violazioni delle leggi imposte a Campanialandia per quanto riguarda le restrizioni imposte al singolo cittadino in caso di positività o di truffe ecc.

1.1. Completeness

Gli utenti $U_1, U_2, \dots, U_n$ possono scaricare e installare l'applicazione e inoltre sono sempre in grado di registrarsi ad essa. Gli utenti del sistema di CT riescono a costruire e mantenere correttamente una lista dei contatti avuti, per un tempo minimo di 15 minuti cumulati, durante il periodo interessato (in questo caso 20 giorni) e vengono correttamente notificati quando uno dei loro contatti stretti viene certificato positivo al tampone dall'HA. Inoltre, quando ricevono la notifica di contatto stretto, possono ricevere un tampone gratuito se richiesto entro 24h.

Quanto descritto può essere visualizzato graficamente in Figura 1.1.

1.2. Threat model

Si descrivono, di seguito, dei potenziali avversari che vorrebbero utilizzare in modo malevolo il sistema, le loro risorse e le motivazioni.

1.2.1. Il curioso $\mathcal{C}$

Questo avversario vuole estrarre informazioni dal sistema cercando di ledere la privacy degli utenti e di conoscere l'identità degli utenti positivi. Questo avversario include tutti i contatti di un positivo o comunque avversari interessati a rendere pubbliche queste informazioni, nonché un governo straniero, che può figurare nell'azienda "Statt Cas" o quello di Campanialandia stesso, figurante nel MD (n.b. solo l'HA deve poter conoscere l'identità degli utenti positivi). Consideriamo questo avversario, nel caso più forte, in possesso di scanner Bluetooth Low Energy (BLE) e Bluetooth (BT) passivi, dotato di telecamere, accesso alle informazioni sui pagamenti digitali, e qualsiasi strumento per tracciare un utente, oltre che software adeguato e risorse umane adibite allo scopo (nei limiti di un trade off spesa-risultato vantaggioso).

1.2.2. Lo spione $\mathcal{S}$

Questo avversario vorrebbe tracciare gli utenti cercando di scoprire con chi sono stati a contatto e le loro posizioni. Questo avversario potrebbe includere il governo stesso, figurante nel MD, o terze parti quali ad esempio un governo straniero (che potrebbe figurare in "Statt Cas"). Consideriamo questo avversario, nel caso più forte, in possesso di scanner BLE e BT passivi, dotato di telecamere, accesso alle informazioni sui pagamenti digitali, e qualsiasi strumento per tracciare un utente, oltre che software adeguato e risorse umane adibite allo scopo (nei limiti di un trade off spesa-risultato vantaggioso).

1.2.3. Il diffamatore $\mathcal{D}$

Questo avversario ha l'obiettivo di rovinare la reputazione del sistema di tracciamento (o in caso riesca, a rendere il sistema inutilizzabile) e potrebbe figurare in negazionisti, membri di un governo esterno, competitor, …. Si può classificare in diverse sotto-tipologie:

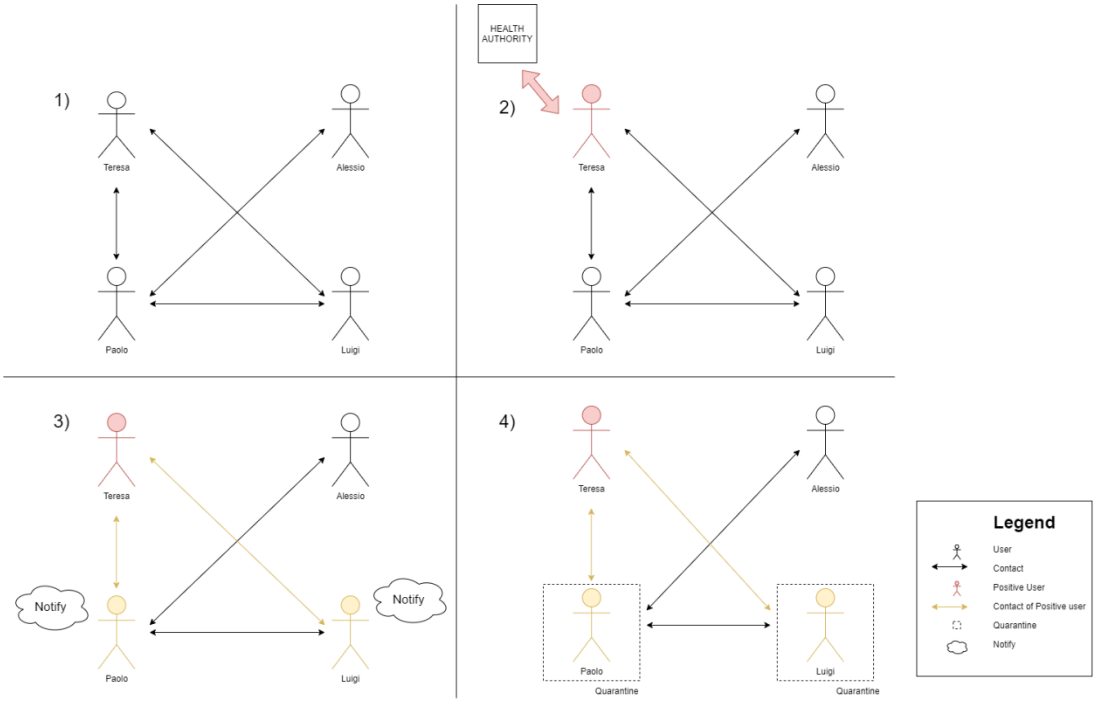

Figura 1.1: Rappresentazione grafica della proprietà di completeness.1) Teresa, Alessio, Paolo e Luigi sono quattro utenti del sistema di CT. I loro contatti sono rappresentati da una freccia bidirezionale nera. 2) Teresa risulta positiva e viene certificata dall'HA. 3) I contatti stretti di Teresa, evidenziati in giallo (Paolo e Luigi), vengono notificati. 4) I contatti stretti si isolano volontariamente e possono richiedere un tampone gratuito all'HA.

1. ***False Positive:*** Questo avversario vorrebbe fingere di essere positivo dopo aver raccolto una lista di contatti in un luogo affollato, o comunque strategico, possibilmente anche utilizzando hardware esterno come scanner BLE per ampliare il bacino di utenza;
2. ***Fake match:*** Questo avversario vuole approfittare della sua positività (risulta realmente positivo al tampone, certificato dall'HA) e dichiarare contatti non avvenuti realmente;
3. ***Sibling****:* Questo avversario cerca di utilizzare più identità oppure impersonare altri utenti, al fine di creare un numero ancora maggiore di false positive o comunicare la positività al posto di altri utenti. Consideriamo questo avversario, nel caso più forte, in possesso di scanner BLE e BT passivi;
4. ***Replay***: Questo avversario vorrebbe utilizzare dei contatti avvenuti realmente con utenti del sistema replicandoli e inviandoli su tutta Campanialandia, potendosi servire anche di più persone, di hardware esterno e di una distribuzione capillare sul territorio per generare falsi contatti tra persone. Si può classificare in
    a. *Attacco in tempo reale / Attacco in differita*: questi avversari vogliono ritrasmettere quanto visto dai normali utenti per creare falsi positivi o per accumulare essi stessi contatti, rispettivamente ri-condividendo nello stesso tempo e in un tempo successivo.
    b. *Attacco su larga scala*: questo avversario vuole ri-condividere informazioni degli utenti onesti, ma su posizioni geografiche abbastanza distanti.

    Consideriamo questo avversario in possesso di scanner BLE e BT attivi nel caso più forte.
5. ***False Negative:*** Questo avversario vuole impedire che, dopo che un utente del sistema venga certificato positivo dall'HA, questo possa notificare i suoi contatti stretti. Consideriamo questo avversario in grado di alterare connessioni TCP/IP.

1.2.4. Il riservato $\mathcal{R}$

Questo avversario è un utente del sistema che però vuole evitare di dichiarare la sua positività al resto degli utenti, nonostante essa sia stata certificata dall'HA.

1.2.5. L'avaro $\mathcal{A}$

Questo avversario vuole effettuare i tamponi a spese di Campanialandia o comunque creare disagi finanziari mediante la richiesta di tamponi gratuiti per cui non si hanno i requisiti. Potrebbe figurare in cittadini avari o anche in un governo/istituzione interessata a far spendere più del dovuto al Governo.

1.3. Integrità

In presenza di avversari il sistema deve garantire che:

I.1. Un utente o avversario non positivo non possa notificare la falsa positività ai suoi contatti;
I.2. Un utente non possa far arrivare notifiche ad un altro utente con cui non è stato realmente in contatto;
I.3. Nessun avversario può riuscire a fingersi un altro utente del sistema;
I.4. Un contatto stretto di un positivo debba sempre essere notificato;
I.5. Soltanto un utente che ha ricevuto una notifica può richiedere un tampone gratuito.

1.4. Confidenzialità

In presenza di avversari il sistema deve garantire che:

C.1. Nessun utente può conoscere i contatti altrui ma soltanto i propri;
C.2. In caso di ricezione di una notifica di contatto un utente non può avere più informazioni su chi sia il contatto stretto positivo di una scelta casuale tra i suoi contatti stretti (n.b. nell'ipotesi che nessun utente del sistema condivida volontariamente queste informazioni personalmente o sui social);
C.3. Nessuno, a meno dell'HA, può conoscere l'identità degli utenti positivi (rispettando le stesse assunzioni fatte in C.2.);
C.4. Un utente non deve essere tracciabile mediante l'utilizzo dell'applicazione.

2. WP2 - Progettazione

Il sistema proposto si articola in cinque fasi:

1. **Fase di inizializzazione (applicazione):** In questa fase, che temporalmente si trova a monte della distribuzione dell'applicazione, vengono generati i parametri per utilizzare un sistema di cifratura a chiave pubblica con certificati e chiavi su curve ellittiche (i cui parametri verranno scaricati dagli utenti mediante il download dell'app, quindi si trovano sicuramente all'interno dell'applicazione). Inoltre, vengono implementati i server necessari all'utilizzo dell'applicazione da parte del MD.
2. **Fase di registrazione (utente):** In questa fase, l'utente effettua la registrazione compilando un form e si iscrive all'applicazione. Per mezzo della sua identificazione, riceve un certificato digitale firmato dal MD.
3. **Fase dei contatti:** In questa fase vengono utilizzati degli ID di 16 byte inviati in advertising mode con BLE. I dispositivi scambiano gli id ed in alcune condizioni, in cui si rendono conto di aver accumulato il tempo richiesto dalle specifiche, convalidano il contatto.
4. **Fase di accertata positività:** Quando un utente risulta positivo al tampone, entra in questa fase dove deve autenticare la sua positività mediante una procedura con l'HA, dopodiché potrà comunicare i contatti in modo anonimo, la cui veridicità potrà essere verificata senza il rilascio di informazioni direttamente sensibili.
5. **Notifica:** Periodicamente gli utenti si collegano al server dell'MD e verificano se vi è una entry che contiene uno degli ID utilizzati da loro negli ultimi 20 giorni. In caso positivo, l'applicazione genera una notifica con la quale è possibile richiedere un tampone.

2.1. Fase di inizializzazione (applicazione)

In questa fase il MD genera i parametri con i quali gli utenti potranno registrarsi all'applicazione. In particolare, la cifratura vuole essere basata su chiave pubblica utilizzando curve ellittiche. Il ministero deve quindi definire i parametri

$$T = (p, a, b, G, n, h) \tag{2.1}$$

dove, in particolare, si vuole utilizzare la curva *secp256k1* (la stessa del Bitcoin) avente i seguenti parametri:

- $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- $a = 0$
- $b = 7$
- G = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
- n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
- $h = 1$

Il MD genera una coppia (SK_{MD}, PK_{MD}) e richiede un certificato ad un'entità di certificazione. Inoltre, il MD farà da autorità di certificazione per gli utenti, quindi necessita di un certificato adeguato.

Inoltre, l'MD deve implementare due server:

- un server utilizzato per effettuare dei commitment da parte dell'utente, che lo fa autenticandosi con certificato digitale e caricando il commit. Si stima che il volume di utilizzo massimo, in caso tutti i dispositivi di Campanialandia dovessero aderire, a quattro milioni di accessi giornalieri (uno per dispositivo). Per questo motivo il server deve essere sufficientemente efficiente. I commit più vecchi di 20 giorni potranno essere cancellati, ottenendo un'ipotesi sui volumi di 80

milioni di commit in totale immagazzinati sul server con 4 milioni di scritture giornaliere e un numero di accessi molto basso in quanto eseguiti unicamente dall'HA.

- Un server a cui gli utenti si collegano in modalità anonima e si autenticano attraverso un token fornito dall'HA, che certifica la positività al tampone di quell'utente. Questo server deve essere in grado di verificare i token e i contatti nelle modalità che verranno descritte nelle fasi successive. L'ipotesi sui volumi in scrittura sarebbe basata sul picco di contagi giornalieri (ipotizziamo 10000) in quanto questi utenti dovrebbero comunicare i loro contatti e anche 4 milioni di accessi in lettura (per questo, però, potrebbe essere necessario implementare un datacenter più sofisticato).

All'interno dell'applicazione verranno inclusi le informazioni sulla curva, i rispettivi generatori, i link al sistema informativo dell'HA per comunicare la positività e prenotare il tampone ed i riferimenti ai server a cui connettersi per ottenere le informazioni desiderate in ogni momento appena descritti.

2.2. Fase di registrazione (utente)

In questa fase gli utenti scaricano l'applicazione e vogliono autenticarsi al sistema.

1. L'utente genera, a partire dai generatori dei gruppi salvati nell'applicazione, una coppia
$$(SK_U, PK_U) \leftarrow \mathrm{Gen}(1^n) \tag{2.2}$$
che vorrà utilizzare per cifrare e firmare all'interno dell'applicazione.
2. L'utente si collega, utilizzando una connessione basata su TLS al server di registrazione del servizio di CT. L'utente dovrà compilare una serie di form ed eventualmente caricare documenti o utilizzare un'identità digitale in modo che il MD possa identificarlo. L'utente invierà anche la chiave pubblica PK_U appena generata.
3. Il MD potrà, con i suoi mezzi, verificare la veridicità di queste informazioni e, in caso di informazioni fasulle o alterate, allertare la P che potrà procedere con delle indagini per verificare quanto accaduto. Nel caso di informazioni correttamente verificate, l'MD firma il certificato digitale all'utente, rendendolo ufficialmente iscritto al sistema. Le informazioni legate al certificato saranno verificabili soltanto dall'MD senza necessità di condivisione da parte dell'utente. L'utente potrà comunque utilizzarlo per instaurare connessioni sicure con altri enti, come l'HA.

Allo scadere del certificato potrebbe essere richiesto di riconfermare l'identità.

2.3. Fase dei contatti

In questa fase vengono utilizzati degli ID di 16 byte inviati in advertising mode con BLE.

I dispositivi sono in grado di riconoscere dalla potenza del segnale ricevuto se un dispositivo si trova a due metri da esso o meno (a meno di un ragionevole errore, ma è di nostra convenienza ragionare per eccesso e non perdere potenziali contatti).

Si definisce inoltre un tempo T_B, scelto in fase di progettazione dell'applicazione, che indica la distanza temporale tra due advertisement BLE di uno stesso utente (si sta dicendo che si dovrà inviare, o che comunque ci si aspetta di ricevere da un altro utente, advertisement BLE con frequenza $f_B = 1/T_B$). Di conseguenza, essendo richiesto un contatto di almeno 15 minuti cumulati, si deciderà che questi sono avvenuti se e solo se l'utente ha ricevuto $15min/T_B$ pacchetti da un secondo utente (es., Teresa deciderà che Paolo ha avuto un contatto con lei di 15 minuti cumulati se, scelto $T_B = 1min$, avrà ricevuto da Paolo 15 pacchetti con il suo ID a una distanza di almeno T_B l'uno dall'altro). I pacchetti

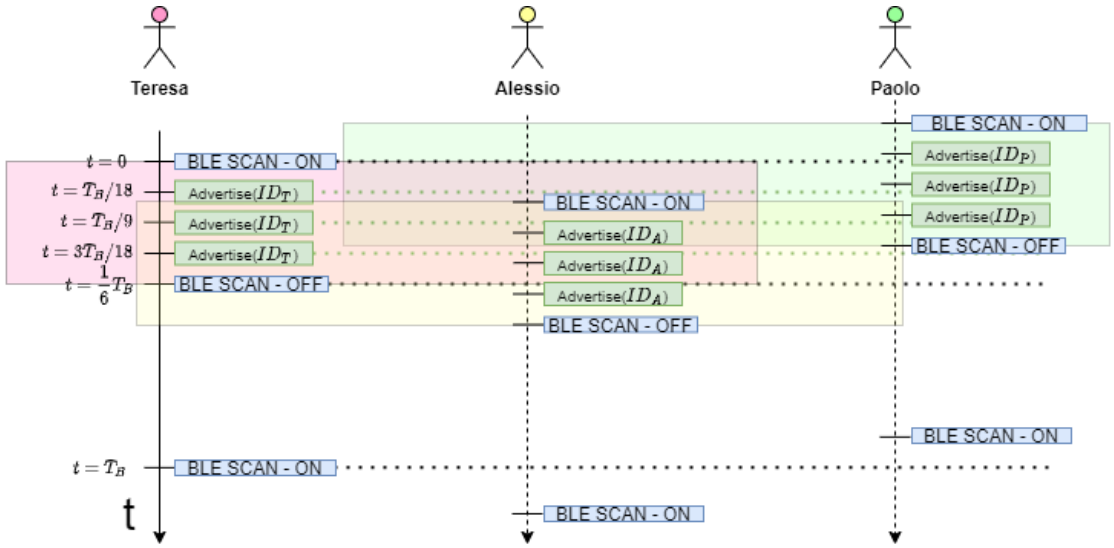

Figura 2.1 Una rappresentazione grafica dei tempi di scan e di invio dell'ID in Advertising BLE, mostrando il concetto di ragionevole margine di errore legato alla sincronizzazione tra dispositivi non omogenei.

dovranno essere distanziati l'uno dall'altro di almeno T_B (con un ragionevole margine di errore) ed infatti i dispositivi salveranno le informazioni necessarie per verificare queste tempistiche; pacchetti ricevuti a distanze temporali inferiori saranno scartati.

Inoltre, lo scan BLE e l'inoltro degli Advertisement BLE viene effettuato soltanto per $1/6T_B$ (e non istantaneamente per garantire un po' di margine di errore dovuto alla non omogeneità dei dispositivi) in base a delle regole fissate dall'applicazione sulla sincronizzazione con un orario (che i dispositivi dovrebbero avere tutti più o meno sincronizzato dato che sono connessi ad internet).

Quanto descritto fino ad ora può essere visualizzato graficamente nella Figura 2.1.

2.3.1. Generazione chiavi effimere

Ogni 24h l'utente genera una coppia fittizia $(SK_f, PK_f) \leftarrow \mathrm{Gen}(1^n)$ che NON viene autenticata dal ministero.

L'utente effettua un commitment (avente proprietà di Hiding e Binding) della PK appena generata al server dell'MD descritto nella fase 2.1. Lo fa collegandosi al server in una connessione cifrata TLS utilizzando anche il proprio certificato. Una volta autenticato invia al server un commitment implementato come

$$\mathrm{c_{PK_f}} = \mathrm{SHA256(r \parallel PK_U \parallel PK_f \parallel DATE)} \tag{2.3}$$

dove

- r è una stringa di randomness di 256 bit;
- PK_U è la chiave pubblica dell'utente autenticata dal MD (nel certificato);
- PK_f è la chiave effimera appena generata;
- $DATE$ è la data del giorno in cui PK_f è stata generata.

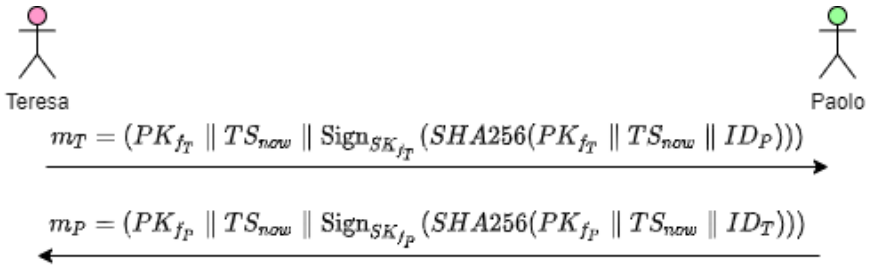

Figura 2.2 Messaggi scambiati nella convalida dei contatti descritta nella fase 2.3.3.

Ovviamente l'utente conserverà la r necessaria ad aprire c_{PK_f} per gli ultimi 20 giorni, oltre che le coppie di chiavi stesse, il campo DATE e, per comodità, l'ID associato alla PK_f calcolato successivamente nella *(2.4)*. Il MD immagazzina c_{PK_f} legandolo alla PK_U dell'utente ed al giorno corrente.

2.3.2. Scambio degli ID

In questo momento i dispositivi si scambiano in Advertisement BLE i loro ID generati come

$$ID_U = SHA256\big(PK_f\big)[0{:}\,127] \tag{2.4}$$

dove

- PK_f è la chiave effimera dell'utente generata al passo 2.3.1;
- $[0{:}\,127]$ è il troncamento dell'output della funzione $SHA256$ ai primi 16 byte.

La generazione dell'ID può essere vista come casuale considerando che la funzione SHA256 troncata ai primi 128 bit può essere modellata come Random Oracle. In più, si può utilizzare questo Random Oracle per dimostrare che un utente con una certa chiave effimera abbia davvero utilizzato quell'ID.

Gli utenti scambiano questi ID nelle modalità illustrate precedentemente e riassunte nella Figura 2.1. Per ogni ID ricevuto in advertising BLE, l'utente memorizza

$$(ID, \#\text{occorrenze}, \text{Timestamp ultima occorrenza}) \tag{2.5}$$

dove

- si utilizza il campo Timestamp per verificare se le proprietà sui pacchetti broadcast BLE esposte prima sono valide (che arrivi un pacchetto ogni T_B)
- il campo #occorrenze viene incrementato di uno ogni volta che si riceve un pacchetto corretto.

2.3.3. Convalida dei contatti

Si consideri ora lo scenario in cui due utenti del sistema abbiano cumulato $15min/T_B$ pacchetti e quindi un tempo di contatto totale di $15min$. In particolare, Teresa si accorge per prima di aver raggiunto questa soglia (è probabile che a Paolo manchi ancora un pacchetto, ma si può considerare questa situazione, comunque, come un contatto per prevenire altri tipi di problemi alla completeness).

1. Il dispositivo di Teresa fa un pair BT con quello di Paolo. A questo punto il dispositivo di Paolo si aspetta un messaggio esattamente nel formato come verrà mostrato, altrimenti abortirà immediatamente la comunicazione.
2. Teresa invia a Paolo un messaggio composto come segue

$$m_T = \Big(PK_{f_T} \parallel TS_{now} \parallel \mathrm{Sign}_{SK_{f_T}}\Big(SHA256\big(PK_{f_T} \parallel TS_{now} \parallel ID_P\big)\Big)\Big) \tag{2.6}$$

dove

a. PK_{f_T} è la chiave effimera utilizzata da Teresa nella giornata corrente;
b. TS_{now} è un timestamp generato al momento della generazione del contatto;
c. $\text{Sign}_{SK_{f_T}}\left(SHA256\left(PK_{f_T} \parallel TS_{now} \parallel ID_P\right)\right)$ è la firma con chiave segreta effimera di Teresa dell'output della funzione SHA256 (modellata come Random Oracle) che prende in ingresso la concatenazione della chiave pubblica effimera di Teresa, il timestamp precedente e l'ID_P di Paolo.

3. Paolo verifica le seguenti cose:
   a. il TS_{now} arrivato non differisce più del tempo corrente nel suo dispositivo di 30 secondi.
   b. l'ID dell'utente che ha richiesto il pairing è dato dalla formula (2.4) utilizzando come input PK_{f_T} ricevuta;
   c. la firma è eseguita correttamente, quindi che

$$\text{Vrfy}_{PK_{f_T}}\left(SHA256\left(PK_{f_T} \parallel TS_{now} \parallel ID_P\right)\right) == 1 \tag{2.7}$$

   dove il messaggio $SHA256\left(PK_{f_T} \parallel TS_{now} \parallel ID_P\right)$ viene visto come challenge in quanto può essere modellata come Random Oracle ed inoltre viene ricostruito autonomamente da Paolo, sostituendo il suo ID attuale nella formula.

Se le verifiche non hanno esito positivo l'operazione viene abortita. Altrimenti Paolo invia un messaggio speculare e Teresa effettua le medesime verifiche. Il messaggio viene riportato ed è il seguente

$$m_P = \left(PK_{f_P} \parallel TS_{now} \parallel \text{Sign}_{SK_{f_P}}\left(SHA256\left(PK_{f_P} \parallel TS_{now} \parallel ID_T\right)\right)\right) \tag{2.8}$$

Entrambi gli utenti, in caso positivo, memorizzano il messaggio scambiato che vale come convalida del contatto.

Quanto descritto viene rappresentato graficamente nella Figura 2.2.

2.4. Fase di accertata positività

Quando un utente risulta positivo al tampone può richiedere all'HA un token da utilizzare per caricare i contatti autenticati sull'apposito server dell'MD.

1. L'utente si connette con protocollo TLS al server dell'HA, utilizzando anche il suo certificato.
2. Il sistema digitalizzato dell'HA verifica l'identità e l'avvenuta positività.
3. L'utente comunica all'HA le ultime venti PK_{f_U} utilizzate negli ultimi 20 giorni (o meno nel caso l'applicazione non sia stata avviata un giorno o non abbia riscontrato contatti, ma sicuramente non di più), oltre che i corrispondenti campi DATE (che rappresenta il giorno di utilizzo) e r (la randomness utilizzata per effettuare il commitment).
4. Per ogni tripla $\left(PK_{f_U} \parallel \text{DATE} \parallel r\right)$, l'HA apre il commitment (supponiamo che abbia accesso in lettura al DB contenente i commitment) e lo verifica, valutando se

$$c_{PK_f} == SHA256\left(r \parallel PK_U \parallel PK_f \parallel DATE\right) \tag{2.9}$$

   e nel caso l'apertura non vada a buon fine allerta la P e l'AG che faranno le dovute verifiche. In caso positivo, per ogni tripla, l'HA invia all'utente un token T così composto

$$\text{T} = \left(PK_{f_U} \parallel DATE \parallel \text{Sign}_{SK_{HA}}\left(SHA256\left(PK_{f_U} \parallel DATE\right)\right)\right) \tag{2.10}$$

Subito dopo aver ricevuto il token, l'utente si connette al server dell'MD con una connessione TLS (senza utilizzare il certificato dell'utente ed evitando autenticazioni) e invia un token alla volta (per ogni token

viene creata una nuova connessione). Il ministero verifica che la firma del output del Random Oracle del messaggio con la chiave pubblica del HA corrisponda e accetta i contatti autenticati come nella (2.6). Per ogni contatto ricevuto l'MD verifica che:

- la firma del messaggio, dove ID viene generato come nella (2.4) è corretta;
- il TS nel contatto autenticato corrisponda al DATE presente nel token;

e nel caso le verifiche siano corrette carica sull'apposito server soltanto l'ID generato come nella (2.4). In caso negativo avvia un procedimento con P e AG per verificare quanto accaduto (l'identità potrebbe non essere protetta in quanto l'HA potrebbe rivelare chi ha ricevuto il token in caso di truffe o illeciti).

Quanto descritto è schematizzato in Figura 2.3.

L'MD utilizzerà un server con una struttura replicata per fare in modo che tutti gli utenti possano controllare se uno dei loro ID è stato inserito, quindi si può prevedere che ci saranno alcuni momenti della giornata (es., 2 o 3) dove i dati vengono messi a disposizione di tutti gli utenti.

2.5. Notifica

I dispositivi degli utenti periodicamente (preferibilmente con connessione Wi-Fi) si connettono al server dell'MD e verificano se negli ID ancora non letti vi è uno che corrisponde ad uno degli ID che hanno avuto negli ultimi 20 giorni. In caso positivo vengono notificati dall'applicazione. Si potrà richiedere un tampone gratuito tramite l'applicazione dove

1. L'utente U si connette al HA utilizzando una connessione basata su TLS e comunica di voler effettuare un tampone utilizzando l'ID che corrisponde alla *(2.4)* con la PK_{f_U} corrispondente a quell'ID (essendo modellato come Random Oracle, soltanto chi possiede la PK_{f_U} può dimostrare di aver avuto un contatto). Inoltre, l'utente, invia la randomness r e il campo $DATE$ relativi al commitment (*2.3*), in modo che l'HA possa verificare se quella PK_{f_U} è associata alla PK_U dell'utente che sta richiedendo il tampone.
2. L'HA verifica l'esistenza dell'ID all'interno del server dell'MD. L'HA, inoltre, verifica se questa richiesta è arrivata nelle 24h successive alla pubblicazione dell'ID viene concesso un tampone gratuito; altrimenti richiede il pagamento di 0.001BTC per procedere alla prenotazione.

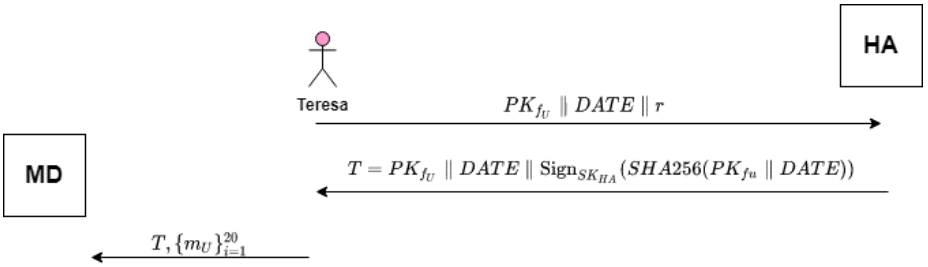

Figura 2.3 Schematizzazione semplificata della fase di comunicazione dei contatti. Per ogni chiave effimera PK_{f_U} viene richiesto un token T, che viene poi utilizzato per comunicare i contatti all'MD.

3. WP3 – Analisi

3.1. Completeness

Supponendo che tutti seguano il protocollo descritto possiamo notare che i contatti vengono correttamente registrati, che la positività può essere comunicata senza problemi, che le notifiche arrivano correttamente agli utenti interessati e che la possibilità di eseguire il tampone gratuito viene garantita. In assenza di avversari, sia le verifiche effettuate dagli utenti nella fase di pairing sia le verifiche effettuate dall' HA e dall'MD durante la fase di comunicazione dei contatti saranno sempre positive.

Questo si può verificare anche dalla simulazione effettuata nel WP4.

3.2. Confidenzialità

Consideriamo uno ad uno gli avversari descritti nel paragrafo 1.2.

3.2.1. Il curioso $\mathcal{C}$

Distinguiamo due casi nei possibili attacchi da parte di questo avversario: se questo ha ricevuto una notifica e vuole scoprire quale altro utente l'ha generata o se questo utente vuole indistintamente conoscere tutti i positivi presenti nel sistema di CT.

Nel primo caso si noti che un utente (in questo caso avversario) nella fase di notifica controlla se negli ID postati sul server dell'MD vi è uno degli ID che ha usato negli ultimi 20 giorni. Non sono presenti informazioni su chi sta dall'altra parte del contatto a questo livello e l'avversario non può neanche cercarlo tra le sue informazioni in quanto il suo ID si trova (ovviamente) in tutti i contatti che ha scambiato. Da questo deriva che l'avversario non può far altro che provare a indovinare a caso uno tra i suoi contatti (a cui potrebbe aver associato un'identità) soddisfacendo la proprietà C.2.

Nel secondo caso, un avversario avrebbe difficoltà a capire chi è un contatto positivo. L'unica informazione che ha a disposizione sono infatti gli ID dei contatti dei positivi, caricati sull'apposito server (n.b. si ricorda che gli ID non sono direttamente associati all'identità di un cittadino). Anche se potesse associare tutti gli ID alle persone, utilizzando antenne e strumenti di riconoscimento come videocamere, pagamenti elettronici, posti di blocco, un software aggiuntivo da distribuire su questi elementi sparsi sul territorio e del personale dedicato, dovrebbe conoscere tutti i contatti degli utenti il cui id è stato pubblicato sul server dell'MD, aumentando quindi la probabilità di scoprire il positivo, che sarebbe maggiore di quella richiesta in C.3. In generale questo attacco è possibile ma un avversario potrebbe essere dissuaso dall'elevato costo; consideriamo comunque questo attacco possibile e quindi diciamo che la proprietà C.3 è rispettata quasi pienamente. Se questo avversario figurasse nell'MD che collude con l'HA, allora potrebbe conoscere tutti gli utenti positivi (ma questo anche senza applicazione) e anche le identità degli utenti che sono contatti a rischio che si prenotano per il tampone dopo della notifica in quanto l'HA potrebbe comunicare i dati dall'apertura del commitment (fase 2.5). In generale assumiamo che la reputazione dell'HA sia buona in quanto, data la sua struttura intrinsecamente decentralizzata, anche se si dovessero scoprire alcuni membri corrotti (es., medici) questi verrebbero facilmente radiati e i loro misfatti verrebbero velocemente riparati (nei limiti del riparabile) in quanto è vitale che questa autorità mantenga un'ottima reputazione.

3.2.2. Lo spione $\mathcal{S}$

Consideriamo i due casi (spiare le posizioni o i contatti) separatamente.

Nel primo caso, questo avversario non può ottenere informazioni direttamente dallo stesso sistema in quanto le identità delle persone sono sempre mascherate sotto una falsa PK_f. Quest'ultima non viene condivisa dall'MD, ma soltanto con i contatti stretti o comunque la sua funzione (2.4) in Advertisement

BLE. Nella fase 2.4 questa viene comunicata all'HA, che potrebbe quindi associarla all'identità di un cittadino (ricordiamo che l'HA conosce già gli utenti positivi ed è ammesso che li conosca). Inoltre, sempre in questa fase, quando viene inoltrato il token, l'MD, senza ulteriori informazioni non è in grado di associare l'identità dell'utente alla PK_f o all'ID.

Consideriamo però il caso in cui l'avversario voglia utilizzare informazioni esterne all'applicazione, creando un link tra l'ID o la PK_f con la vera identità del cittadino (questo può essere effettuato in tanti modi, usando antenne BT combinate con telecamere, sistemi di pagamento, ecc.). In questo caso, l'avversario sarebbe in grado di tracciare gli spostamenti. Per effettuare un link tra persona e ID, l'avversario deve disporre di scanner BT sul territorio (e uno degli strumenti prima descritti), che possono raggiungere un costo molto alto se le zone da coprire sono vaste. Si ricorda, infatti, che è più semplice effettuare il link in un luogo poco affollato che in un luogo affollato, ma in questo caso la spesa per l'avversario aumenterebbe. Per questi motivi consideriamo l'attacco possibile, ma lo riteniamo allo stesso tempo poco probabile date le richieste economiche, umane e computazionali.

Consideriamo quindi la proprietà C.4 non completamente rispettata.

Nel secondo caso, un avversario esterno al sistema non può in nessun momento accedere dall'interno del sistema a informazioni sui contatti. Può però decidere di utilizzare la stessa infrastruttura appena descritta per cercare di captare le convalide dei messaggi generati nella fase 2.3.3. Se l'avversario fosse interno al sistema (es., MD), allora potrebbe visionare informazioni sui contatti al punto 2.4, dove gli vengono esplicitamente inviate se un utente risulta positivo. In questo caso otterrebbe informazioni parziali e da utenti casuali, e comunque non legate all'identità degli utenti presenti nei contatti ma soltanto alla chiave effimera PK_f. Dovrebbe quindi almeno associare le PK_f all'identità, come descritto precedentemente. Per questi motivi consideriamo la proprietà C.1 non completamente rispettata.

Per quest'ultimo avversario, il tempo di tracciamento, avendo associato l'ID o la PK_f all'identità, dura per il tempo di durata della PK_f (in questo caso un giorno). Quando verrà generata una nuova PK_f allora sara necessario riuscire a ripetere tutto il processo. Aumentando la frequenza di aggiornamento della PK_f (e di conseguenza dell'ID) il sistema aumenterebbe in confidenzialità, ma ne perderebbe in efficienza in quanto l'ipotesi dei volumi crescerebbe linearmente sui due server dell'MD descritti in 2.1 per la stessa costante di cui si aumenta la frequenza di aggiornamento, nonché nella memoria utilizzata da ogni dispositivo.

3.3. Integrità

3.3.1. Il diffamatore $\mathcal{D}$

Consideriamo ognuna delle possibili tipologie per questo avversario

- ***False positive:*** *Questo avvers*ario non può manipolare direttamente le notifiche in quanto è direttamente l'utente a verificare la sua presenza in un database contenente gli ID con contatto a rischio (fase 2.5).
  Questo avversario non può fingere di essere positivo in quanto nella fase 2.4 (riassunta nella Figura 2.3) l'HA confronta l'identità reale (letta dal certificato digitale) con una propria lista interna dei positivi, quindi non falsificabile dal lato dell'applicazione. Per consentire di caricare i contatti, l'utente positivo comunica, per ogni PK_{f_U} utilizzata negli ultimi 20 giorni, la randomness r e $DATE$ in modo che l'HA possa aprire il commitment (*2.3*) e verificare che quella PK_{f_U} sia stata davvero associata all'utente. In questo caso stiamo utilizzando la proprietà di Binding del commitment.

Non si può cercare di falsificare il token (*2.10*) in quanto è firmato dall'HA e il messaggio firmato può essere vista come randomico in quanto output di un Random Oracle, quindi sensibilmente difficile da forgiare. Non si può neanche aprire un token qualsiasi (avendo ottenuto informazioni su di esso in un qualsiasi modo, anche comprandole) in quanto è riportata la PK_{f_U} all'interno, quindi non si avrebbero contatti da comunicare in quanto l'ID riportato nel contatto (*2.6*) è quello dell'utente che sta comunicando il contatto e vi è una verifica che l'ID venga generato a partire dalla PK_{f_U} come nella (*2.4*).
Quindi questo avversario soddisfa pienamente la proprietà I.1

- ***Fake match:*** Se questo avversario risultasse davvero positivo allora nessuno potrebbe impedirgli di comunicare contatti anche non avvenuti e raccolti mediante l'utilizzo di hardware esterno come antenne ecc. Tuttavia, deve comunque essere registrato al sistema, quindi tutti i contatti devono essere associati all'avversario che dovrà realmente risultare positivo. Consideriamo questa possibilità davvero poco probabile. Inoltre, essendo la sua identità reale associata al sistema, potrebbe facilmente rischiare di andare incontro a problemi legali.
  Questo avversario soddisfa quasi pienamente la proprietà I.2.
- ***Sibling:*** Questo avversario non può impersonare altri utenti perché dovrebbe riuscire a firmare messaggi a partire dalla PK_{f_U} che stanno utilizzando (e quindi scoprire la SK_{f_U} in meno di un giorno per poi avere margine per utilizzarla). Non può neanche utilizzare un ID che potrebbe aver associato ad un'altra persona al posto del suo in quanto nella fase 2.3.3 vi è una verifica da parte dell'utente che riceve il contatto dove viene verificato che l'ID è ottenuto come nella (*2.4*).
  Questo avversario soddisfa la proprietà I.3.
- ***Replay:*** Considerando un attacco di questo tipo a tempo differito, facciamo notare che i messaggi di convalida dei contatti sono validi soltanto per 30 secondi e dopo questo tempo vengono rifiutati (fase 2.3.3). Inoltre, potrebbe essere difficile per l'avversario sfruttare una malleabilità della firma in quanto viene firmato un messaggio output di un Random Oracle, quindi che può essere visto come casuale.
  Si potrebbero però collegare utenti che si trovano in luoghi diversi semplicemente ripetendo in tempo reale i segnali comunicati dai due utenti. In questo caso i contatti verrebbero correttamente salvati. Si consideri però che i ripetitori BT devono essere distribuiti davvero capillarmente sul territorio perché perdere alcuni pacchetti da parte di un utente renderà completamente inefficace l'attacco. Questo attacco potrebbe essere risolto utilizzando il GPS del dispositivo per verificare l'effettiva vicinanza tra i due utenti, ma preferiamo non includere questa funzionalità nella trattazione in quanto peggiorerebbe l'efficienza e potrebbe influenzare negativamente gli utenti spingendoli a non utilizzare l'applicazione, che è comunque su base volontaria, per evitare di indurli a credere di essere controllati.
  La proprietà I.2 non è completamente rispettata.
- ***False Negative:*** Questo avversario non può non far arrivare le notifiche in quanto dovrebbe alterare le informazioni sul server dell'MD (se l'avversario fosse l'MD stesso allora l'attacco avrebbe successo ma consideriamo questa eventualità non sensata in quanto è di interesse dell'MD mantenere complete le funzionalità dell'applicazione). Questo avversario non può neanche alterare la lettura degli ID a contatto a rischio in quanto questa è effettuata mediante connessione TLS.
  Questo avversario rispetta pienamente la proprietà I.4.

3.3.2. Il riservato $\mathcal{R}$

Questo avversario rimane in grado di agire e quindi non comunicare le proprie informazioni al sistema, che si basa sulla volontarietà di partecipazione. Essendo la trasparenza molto garantita non avendo introdotto parti fidate, allora si suppone che questo tipo di avversari siano in un numero molto ristretto

3.3.3. L'avaro $\mathcal{A}$

Questo avversario non è in grado di fingere di aver ricevuto una notifica in quanto può visualizzare soltanto gli ID, e gli viene richiesto di conoscere la PK_{f_U} associata all'ID secondo la (*2.4*), oltre che i campi $DATE$ e r per aprire il commitment e verificare che la PK_{f_U} sia davvero associata alla sua identità.

Questo avversario rispetta la proprietà I.5

3.4. Trasparenza

La trasparenza è totale in quanto in ogni momento un utente può controllare cosa sta ricevendo sul proprio dispositivo e quali sono i dati che ha già immagazzinato. L'unica assunzione che viene fatta rispetto agli attori del sistema è che l'HA non abbia motivo di avere comportamenti da avversario e che non colluda con il MD. Per il resto, nessuna parte del sistema conosce tutte le informazioni.

3.5. Efficienza

Il sistema progettato presenta delle vulnerabilità in efficienza a causa del pairing che deve essere effettuato tra i dispositivi, che potrebbe essere computazionalmente costoso in caso di numerosi contatti stretti contemporaneamente, ed anche a causa dell'elevato numero di messaggi scambiati e da conservare. Inoltre, l'efficienza è minata anche dalle importanti ipotesi sui volumi fatte nella fase 2.1. L'efficienza però non può essere migliorata senza avere delle perdite di integrità o confidenzialità sugli attaccanti descritti, dal momento che rendere le operazioni più rapide e snelle implicherebbe lasciare margine a possibili attacchi che minerebbero alla integrità o alla confidenzialità.

3.6. Riepilogo

Nella Figura 3.1 viene riportato il rispetto delle 4 proprietà. Nella Tabella 1 vengono riportati i riferimenti alle primitive crittografiche utilizzate.

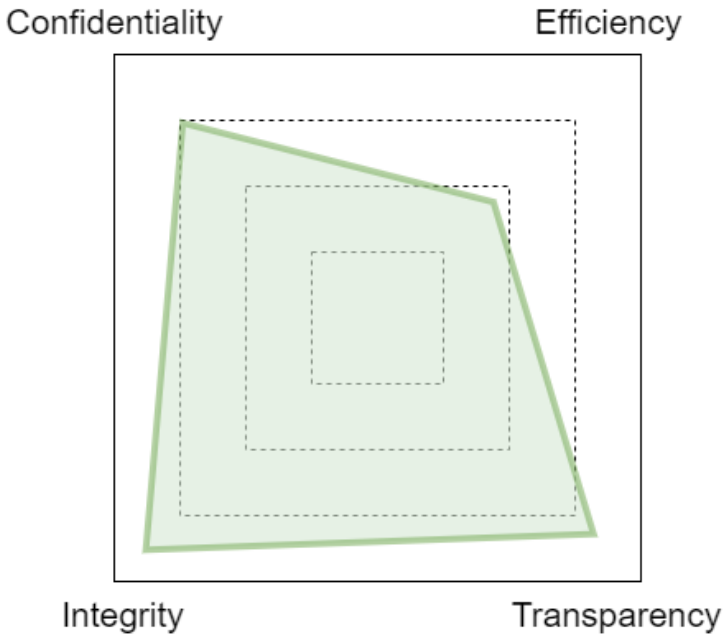

Figura 3.1 Rappresentazione grafica del soddisfacimento delle proprietà.

Primitiva Crittografica	Fase	Descrizione
Certificato Digitale	2.1 2.2 2.4, 2.5	L'MD si fa firmare un certificato per diventare autorità di certificazione per gli utenti. Il'MD certifica gli utenti. Combinati con TLS per autenticarsi all'HA.
TLS (solo certificato server)	2.2 2.4	L'utente utilizza una connessione TLS con l'MD per inviare sue informazioni sensibili. L'utente utilizza una connessione TLS con l'MD non autenticata per inviare token dell'HA e contatti verificati.
TLS (Certificato server e client)	2.4, 2.5	L'utente utilizza TLS autenticandosi all'HA, che necessita di conoscere la sua identità.
Commitment (Hiding)	2.3.1	L'utente usa il commitment per dichiarare di avere una chiave pubblica effimera ma non rivelarla.
Commitment (Binding)	2.4, 2.5	L'utente invia il messaggio e la randomness per consentire all'HA di verificare la veridicità della sua chiave effimera.
Random Oracle	2.3.2, 2.3.3	L'utente genera un ID a partire dalla sua chiave effimera attraverso un random oracle, che servirà anche per effettuare la verifica 3.b nella fase 2.3.3.
Random Oracle, Firma Digitale	2.3.3, 2.4	L'utente utilizza un Random Oracle per generare un messaggio casuale (challenge) e firmarlo, dimostrando di essere lui.

Tabella 1 Utilizzo primitive crittografiche.

4. WP4 – Simulazione di funzionamento

Per la realizzazione del prototipo ricordiamo le semplificazioni concordate:

- Utilizzo di certificati self-signed basati su RSA invece delle curve ellittiche secp256k1;
- Utilizzo di un'unica coppia (PK_f, SK_f) per ogni utente (2.3.1), quindi la simulazione di un'unica giornata del sistema;
- Lo scambio degli ID viene supposto già accaduto e viene rappresentata una situazione nota, eseguendo soltanto la fase 2.3.3 utilizzando una connessione TCP per simulare il pairing Bluetooth.
- Semplificazione rispetto alle comunicazioni tra HA e MD: queste non vengono effettuate su rete ma direttamente nel codice passando i riferimenti.

Si è deciso di rappresentare una simulazione di quanto accade nella Figura 1.1. La simulazione può essere eseguita semplicemente compilando tutti i file presenti nel package `it.diem.unisa.cs.gruppo10` e lanciando il file `MainSimulation.java` che richiama tutti i metodi e le classi utilizzate per la simulazione. All'interno dei file forniti sono presenti anche i vari certificati, trust store e key store utilizzati per le comunicazioni TLS. Caricando il progetto in un IDE come Intellij Idea Community Edition che supporta nativamente Java Maven, tutte le dipendenze (tra cui `BouncyCastle`) vengono risolte automaticamente. Inoltre, va modificata la stringa `Util.resourcesPath` in quanto va inserito in base alla current directory su cui si utilizza Java; l'importante è che questa arrivi alla directory `"./src/main/resources/"` che contiene i file di configurazione, i trust store ed i key store generati.

Tutto il materiale può essere reperito su GitHub alla repository https://github.com/pepealessio/CS_Team10_2021 ed è stato inoltre allegato in fase di consegna.

L'UML che rappresenta il codice realizzato è mostrato in Figura 4.1 ed inoltre è descritto testualmente nel file `gruppo10.plantuml`.

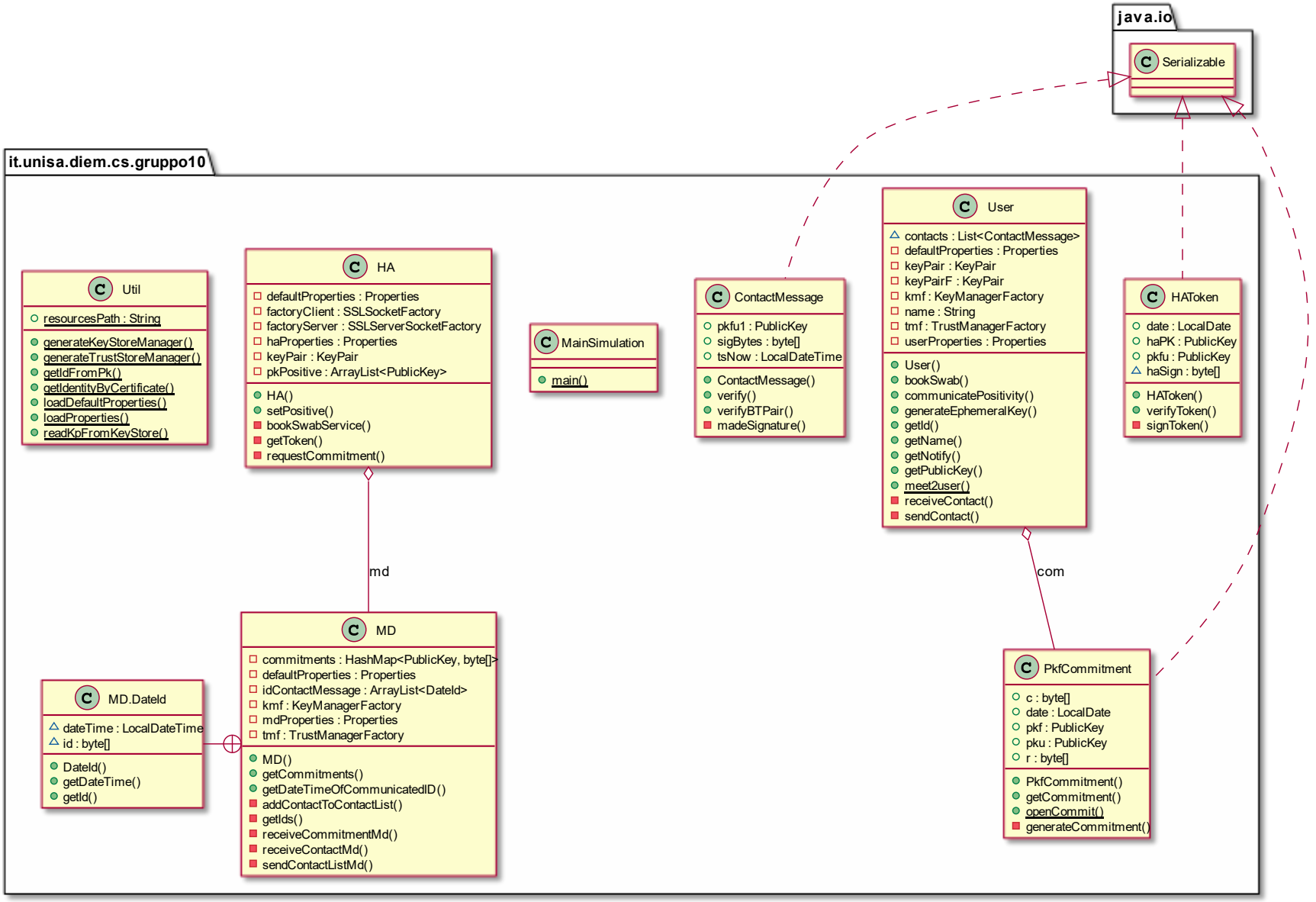

Figura 4.1 WP4 Class Diagram

4.1. Inizializzazione del sistema

Questa fase vuole essere una simulazione di quanto realizzato nella fase 2.1 che riguarda la creazione dei certificati, key store e trust store utilizzati dall'MD e dall'HA.

Di seguito viene riportato il processo eseguito (in Ubuntu) per generare i certificati, i trust store ed i key store. La creazione dei certificati, dei trust store e dei key store è avvenuta tramite l'utilizzo del Java Key tool, uno strumento utilizzato per la gestione dei certificati che consente di poter creare le chiavi pubbliche, private e dei certificati che le contengono. Questi certificati vengono memorizzati in file con estensione `.jks` e risultano essere necessari per generare le catene di fiducia.

Al fine di poter utilizzare gli script così come descritti è conveniente muoversi nella cartella dove si vogliono generare i file e generare le sottocartelle necessarie, pertanto, nella nostra VM:

```
cd CybersecurityProject

mkdir ./MDFiles
mkdir ./HAFiles
mkdir ./PublicFiles
mkdir ./UserFiles
mkdir ./UserFiles/Alessio
mkdir ./UserFiles/Paolo
mkdir ./UserFiles/Teresa
mkdir ./UserFiles/Luigi
```

Nella fase di inizializzazione del sistema è necessario che avvenga la creazione dei key store appartenenti all'MD e all'HA, nonché la generazione dei certificati associati alle due autorità:

- Creazione del key store dell'MD, definito come MDKey

```
sudo keytool -genkey -noprompt -trustcacerts -keyalg RSA -keysize 4096 -alias MD -dname
"cn=localhost,ou=MD,o=Government,c=CL" -keypass MDPassword -keystore ./MDFiles/MDKey.jks
-storepass MDPassword
```

- Creazione del key store dell'HA, definito come HAKey

```
sudo keytool -genkey -noprompt -trustcacerts -keyalg RSA -keysize 4096 -alias HA -dname
"cn=localhost, ou=HA, o=Sanity, c=CL" -keypass HAPassword -keystore ./HAFiles/HAKey.jks
-storepass HAPassword
```

- Creazione del certificato dell'MD, definito come MDCer, a partire dal suo key store

```
sudo keytool -export -alias MD -storepass MDPassword -file ./MDFiles/MDCer.cer -keystore
./MDFiles/MDKey.jks
```

- Creazione del certificato dell'HA, definito come HACer, a partire dal suo key store

```
sudo keytool -export -alias HA -storepass HAPassword -file ./HAFiles/HACer.cer -keystore
./HAFiles/HAKey.jks
```

Contestualmente, è necessario che venga creato un trust store pubblico, definito come PublicTrust, in cui devono essere aggiunti i certificati dell'MD e dell'HA:

- Creazione di PublicTrust e aggiunta del certificato dell'MD

```
sudo keytool -import -noprompt -v -trustcacerts -alias MDCert -keystore
./PublicFiles/PublicTrust.jks -file ./MDFiles/MDCer.cer -keypass MDPassword -storepass
PublicPassword
```

- Aggiunta del certificato dell'HA nel PublicTrust

```
sudo keytool -import -noprompt -v -trustcacerts -alias HACert -keystore
./PublicFiles/PublicTrust.jks -file ./HAFiles/HACer.cer -keypass HAPassword -storepass
PublicPassword
```

Infine, è necessario che vengano creati i trust store dell'MD e dell'HA, definiti come MDTrust e HATrust, che dovranno contenere i rispettivi certificati:

- Creazione di MDTrust e aggiunta di HACer

```
sudo keytool -import -noprompt -v -trustcacerts -alias HACert -keystore
./MDFiles/MDTrust.jks -file ./HAFiles/HACer.cer -keypass HAPassword -storepass
MDPassword
```

- Creazione di HATrust e aggiunta di MDCer

```
sudo keytool -import -noprompt -v -trustcacerts -alias MDCert -keystore
./HAFiles/HATrust.jks -file ./MDFiles/MDCer.cer -keypass MDPassword -storepass
HAPassword
```

A rappresentare l'MD vi è la classe MD che mediante l'utilizzo di Thread simula l'esistenza dei server descritti nella progettazione. Questi vengono avviati direttamente all'avvio della classe MD `public MD()` e vengono automaticamente disattivati alla terminazione del main (in quanto dichiarati daemon).

A rappresentare l'HA vi è la classe HA costruita in modo identico all'MD, ma con le sue funzionalità.

Si riporta il codice rilevante utilizzato in questa fase nel main della simulazione:

```
MD md = new MD();
HA ha = new HA(md);
```

(In un'implementazione reale sarebbe stato necessario utilizzare dei certificati collegati con una catena di certificazione fino ad un'autorità di certificazione root. Per quanto riguarda i server dell'MD e dell'HA, avrebbero dovuto essere realmente implementati con apposito hardware e non simulati nella stessa macchina.)

4.2. Registrazione degli utenti

Una volta completata la fase di inizializzazione dei certificati, dei key store e dei trust store collegati all'MD e all'HA, nonché del trust store pubblico che simula quello che verrebbe inserito in tutte le applicazioni, nella fase 2.12 è necessario attendere l'iscrizione degli utenti al sistema. Per ogni utente generico iscritto al sistema verranno effettuati una serie di passaggi (in questo caso prendiamo come esempio Alessio):

- Creazione del key store dell'utente, definito come AlessioKey

```
sudo keytool -genkey -noprompt -trustcacerts -keyalg RSA -keysize 4096 -alias Alessio -
dname "cn=localhost, ou=Alessio, o=Citizen, c=CL" -keypass AlessioPassword -keystore
./UserFiles/Alessio/AlessioKey.jks -storepass AlessioPassword
```

- Creazione del certificato dell'utente, definito come AlessioCer, a partire dal suo keystore

```
sudo keytool -export -alias Alessio -storepass AlessioPassword -file
./UserFiles/Alessio/AlessioCer.cer -keystore ./UserFiles/Alessio/AlessioKey.jks
```

- Aggiunta del certificato dell'utente ad HATrust

```
sudo keytool -import -noprompt -v -trustcacerts -alias AlessioCer -keystore
./HAFiles/HATrust.jks -file ./UserFiles/Alessio/AlessioCer.cer -keypass AlessioPassword -
storepass HAPassword
```

- Aggiunta del certificato dell'utente a MDTrust

```
sudo keytool -import -noprompt -v -trustcacerts -alias AlessioCer -keystore
./MDFiles/MDTrust.jks -file ./UserFiles/Alessio/AlessioCer.cer -keypass AlessioPassword
-storepass MDPassword
```

Lo script completo da eseguire per ottenere tutti i file è contenuto nella cartella resources nel file generateCertificate.txt. L'unica differenza tra un utente e l'altro è che cambia il nome.

In questa fase viene utilizzata parte della Classe User. La classe User ha un costruttore che richiede venga passata una stringa con il nome dell'utente (che funge anche da Alias).

```
public User(String name)
```

Nel costruttore viene inizializzata una lista di contatti vuota e vengono caricate le chiavi rilasciate dall'MD dal Key Store dell'utente utilizzando il metodo

```
public static KeyPair readKpFromKeyStore(String filePath, String password, String alias)
```

presente nella classe `Util`.

Si riporta il codice rilevante utilizzato in questa fase nel main della simulazione:

```
User teresa = new User("Teresa");
User paolo = new User("Paolo");
User alessio = new User("Alessio");
User luigi = new User("Luigi");
```

(In un'implementazione reale anche questi sarebbero stati identificati dall'applicazione utente che avrebbe contenuto comunque tutti i tipi necessari per scambiare i messaggi. Anche i loro certificati avrebbero dovuto essere collegati ad una autorità di certificazione per mezzo dell'MD)

4.3. Fase dei contatti

In questa fase si è voluta rappresentare la fase 2.3, che con le semplificazioni introdotte in precedenza corrisponde alle fasi 2.3.1 e 2.3.3.

4.3.1. Generazione delle chiavi e commitment

Rappresentante la fase 2.3.1 viene utilizzato il metodo

```
public void generateEphemeralKey()
```

della classe User per generale la coppia (*2.2*). Si ricorda che viene salvata un'unica coppia in quanto la simulazione è di un unico giorno di contatti. (Nella reale implementazione si sarebbero dovute mantenere le ultime venti PK_{f_U} associate alla data di utilizzo)

In questo metodo viene anche inizializzata una comunicazione TLS con l'MD dove viene effettuato il commitment della chiave.

Il commitment è rappresentato dalla classe `PkfCommitment`, che sostanzialmente immagazzina tutti i campi necessari per effettuare il commitment ed anche il commitment stesso e fornisce due metodi:

- `private void generateCommitment()` che calcola il commitment come nella formula (*2.3*); viene per semplicità richiamato dal costruttore;
- `public static boolean openCommit(byte[] r, PublicKey pku, PublicKey pkf, LocalDate date, byte[] c)` che, passando in input la randomness, il messaggio (le parti che lo compongono per semplicità) ed il commitment precedentemente generato, apre il commitment verificando se i dati comunicati sono validi.

Per simulare la connessione al server dell'MD (e poi dell'HA) sono state utilizzate delle connessioni TLS gestite tramite Socket SSL costruite sul localhost, su delle specifiche porte. In particolare, ogni `SSLServerSocket` è posta all'interno di un ciclo infinito posto in un Thread separato, per simulare la continua possibilità di un server di accettare connessioni. Per ogni classe è stato predisposto in un file di proprietà i nomi e le relative password dei trust store e dei key store dei vari utenti del sistema, generati al passo precedente (uno per l'MD, uno per l'HA, uno per ogni utente della simulazione). È stato inoltre predisposto il file `default.properties` per i numeri di porta dei server su cui effettuare la connessione.

Ogni classe presenta tra i propri attributi un `KeyManagerFactory` ed un `TrustManagerFactory`, che vengono inizializzati nei costruttori delle varie classi mediante i metodi statici `generateKeyStoreManager` e `generateTrustStoreManager` presenti nella classe `Util.java`. Questi metodi prendono in input il percorso del file e la password rispettivamente del Key Store e del Trust Store per il quale si sta utilizzando restituendo un gestore di Key Store e Trust Store.

```
public static TrustManagerFactory generateTrustStoreManager(String filePath, String
password) throws NoSuchAlgorithmException, KeyStoreException {
    TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
    KeyStore ts = KeyStore.getInstance("JKS");
    char[] passTs = password.toCharArray();
    try {
        ts.load(new FileInputStream(filePath), passTs);
    } catch (IOException | CertificateException | NoSuchAlgorithmException e) {
        System.err.println("Trust store not found!");
        System.exit(1);
    }
    tmf.init(ts);
    return tmf;
}

public static KeyManagerFactory generateKeyStoreManager(String filePath, String
password) throws NoSuchAlgorithmException, KeyStoreException, UnrecoverableKeyException
{
    KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
    KeyStore ks = KeyStore.getInstance("JKS");
    char[] passKs = password.toCharArray();
    try {
        ks.load(new FileInputStream(filePath), passKs);
    } catch (IOException | CertificateException e) {
        System.err.println("User key store not found!");
        System.exit(1);
    }
    kmf.init(ks, passKs);
    return kmf;
}
```

In questo modo, ogni qualvolta si vuole aprire una socket SSL, per caricare il Trust Store ed il Key Store, basta creare un `SSLContext`, da inizializzare passando un gestore di ogni chiave e certificato presenti negli store, ricavati a partire rispettivamente dai `KeyManagerFactory` e `TrustManagerFactory`, oltre ad una sorgente di randomness. Dal contesto appena definito, è possibile costruire un oggetto factory per il tipo di Socket che si vuole costruire (`SSLSocket` se costruita per il lato Client e `SSLServerSocket` per il lato Server della connessione) e da quest'ultimo è possibile aprire la specifica socket passando al metodo `createSocket` la sorgente (nel caso in essere è stato sempre usato "localhost") e la specifica porta

estratta dal file di proprietà. Di seguito viene mostrato il codice riguardante la creazione di una delle numerose `SSLServerSocket` presenti nell'applicazione.

```
SSLContext ctx;
try {
    ctx = SSLContext.getInstance("TLS");
    ctx.init(kmf.getKeyManagers(), tmf.getTrustManagers(), new SecureRandom());
    SSLServerSocketFactory factory = ctx.getServerSocketFactory();
    SSLServerSocket sSock = (SSLServerSocket)
factory.createServerSocket(Integer.parseInt(defaultProperties.getProperty("MDTlsSocketRe
ceiveCommitment")));
    sSock.setNeedClientAuth(true);
    while (true) {
        SSLSocket sslSock = (SSLSocket) sSock.accept();

              // Codice relativo alla particolare applicazione della connessione TLS
            }
        }
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

La connessione TLS che viene utilizzata in questa fase è stata impostata sulla porta definita dal campo `MDTlsSocketReceiveCommitment` del `default.properties` file, sul quale l'MD apre una `SSLServerSocket` e resta in attesa della connessione da parte di un utente che abbia creato la relativa `SSLSocket`. È importante notare che per questa connessione è richiesta l'autenticazione anche per il client, e ciò è ottenuto utilizzando `sSock.setNeedAuth(true)` nel `SSLServerSocket`. A valle del two way handshake, il client manda al server il commitment generato al punto precedente, e in seguito chiude la connessione. Il server, dopo aver ricevuto il commitment, lo aggiunge alla HashMap che sta a rappresentare il database di riferimento del MD, e poi torna in attesa di nuove connessioni. Di seguito è riportato il codice di queste attività svolte dall'MD:

```
try (ObjectInputStream in = new ObjectInputStream(sslSock.getInputStream())) {
    byte[] newCom = (byte[]) in.readObject();
    X509Certificate cert = (X509Certificate)
sslSock.getSession().getPeerCertificates()[0];
    synchronized (commitments) {
        commitments.put(cert.getPublicKey(), newCom);
    }
}
```

Questa operazione di store del commitment ha complessità $O(1^*)$ in quanto si sta inserendo in un hash table. (In un'implementazione reale si dovrebbe utilizzare un DB, probabilmente basato su documenti per una maggiore scalabilità.)

Si riporta il codice rilevante utilizzato in questa fase nel main della simulazione:

```
teresa.generateEphemeralKey();
paolo.generateEphemeralKey();
luigi.generateEphemeralKey();
alessio.generateEphemeralKey();
```

4.3.2. Scambio dei contatti

Per scambiare i contatti, come nella fase 2.3.3, vengono utilizzati due thread per i due utenti e una connessione TCP. Il contatto è rappresentato nella classe `ContactMessage`, costituita proprio come la (*2.4*). Fornisce 2 metodi:

- `private void madeSignature(PrivateKey skf, byte[] idByte)` che prende in input la chiave segreta effimera dell'utente che sta comunicando il contatto e l'ID dell'utente a cui lo sta comunicando. Questo metodo esegue la firma come nella *(2.4)* utilizzando `Signature.getInstance("SHA256withECDSA")`.
- `public boolean verifyBTPair(byte[] idByte)` che preso in input l'ID (in particolare quello dell'utente che riceve il contatto) e restituisce True se la firma è valida ed il timestamp è stato generato non più di 30 secondi prima , altrimenti False.

Gli utenti scambiano i contatti utilizzando i metodi `private void sendContact(User u2)` e `private void receiveContact(User u1)`, rispettivamente il primo per l'utente che inizializza la comunicazione ed il secondo per l'utente che accetta il pairing BT. Entrambi vengono richiamati all'interno di `public static void meet2user(User u1, User u2)` che utilizza due thread per far lavorare i due utenti insieme in modo semplice durante la simulazione.

Si riporta il codice rilevante utilizzato in questa fase nel main della simulazione:

```
User.meet2user(teresa, paolo);
User.meet2user(alessio, paolo);
User.meet2user(teresa, luigi);
User.meet2user(paolo, luigi);
```

(In un'implementazione reale si sarebbero chiaramente dovuti effettuare dei pairing BT.)

4.4. Comunicazione Positività

In questa fase viene descritta l'implementazione della fase 2.4.

4.4.1. Riscontro Positivo al Tampone

L'avvenuta positività in seguito al tampone è stata simulata mediante il metodo `public void setPositive(User u)` dell'HA che aggiunge la chiave pubblica dell'utente passato come input ad una lista che simula il database di persone confermate positive al tampone molecolare. Si assume che in questa fase l'utente in questione sia notificato dell'avvenuta positività tramite altri mezzi al di fuori dell'applicazione che si sta descrivendo.

```
public void setPositive(User u) {
    pkPositive.add(u.getPublicKey());
    System.out.println("HA: " + u.getName() + " is positive to Molecular Swab");
}
```

L'aggiunta di un positivo alla lista dell'HA, in un contesto reale, non sarebbe gestita dall'applicazione ma questi sarebbero immagazzinati in un database posseduto dall'HA (e chiaramente non con la PK dell'applicazione ma mediante un loro identificativo come il CF). In questo caso questa operazione ha complessità $O(1)$ in quanto si inserisce in una lista non ordinata.

4.4.2. Richiesta del Token

Se l'utente decide di comunicare la propria positività, allora tramite il metodo `public void communicatePositivity()` apre una `SSLSocket` sulla porta `HATlsSReceiveToken` per connettersi con la `SSLServerSocket` aperta dall'HA sulla medesima porta. A valle dell'handshake, che richiede l'autenticazione dell'utente, il client invia il `PKfCommitment` generato al punto 4.3.1 e poi resta in attesa del token dell'HA (*2.10*). Dopo aver ricevuto i dati dall'utente, l'HA estrae la chiave pubblica dell'utente dal certificato utilizzato per autenticare la connessione, e in seguito verifica il commitment relativo a quella chiave (simulando la lettura sul DB con il metodo `public byte[] requestCommitment(MD md, PublicKey pku)` che estrae la entry corretta dalla HashTable dei commitment mantenuta dal MD). Dopo la ricezione del commitment, l'HA effettua le verifiche dovute all'apertura del commitment, di cui al

punto (*2.9*), e, in caso di successo, genera ed invia un nuovo token. Il token è rappresentato dalla classe `HAToken`, dotata dei metodi per la firma e la verifica del token, oltre che del costruttore per la creazione dello stesso, effettuata così come al punto (*2.10*), che prende in input la chiave pubblica dell'utente, la data di creazione e la coppia PK-SK dell'HA. Di seguito è riportato il codice che esegue quanto descritto all'interno del server dell'HA, in seguito alla connessione del client.

```
// Connessione con l'utente avvenuta
try (ObjectOutputStream out = new ObjectOutputStream(sslSock.getOutputStream());
     ObjectInputStream in = new ObjectInputStream(sslSock.getInputStream())) {

    PkfCommitment commUser = (PkfCommitment) in.readObject();
    X509Certificate cert = (X509Certificate)
sslSock.getSession().getPeerCertificates()[0];

    // Simulazione Connessione con Server MD
    byte[] commMD = requestCommitment(md, cert.getPublicKey());

    // Verifica del commitment
    if (pkPositive.contains(cert.getPublicKey()) &&
            PkfCommitment.openCommit(commUser.r, cert.getPublicKey(), commUser.pkf,
commUser.date, commMD)) {
        // Invio Token
        HAToken token = new HAToken(commUser.pkf, commUser.date, keyPair);
        out.writeObject(token);
    } else {
        System.err.println("Commitment non valido");
    }
}
```

In un'implementazione reale non si sarebbe dovuta ricevere un oggetto come quello di PkfCommitment, che contiene informazioni superflue a quelle necessarie (ma che abbiamo utilizzato per semplificare il protocollo ed in generale il codice avendo meno strutture da considerare). Inoltre, l'HA avrebbe dovuto accedere al DB dell'MD e non ad una sua hash table. Nel caso del codice della simulazione questa operazione ha una complessità di $O(1^*)$. Per quanto riguarda la verifica dell'effettiva positività, in un'applicazione reale l'HA avrebbe dovuto matchare se nell'elenco dei positivi da lei posseduto vi era l'utente corrispondente all'identità contenuta nel certificato. In questo caso la ricerca viene fatta su una lista non ordinata e quindi ha complessità $O(n)$. L'apertura del commitment ha la complessità della funzione di hash utilizzata (in questo caso SHA256). La creazione del token ha invece complessità della funzione di hash utilizzata sommata alla complessità dell'algoritmo di firma digitale (in questo caso SHA256 e RSA).

4.4.3. Comunicazione dei contatti avuti

A valle della ricezione del token, l'utente apre una nuova `SSLSocket`, sulla porta `MDTlsSocketReceiveContacts`, sulla quale l'MD ha attivato una `SSLServerSocket` che non richiede l'autenticazione del client. Dopo l'avvenuta connessione quindi, l'utente invia sia l'`HAToken` ricevuto dall'HA che la lista di `ContactMessage` generata al punto 4.3.2. L'MD quindi verifica il Token ricevuto mediante il metodo apposito della classe `HAToken`, effettua tutte le verifiche sui contatti descritte in fase 2.4 e, se tutte le verifiche sono passate con successo, aggiunge alla lista che simula il database dei contatti a rischio per ogni contatto ricevuto il relativo ID calcolato come descritto al punto (*2.4*). Di seguito è riportato il codice che esegue quanto descritto all'interno del server dell'MD, in seguito alla connessione del client.

```
// Connessione con l'utente Avvenuta
try (ObjectInputStream in = new ObjectInputStream(sslSock.getInputStream())) {
    HAToken token = (HAToken) in.readObject();
    if (!token.verifyToken()) {
```

```
            System.err.println("Token non valido");
            break;
        }
        ArrayList<ContactMessage> c = (ArrayList<ContactMessage>) in.readObject();
        addContactToContactList(token, c);
    }
```

```
private synchronized void addContactToContactList(HAToken token,
ArrayList<ContactMessage> contactList) throws NoSuchAlgorithmException,
SignatureException, InvalidKeyException {
    // Get ID from token
    byte[] idSender = Util.getIdFromPk(token.pkfu);
    // Verification of Contacts as described in 2.4 phase
    ArrayList<DateId> newIdContactMessage = new ArrayList<>();
    for (ContactMessage c : contactList) {
        if (c.verify(idSender) && token.date.equals(c.tsNow.toLocalDate())) {
            newIdContactMessage.add(new DateId(Util.getIdFromPk(c.pkfu1)));
        } else {
            System.out.println("MD : Communication of Fake contacts");
            return;
        }
    }
    // Adding of the contacts
    synchronized (idContactMessage) {
        idContactMessage.addAll(newIdContactMessage);
    }
}
```

In un'implementazione reale gli ID a rischio avrebbero dovuto essere salvati in un database correttamente strutturato in modo da consentire la richiesta di una parte di questo (ovvero soltanto gli id ancora non letti). In questo caso utilizziamo una lista di ByteArray. La complessità della verifica del token dipende dalla funzione di hash e dall'algoritmo di firma (in questo caso SHA256 e RSA). La verifica dei contatti ha come complessità il numero di contatti moltiplicato per la somma delle complessità della funzione di hash utilizzata e dell'algoritmo di firma digitale (in questo caso SHA256 e ECDSA). In questo caso l'aggiunta alla lista ha come complessità il numero di contatti.

Di seguito viene riportato il codice presente nel main relativo a quanto descritto.

```
System.out.println("\n\n Simulate phase 2.4----------------------\n" );
ha.setPositive(teresa);
teresa.communicatePositivity();
```

4.5. Generazione e gestione delle Notifiche

In questa fase viene descritta l'implementazione del punto 2.5.

4.5.1. Ricezione della Notifica

Il metodo `public boolean getNotify()` simula la procedura che i dispositivi degli utenti svolgono periodicamente per verificare se si trovino in condizione di rischio. Attraverso questo metodo, il client apre una `SSLSocket` sulla porta `MDTlsSocketSendRiskId`, sul quale l'MD ha aperto una `SSLServerSocket` che richiede l'autenticazione del client. In seguito all'avvenuta connessione, il server invia al client la lista di tutti gli id a rischio (come definita nel punto 4.4.3). Il client controlla quindi se tra gli ID ricevuti è presente anche il suo, e solo in tal caso genera la notifica di esposizione. Viene infine restituito un booleano che indica se si è a rischio o meno. Di seguito è riportato il codice che esegue quanto descritto nel Client, in seguito alla creazione della `SSLSocket`.

```
cSock.startHandshake();
ArrayList<byte[]> idList;
// Opening of a Stream for the communication with the server
try (ObjectInputStream in = new ObjectInputStream(cSock.getInputStream())) {
    // Download of the list od IDs at risk
    idList = (ArrayList<byte[]>) in.readObject();
}
// Closing communication
cSock.close();
// Check if my ID if present among the IDs at risk
for (byte[] c : idList) {
    if (Arrays.equals(id, c)) {
        System.out.println(name + ": I've received an exposition notification.");
        return true;
    }
}
System.out.println(name + ": I've NOT received an exposition notification.");
return false;
```

In un'applicazione reale, come già anticipato prima, si sarebbe dovuto utilizzare un DB correttamente strutturato per effettuare unicamente la lettura degli ID ancora non letti; invece, la verifica sarebbe rimasta la stessa (chiaramente a meno del ridotto numero di ID da verificare).

4.5.2. Richiesta di un Tampone

Gli Utenti possono utilizzare il metodo `public void bookSwab()` per prenotare un tampone. Questo metodo apre una `SSLSocket` come descritto al punto 4.3.1 sulla porta `HATlsBookSwab` per connettersi alla relativa `SSLServerSocket` generata dall'HA e impostata per eseguire un two-ways handshake. A connessione avvenuta, il client invia il `PkfCommitment` all'HA, che lo utilizza, insieme al commitment relativo alla chiave pubblica estratta dal certificato ricevuto dal MD, per effettuare i controlli descritti nel punto 2.5.1. Se questo controllo viene superato correttamente, allora l'HA controlla se la data della richiesta del tampone è avvenuta entro 24 ore dalla data di registrazione della positività nel sistema, ricavata tramite il metodo `private LocalDateTime getDateTimeofCommunicatedIF(byte[] id )`. Questo metodo, infatti, effettua una ricerca dell'ID passato come input tra gli ID in possesso dell'MD nella lista dei contatti accertati e ne restituisce la data in caso di match. Tale metodo rappresenta una semplificazione applicata in luogo dell'implementazione di una nuova connessione TLS tra i server dell'HA e dell'MD, che invece è stata proposta in fase di progettazione della soluzione. Se il controllo ha riscontro positivo, il tampone viene segnalato come gratuito, altrimenti viene addebitata la relativa somma in BTC, come descritto al punto 2.5.2 . Viene riportato il codice del Thread dell'HA relativo a quanto descritto, tralasciando la creazione della Socket.

```
// Connessione con l'utente avvenuta
try (ObjectInputStream in = new ObjectInputStream(sslSock.getInputStream())) {
    PkfCommitment commUser = (PkfCommitment) in.readObject();
    X509Certificate cert = (X509Certificate)
sslSock.getSession().getPeerCertificates()[0];
    // Connessione con Server MD
    byte[] commMD = requestCommitment(md, cert.getPublicKey());
    // Verification of commitment
    LocalDateTime dtPositivity =
md.getDateTimeOfCommunicatedID(Util.getIdFromPk(commUser.pkf));
    if (PkfCommitment.openCommit(commUser.r, cert.getPublicKey(), commUser.pkf,
commUser.date, commMD)
            && dtPositivity != null) {
        if (dtPositivity.plus(1, ChronoUnit.DAYS).isAfter(LocalDateTime.now())) {
            System.out.println("HA: booked a swab to " +
Util.getIdentityByCertificate(cert).get(2) + " for free");
```

```
        } else {
            System.out.println("HA: booked a swab to " +
Util.getIdentityByCertificate(cert).get(2) + " for 0.001 BTC");
        }
    } else {
        System.err.println("Not Valid Commitment or Data");
    }
}
```

In un'implementazione reale, come anche in un punto precedente, si sarebbe dovuta utilizzare una struttura contenente unicamente i campi necessari. Per semplicità nella simulazione è stata inviata la struttura rappresentante il commitment per semplificare la gestione del protocollo. La complessità dell'operazione di verifica di accesso al tampone gratuito è data dalla somma delle complessità della funzione di hash utilizzata (in questo caso SHA256).

Di seguito è riportato il codice presente nel main relativo a questa fase.

```
System.out.println("\n\nSimulate phase 2.5 ----------------------\n");

if (teresa.getNotify()) {
    teresa.bookSwab();
}
if (paolo.getNotify()) {
    paolo.bookSwab();
}
if (luigi.getNotify()) {
    luigi.bookSwab();
}
if (alessio.getNotify()) {
    alessio.bookSwab();
}
```

5. Conclusioni

Nel documento è stato presentato un modello di applicazione di contact tracing, illustrando gli attori che prendono parte al sistema nonché i possibili avversari con le loro motivazioni e le proprietà che il sistema dovrebbe avere riguardo l'integrità e la confidenzialità. È stata poi discussa una proposta di soluzione, articolata in fasi, che fornisce una visione complessiva del sistema, l'utilizzo di primitive crittografiche e tutti i protocolli utilizzati. In seguito, è stata effettuata un'analisi rispetto alle proprietà ed agli avversari proposti, illustrandone il soddisfacimento in termini di integrità, confidenzialità e trasparenza ed efficienza. Si sono mostrati buoni risultati in integrità, confidenzialità e trasparenza a discapito di un'efficienza non ottimale. Infine, è stata implementata una simulazione dell'intero sistema in linguaggio Java, che utilizza le primitive descritte e simula delle connessioni su rete con dei protocolli semplificati, dimostrandone l'effettivo funzionamento e la realizzabilità.