UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di matematica e informatica

corso di laurea triennale in Informatica

Mario Toscano

DP3T & GAEN: studio di protocolli per il contact tracing e implementazione su microcontrollori ESP32

Tesi di Laurea

Relatore:
Chiar.mo Prof. Mario Di Raimondo

---

anno accademico 2021/2022

Indice

1 Introduzione

In questa tesi viene trattato il protocollo DP3T nelle sue diverse implementazioni: low-cost decentralized proximity tracing, unlinkable decentralized proximity tracing e hybrid decentralized proximity tracing. Questo protocollo nasce dall'esigenza di poter tenere traccia dei contatti tra i vari utenti nel periodo della pandemia di Covid-19. Il Covid-19 è un virus che si è diffuso nella maggior parte del mondo a partire dall'anno 2020. L'elevata contagiosità del virus e l'altrettanto pericolosità ha portato le varie autorità di salute pubblica ad adottare diverse misure per il contenimento dei contagi in modo da poter alleggerire il carico di pazienti nelle sale di riabilitazione le quali presentavano un numero inferiore di posti disponibili rispetto alla richiesta. Tra le tante misure adottate vi è stato anche in molti paesi il periodo di quarantena obbligatoria tranne per determinati motivi tra cui specifici lavori e motivi di salute. Durante questo periodo in cui i contatti erano ridotti al minimo si è iniziato a pensare ad un modo per potere tenere traccia dei contatti in modo anonimo, sicuro e mantenendo la privacy degli utenti. Nasce così DP3T il quale sfrutta le potenzialità dei dispositivi e delle tecnologie moderne e si pone come obiettivo quello di automatizzare il tracciamento dei contatti. Questo protocollo utilizza la tecnologia bluetooth, nello specifico utilizza il Bluetooth Low Energy per individuare i dispositivi all'interno di una specifica area. I tre design sono pensati per soddisfare diverse esigenze come il minor consumo di banda o l'elevata sicurezza e non tracciabilità degli utenti inoltre il protocollo è decentralizzato, ovvero i dati ricevuti sono immagazzinati solo all'interno del dispositivo ed i valori usati dall'utente sono caricati sul server solo nel caso in cui esso risulti positivo al Covid-19. Da questo protocollo Apple e Google elaborano congiuntamente Google Apple Exposure Notification (GAEN) il quale è fortemente ispirato al design Hybrid di DP3T. Questo è tutt'oggi il protocollo usato dalla maggior parte delle applicazioni per il tracciamento dei contatti create per fronteggiare il Covid-19 tra cui l'app usata in italia "Immuni".

Inoltre in questo elaborato viene presentata una implementazione del protocollo Hybrid su un microcontrollore: ESP32 prodotto da Espressif System. Si è cercato di realizzare una versione semplificata del sopra specificato protocollo usando la board TTGO-Camera Plus che contiene al suo interno tutti i componenti necessari per la realizzazione dello stesso.

Nel capitolo 2 sono illustrati i vari strumenti usati nell'elaborato. Vengono presentati alcuni cenni di crittologia evidenziando le differenza tra la crittografia a chiave simmetrica ovvero una metodologia di cifrazione che utilizza una sola chiave per la codifica e la decodifica e la crittografia a chiave asimmetrica, di più recente scoperta la quale invece usa una coppia di chiavi, una per decifrare e l'altra per cifrare. Sono illustrate inoltre le funzioni hash, ovvero un particolare tipo di funzioni che prendono in input una stringa di dimensione arbitraria e ne restituiscono una di una lunghezza fissa, e le funzioni hash crittografiche che sono una particolare famiglia di funzioni hash con proprietà utili nell'ambito della crittografia di cui fanno parte la famiglia di funzioni SHA. In più sono descritte le Pseudo Random Function e le Pseudo Random generator tra cui HMAC e AES. Nell'ultimo paragrafo viene esposta la tecnologia Bluetooth che consente di inviare dati attraverso onde elettromagnetiche.

Nel capitolo 3 vengono illustrati i vari protocolli presentati all'inizio di questa introduzione specificando per ognuno di essi i vantaggi che offre e le specifiche del protocollo. Inoltre è esposto un breve confronto tra il protocollo Hybrid e il GAEN.

Nel quarto capitolo vengono esposte le specifiche della scheda adoperata per la realizzazione del progetto e l'ambiente di sviluppo usato, specificando anche le repository dalle quali è possibile scaricare le librerie usate.

Il capitolo 5 illustra la realizzazione del progetto, specificando le strategie adoperate e i metodi impiegati per eseguire il protocollo. Esso è diviso in diverse sezioni che rappresentano le principali fasi del protocollo. Nella prima sezione vengono specificate le operazioni che vengono eseguite nel momento in cui il dispositivo viene acceso la prima volta dall'utente. A seguire viene evidenziato come vengono eseguiti i passaggi per la creazione dei vari ephimeral identifier, Ephid, che sono inviati tramite la tecnologia bluetooth i cui passaggi sono illustrati nella sezione 3. La sezione 4 raccoglie le strategie e le soluzioni impiegate per la memorizzazione dei dati all'interno della memoria secondaria mentre l'ultima sezione si occupa di illustrare le fasi di collegamento con il server centrale e i metodi impiegati per la connessione.

2 Strumenti

2.1 Funzioni Hash

Le funzioni hash sono un particolare tipo di funzioni che prendono in input una stringa di dati di dimensione arbitraria e ne restituiscono una di dimensione predefinita. Queste funzioni hanno un'ampia varietà di utilizzi in informatica, soprattutto nella crittografia.

Una funzione hash è una funzione $f: A -> B$ dove A è una sequenza di bit di lunghezza arbitraria e B è una sequenza di bit di lunghezza fissa. L'input di tale funzione è definito *messaggio* mentre la sequenza ritornata in output prende il nome di *digest*. Molti hash sono costruiti utilizzando funzioni rapide che lavorano su domini piccoli e finiti e un metodo attraverso il quale queste funzioni possono accettare input di lunghezza arbitraria.

Le funzioni hash crittografiche sono funzioni hash che rispettano delle determinate proprietà, cioè sono una famiglia di funzioni hash con determinate caratteristiche.

Le proprietà fondamentali ideali sono:

- deve identificare univocamente il messaggio, cioè avendo due testi m_1 e m_2 essi, se vengono dati in input alla funzione hash, devono produrre due digest differenti, anche nel caso in cui i due testi siano molto simili;
- deve essere deterministico, dando in input il testo m l'output deve essere sempre lo stesso;
- deve essere semplice e veloce calcolare la funzione hash, indipendentemente dal tipo di input;
- deve essere difficile e quasi impossibile ricavare il testo m da $h = hash(m)$;

2.1.1 Proprietà delle funzioni Hash Crittografiche

Le funzioni hash, per essere definite sicure, devono rispettare le seguenti proprietà:

- **resistenza alla preimmagine** data una funzione hash che restituisce in output il valore h, passando in input il messaggio m, cioè $hash(m) = h$, deve essere computazionalmente difficile risalire a m. Se questa proprietà non è soddisfatta, si dice che la funzione hash è vulnerabile agli attacchi alla preimmagine.
- **resistenza alla seconda preimmagine** data un testo m_1 deve essere computazionalmente difficile trovare un altro testo m_2 tale che $hash(m_1) = hash(m_2)$. Se questa proprietà non è rispettata, si definisce $hash()$ vulnerabile agli attacchi alla seconda preimmagine.
- **resistenza alle collisioni** dati due testi, m_1 e m_2 deve essere difficile che entrambi diano lo stesso output, ovvere se accade che $hash(m_1) = hash(m_2)$ la coppia è definita collisione di hash crittografica. Questa proprietà implica la resistenza alla seconda preimmagine, ma non la resistenza alla preimmagine.

2.2 SHA

SHA (Secure Hash Algorithm) è una famiglia di funzioni hash crittografiche; SHA-1, SHA-224, SHA-256, SHA-384, SHA-512.
La prima versione, SHA-1, venne sviluppata dal NIST (National Institute of Standards and Technology) nel 1993 come FIPS 180 (Federal information processing Standard).
La seconda versione, nata dopo la compromissione di SHA-1 da parte dei crittoanalisti, venne pubblicata come FIPS 180-2, mentre la FIPS 180-1 fu la prima revisione che introdusse il digest di dimensione 160 bit per SHA-1.
La FIPS 180-2 tratta di SHA -256 e SHA-512 le quali sono delle implementazioni di SHA che producono rispettivamente un digest di 256 e 512 bit.
Le funzioni SHA rispettano la medesima struttura, ma variano nei valori usati, nella dimensione del digest prodotto e da alcuni parametri interni della struttura. I passaggi principali sono:

- **padding** dato un messaggio m su cui si vuole applicare la funzione hash, si aggiunge in coda il bit 1 e poi tanti bit 0 tale che la lunghezza del messaggio finale, definita con $|m|$, sia congruo a $512 - 64$ modulo 512, cioè $|m| = 512 - 64 \; mod \; 512$
- **aggiunta dimensione** viene aggiunto al messaggio finale un intero unsigned di 64 bit che contiene la dimensione del messaggio originale.
- **inizializzazione dei buffer** vengono inizializzati i buffer interni della struttura il cui numero e dimensione varia in base alla versione usata.
- **Elaborazione dei blocchi** il messaggio originale viene diviso in blocchi da 512 bit ed ogni blocco viene ancora suddiviso in blocchi, definite parole, la cui dimensione varia in base alla versione utilizzata. Ogni blocco viene elaborato attraverso un ciclo e vengono utilizzati i buffer preimpostati. Per ogni ciclo la parola viene elaborata ed a fine dello stesso i buffer vengono aggiornati ed usati per il ciclo successivo del medesimo blocco. Per ogni iterazione i buffer del blocco vengono aggiunti ai buffer finali. Quando tutti i blocchi sono stati elaborati, i buffer finali vengono concatenati e da essi è tratto il digest finale.

2.2.1 Pseudo-codice SHA-256

$L = lunghezza\ del\ messaggio$

$Se\ L < 2^{64}$

$Aggiungi\ il\ bit\ 1\ al\ messaggio;$

$Aggiungi\ k\ volte\ il\ bit\ 0,$

$con\ \ k\ numero\ intero\ positivo\ tale\ che\ L + 1 + k = 448\ mod\ 512;$

$Aggiungi\ l'intero\ unsigned\ di\ 64\ bit\ alla\ fine\ del\ messaggio;$

La dimensione finale dopo questo processo è un multiplo di 512.
Inizializzazione di sette buffer contenenti i primi 32 bit della parte frazionaria della radice quadrata dei primi otto numeri primi. I valori sono qui riportati in esadecimale.

$H0 = 6a09e667$

$H1 = bb67ae85$

$H2 = 3c6ef372$

$H3 = a54ff53a$

$H4 = 510e527f$

$H5 = 9b05688c$

$H6 = 1f83d9ab$

$H7 = 5be0cd19$

Inoltre viene inizializzata l'array k, contenente i primi 32 bit della parte frazionaria della radice cubica dei primi 64 numeri primi

$k = 0x428a2f98,\ 0x71374491,\ 0xb5c0fbcf,\ 0xe9b5dba5,$
$0x3956c25b,\ 0x59f111f1,\ 0x923f82a4,\ 0xab1c5ed5,$
$0xd807aa98,\ 0x12835b01,\ 0x243185be,\ 0x550c7dc3,$
$0x72be5d74,\ 0x80deb1fe,\ 0x9bdc06a7,\ 0xc19bf174,$
$0xe49b69c1,\ 0xefbe4786,\ 0x0fc19dc6,\ 0x240ca1cc,$
$0x2de92c6f,\ 0x4a7484aa,\ 0x5cb0a9dc,\ 0x76f988da,$
$0x983e5152,\ 0xa831c66d,\ 0xb00327c8,\ 0xbf597fc7,$

$0xc6e00bf3,\ 0xd5a79147,\ 0x06ca6351,\ 0x14292967,$
$0x27b70a85,\ 0x2e1b2138,\ 0x4d2c6dfc,\ 0x53380d13,$
$0x650a7354,\ 0x766a0abb,\ 0x81c2c92e,\ 0x92722c85,$
$0xa2bfe8a1,\ 0xa81a664b,\ 0xc24b8b70,\ 0xc76c51a3,$
$0xd192e819,\ 0xd6990624,\ 0xf40e3585,\ 0x106aa070,$
$0x19a4c116,\ 0x1e376c08,\ 0x2748774c,\ 0x34b0bcb5,$
$0x391c0cb3,\ 0x4ed8aa4a,\ 0x5b9cca4f,\ 0x682e6ff3,$
$0x748f82ee,\ 0x78a5636f,\ 0x84c87814,\ 0x8cc70208,$
$0x90befffa,\ 0xa4506ceb,\ 0xbef9a3f7,\ 0xc67178f2.$

Il messaggio è elaborato dividendolo in blocchi di 512 bit che vengono identificate con $W_1, W_2, \dots, W_{63}$.

$Per\ ogni\ blocco\ W_i\ con\ i = 1, \dots, N\ con\ N = numero\ di\ blocchi$

$Per\ j = 0\ fino\ a\ j = 15$

$W[j] = M_i[j]$

$Per\ j = 16\ fino\ a\ j = 63$

$x = (W[i-15]\ rightrotate\ 7)$
$\oplus\ (W[j-15]\ rightrotate\ 18)$
$\oplus\ (W[j-15]\ rightrotate\ 18)$

$y = (w[i-2]\ rightrotate\ 17)$
$\oplus\ (W[j-2]\ rightrotate\ 19)$
$\oplus\ (w[i-2]\ rightshift\ 10)$

$W[j] = W[j-16] + x + w[i-7] + y$

$Inizializza\ le\ variabili\ di\ lavoro$:

$a = H0$

$b = H1$

$c = H2$

$d = H3$

$e = H4$

$f = H5$

$g = H6$

$h = H7$

$Per\ j = 0\ fino\ a\ j = 63$

$x = (a\ rightrotate\ 2) \oplus (a\ rightrotate\ 13) \oplus (a\ rightrotate\ 22)$

$y = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$

$t_2 = x + y$

$z = (e\ rightrotate\ 6) \oplus (e\ rightrotate\ 11) \oplus (e\ rightrotate\ 25)$

$u = (e \wedge f) \oplus ((\neg e) \wedge g)$

$t_1 = h + z + u + k[i] + W[i]$

$h = g$

$g = f$

$f = e$

$e = d + t_1$

$d = c$

$c = b$

$b = a$

$a = t_1 + t_2$

$H0 = a + H0$

$H1 = b + H1$

$H2 = c + H2$

$H3 = d + H3$

$H4 = e + H4$

$H5 = f + H5$

$H6 = g + H6$

$H7 = g + H7$

$digest = H0 \,||\, H1 \,||\, H2 \,||\, H3 \,||\, H4 \,||\, H5 \,||\, H6 \,||\, H7$

2.3 Pseudo Random Function e Pseudo Random Generator

Una Pseudo Random Generator (PRG) è una funzione che espande una sequenza di bit di valore randomico in una sequenza di lunghezza maggiore ma mantenendo i valori pseudo-randomici ovvero $PRG(k) \rightarrow k'$ dove k è il seed, la lunghezza di k' è maggiore di quella di k e la funzione $PRG(k)$ è computazionalmente indistinguibile da una sequenza di bit scelta randomicamente in maniera uniforme. Computazionalmente indistinguibile vuol dire che, date due distribuzioni X e Y, un avversario in un tempo polinomiale ha una probabilità trascurabile di poter distinguere le due distribuzioni. Diremo quindi che X e Y sono distribuite allo stesso modo.
Una Pseudo Random Function è una famiglia di funzioni il cui output è computazionalmente indistinguibile dalla distribuzione uniforme. La famiglia di funzione è parametrizzata da una chiave k appartenente all'insieme Keys(F) quindi $F : Keys(F) \times Dom(F) \rightarrow Range(F)$ dove generalmente $Keys(F) \rightarrow \{0,1\}^k$, $Dom(F) \rightarrow \{0,1\}^t$ e $Range(F) \rightarrow \{0,1\}^m$ per qualche intero $k, t, m \geq 1$. Preso come riferimento il modello "black box", ovvero un modello in cui tramite una black box è possibile ottenere output da degli input senza conoscere il meccanismo interno della stessa, un avversario passa ad essa degli input. Dopo un numero polinomiale di input l'avversario deve discernere tra una funzione realmente random e la famiglia di funzioni sopra definita. Se gli output delle due funzioni sono computazionalmente indistinguibili possiamo dire che la famiglia di funzioni F è una Pseudo Random Function.

2.4 HMAC

Hmac è una modalità definita in RFC 2104 che fornisce un meccanismo per l'autenticazione e controllo dell'integrità dei messaggi usando una funzione hash crittografica. Il livello di sicurezza fornito è strettamente legato alla funzione hash usata. Sia H la funzione di hash crittografico usata; la potenzialità di Hmac è proprio quella di poter usare funzioni diverse (come MD5 o SHA) e quindi il suo non legarsi ad una specifica funzione.
Si definiscono inoltre con K la chiave segreta, con B la lunghezza in byte dei blocchi in cui suddividiamo il testo M e con L la lunghezza in byte del digest. La lunghezza della chiave K può assumere valori tra 1 e B byte; nel caso in cui essa ha un valore maggiore di B byte dovrà prima essere passata alla funzione hash H e poi successivamente verranno impiegati gli L byte prodotti dalla funzione. Alla chiave K verranno aggiunti in coda $B - lunghezza\ della\ chiave$ byte $0x00$ in modo da avere una chiave la cui lunghezza in byte sia proprio B. Hmac fa inoltre uso di due valori:
$ipad = repat\ 0x36\ B\ times$ e $opad = repeat\ 0x5c\ B\ times$;
Infine si ha:
$Hmac_k(M) = H(K \oplus opad \,||\, H(K \oplus ipad \,||\, M))$.

2.5 AES

AES (Advanced Encryption Standard) è un algoritmo di cifratura a blocchi con chiave simmetrica la cui prima pubblicazione risale al 1998 ed è il successore del DES (Data Encryption Standard). La dimensione del blocco usata dall'algoritmo è di 128 bit mentre la dimensione della chiave può essere di tre taglie diverse: 128, 192 e 256 bit. AES è veloce sia implementato in hardware sia implementato in software, richiede poca memoria e dà un buon livello di sicurezza. L'algoritmo si avvale di matrici di $4\ x\ 4\ bytes$ che prendono il nome di stati. Il primo stato è il messaggio in chiaro. AES è

composto da quattro funzioni principali: SubBytes, ShiftRows, MixColums ed AddRoundKey. La prima funzione consente la non linearità dell'algoritmo e trasforma un byte della matrice stato utilizzando una S-box, ovvero degli strumenti utilizzati nella crittografia a chiave simmetrica che consentono di oscurare le relazioni tra testo in chiaro e il testo cifrato. Esse seguono il principio di confusione di Shannon. La S-box è costruita usando una funzione inversa combinata ad una trasformazione affine.
La funzione ShiftRows, come si evince dal nome, trasla gli elementi di una riga dipendentemente dal numero di riga. Per esempio la prima riga resta invariata (riga numero 0), gli elementi della seconda riga si spostano di una posizione (riga numero 1) etc. Usando questo metodo l'ultima colonna dello stato formerà la diagonale della matrice. MixColumns invece prende i 4 byte di una colonna e li moltiplica modulo $x^4 + 1$ per il polinomio $3x^3 + x^2 + x + 2$. L'ultima funzione, AddRoundKey, esegue uno XOR tra i byte della matrice stato ottenuta applicando le precedenti funzioni e una chiave di sessione derivata dalla chiave primaria attraverso un Key Scheduler ovvero uno strumento che espande una chiave primaria corta in un certo numero di chiavi.
AES può essere eseguito in diverse modalità tra cui la Counter Mode (AES-CTR). Questa modalità è molto usata in quanto è facile da implementare ed è paralizzabile. Inoltre usa solo la funzione "encrypt" di AES, sia per la cifrare che decifrare, rendendo l'implementazione più leggera rispetto alle altre modalità. Però è sconsigliato usare AES-CRT con chiavi statiche in quanto con l'utilizzo dello stesso vettore IV, della stessa nonce e la medesima chiave con testi in chiaro diversi porta alla perdita di sicurezza. L'algoritmo usa la funzione AES per cifrare un blocco di 128 bit ottenuto tramite la concatenazione di 32 bit che rappresentano la nonce, 64 bit del vettore IV e 32 bit del valore counter. Inoltre il testo in chiaro viene anch'esso suddiviso in blocchi di 128 bit. Al primo blocco del testo in chiaro viene applicata una funzione che effettua lo XOR tra esso e il blocco ottenuto tramite la funzione AES. In questo modo si ottiene il primo blocco del testo cifrato. Dopodichè il valore di counter viene incrementato, si riottiene il blocco da 128 bit concatenando i valori come sopra specificato e si riesegue la funzione AES. Viene eseguito lo XOR ma al blocco successivo del testo in chiaro e così via. Quando si sarà giunti all'ultimo blocco del testo in chiaro, il valore della funzione AES verrà troncato alla stessa lunghezza dello stesso. In pseudocodice avremo:

$CRTBLK = nonce \,||\, IV \,||\, counter;$

$$for\ i = 1\ to\ n - 1$$

$$C[i] = M[i] \oplus AES(CTRBLK)$$

$$CTRBLK = CRTBLK + 1$$

$$C[n] = M[n] \oplus trunc(AES(CTRBLK))$$

Dove n è la dimensione del testo in chiaro, M è il testo in chiaro, C è il testo cifrato e $trunc()$ la funzione di troncamento.

2.6 Cenni di Crittografia

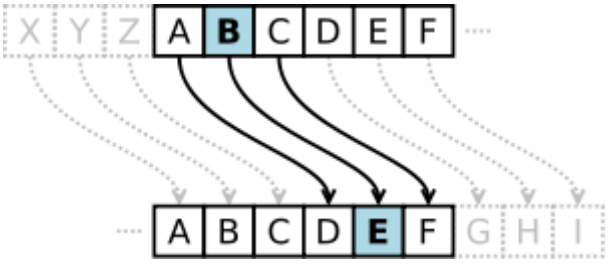
2.6.1 Crittologia

La crittologia è la disciplina che si occupa delle scritture nascoste. Tale disciplina si divide in due ambiti complementari: da una parte abbiamo lo studio e l'ideazione di nuovi metodi sempre più sicuri per occultare il vero significato di determinati simboli, questo ambito prende il nome di crittografia; Dall'altro lato invece avviene lo studio e la decifrazione di testi occulti di cui non si è a conoscenza a priori dei metodi usati: essa prende il nome di crittoanalisi.
I termini tecnici usati nella crittografia sono:

- **testo in chiaro**: testo o file che contiene il vero significato da nascondere;
- **testo cifrato**: testo o file che contiene il significato cifrato;
- **cifratura**: operazione che permette di passare da un testo in chiaro ad uno cifrato;
- **codice**: una regola invariabile per sostituire ad una informazione un oggetto anche di natura diversa;
- **decifratura**: operazione che consente di passare da un testo cifrato ad uno in chiaro all'utente per cui il messaggio era stato cifrato;
- **chiave**: parametro che consente di rendere variabile la cifratura; se la cifratura non è debole basta tenere la chiave privata per mantenere segreto il significato del messaggio;
- **coppia di chiavi**: coppia di parametri usati dalla crittografia asimmetrica; se la cifratura usa uno dei due parametri allora la decifratura dovrà usare l'altro parametro della coppia per decifrare;

2.6.2 Crittografia

La crittografia è una tecnica che consente di cifrare un messaggio rendendolo incomprensibile a tutti tranne a chi è il destinatario del messaggio. Per realizzare tale obiettivo la crittografia si avvale di due operazioni: la cifratura e la codifica.
La cifratura trasforma le lettere individuali di un alfabeto, lavorando quindi non ad un livello semantico ma ad un livello più basso. Un esempio storico di cifratura è il cifrario di Cesare, il quale sostituisce ad una lettera dell'alfabeto la terza lettera successiva alla stessa: la lettera 'A' viene sostituita con la lettera 'D', la 'B' con la 'E' etc.



Schema del cifrario di Cesare

Esso è un sistema di cifratura per sostituzione, in quanto sostituisce una lettera dell'alfabeto con un'altra secondo una regola precisa (la terza lettera seguente alla stessa).
Ci sono anche sistemi di cifratura per trasposizione in cui le lettere nel testo in chiaro vengono trasposte secondo regole prefissate, ottenendo quindi un testo cifrato che è una permutazione (scambio dell'ordine degli elementi di una sequenza) del testo in chiaro.
Un esempio è il cifrario a staccionata in cui il testo in chiaro viene scritto su delle righe ideali diagonalmente verso il basso per poi tornare in alto risalendo una volta che si è arrivati nella riga più in basso e riscendendo una volta arrivati nella riga in alto. Il testo cifrato si ottiene quindi leggendo il testo riga per riga.
Esempio:

avendo il testo in chiaro: "Cifrario a staccionata" usando 4 righe avremo:

C						**I**						**C**						**T**	
	I				**R**		**O**				**A**		**C**				**A**		**A**
		F		**A**				**A**		**T**				**I**		**N**			
			R						**S**						**O**				

Che sarà cifrato in

CICTI ROACA AFAAT INRSO

Inoltre esistono anche sistemi di cifratura che implementano entrambe le soluzioni sopra mostrate che vengono definite cifrature composte. Esse sono generalmente più sicure delle cifrature per sostituzione e per trasposizione.

La codifica, a differenza della cifratura, lavora ad un livello semantico più elevato, che può essere o una frase o una parola.

Entrambi i sistemi si avvalgono di due parti essenziali: un algoritmo e una chiave.

La chiave, insieme al testo in chiaro, viene passata all'algoritmo in input, che darà in output il testo cifrato.

La chiave è una componente fondamentale nella sicurezza di un sistema crittografico. L'algoritmo, per quanto sicuro, è conosciuto anche da eventuali attaccanti, per questo è molto importante che la chiave rimanga segreta. Inoltre un sistema crittografico può avvalersi di più chiavi, ovvero una coppia di chiavi. I sistemi che usano solo una chiave sia per cifrare che per decifrare prendono il nome di sistemi a crittografia simmetrica, mentre quelli che si avvalgono di una coppia di chiavi vengono definiti sistemi a crittografia asimmetrica.

2.7 Crittografia simmetrica

La crittografia simmetrica fino a pochi anni fa era l'unico metodo di crittografia esistente.

Tale metodo presuppone che sia il mittente che il destinatario conoscano la chiave, il che porta a dei problemi di scambio di chiavi, che vengono risolti tramite l'utilizzo della crittografia asimmetrica.

Definiamo:

- ***P*** il testo in chiaro
- ***k*** la chiave
- ***S*** l'algoritmo di crittografia simmetrica
- ***D*** l'algoritmo di decifrazione simmetrica
- ***C*** il messaggio cifrato

Il mittente cifra il messaggio ***P*** tramite l'algoritmo ***S*** passando in input la chiave ***k***. Si ottiene ***C***.

$S(P, k) = C$**.**

Il destinatario riceve il messaggio ***C*** ed utilizza l'algoritmo ***P*** insieme alla chiave ***k*** ottenendo ***P***.

$D(C, k) = P$.

L'eventuale attaccante,anche nel caso in cui intercettasse il messaggio ***C,*** non avrebbe il modo di svelarne il contenuto se non avvalendosi di metodi che richiedono un'elevata potenza di calcolo. Tutto ciò supponendo che la chiave rimanga segreta e che essa venga scambiata in modo sicuro.

La chiave ha una lunghezza fissa nella maggior parte degli algoritmi simmetrici, ma non tutti: per esempio AES adotta una lunghezza di chiave diversa in base al livello di sicurezza che si vuole ottenere e le taglie standard sono: 128, 192 e 256 bit.

Normalmente ogni algoritmo cerca di cifrare una stringa di bit della stessa dimensione della chiave. Tale stringa di bit prende il nome di blocco, la cui dimensione nella maggior parte degli algoritmi è di 128 bit.

In precedenza venivano usati blocchi di 64 bit. Essi sono adesso ritenuti non sicuri in quanto, con l'aumentare della potenza di calcolo , risentono del paradosso del compleanno. Il paradosso afferma che se si riesce ad ottenere un numero di dati uguale alla radice quadrata del numero totale delle combinazioni, allora si avrà la probabilità del 50% che due blocchi si ripetano, consentendo di forzare l'algoritmo. Ovvero in un blocco di 64 bit possiamo avere 2^{64} combinazioni. Segue:

$\sqrt{2^{64}} = 2^{32}$ per il paradosso del compleanno;

$2^{32} \times 8\,B = 32GB$ che è il numero totale di combinazioni per un algoritmo a blocchi di 64 bit. Considerando che con questo quantitativo di informazioni avremo il 50% di probabilità che due blocchi siano uguali, i dati cifrabili con sicurezza diventano poche centinaia di Megabyte;

Invece usando blocchi di dimensione 128 bit (16 byte) avremmo:

$\sqrt{2^{128}} \times 16\,B = 256\,Exabyte$ il quale è numero sufficientemente grande.

2.7.1 Cifratura a blocchi e cifratura a flusso

La differenza tra cifratura a blocchi e cifratura a flusso sta nella dimensione dei dati su cui lavora l'algoritmo.
Nel primo caso l'algoritmo divide il messaggio da cifrare in blocchi di dimensione fissa ed ognuno di questi blocchi viene cifrato con la medesima chiave k. Nella cifratura a flusso un byte del testo in chiaro viene cifrato direttamente in un byte del messaggio cifrato.
Per la cifratura a blocchi esistono tre implementazioni principali: Electronic Code Book, Cipher Block Chaining e Cipher Feed-Back.
Definiti:

- k la chiave;
- B_i l'i-esimo blocco del testo in chiaro ;
- C_i l'i-esima blocco del testo cifrato;

con $i = 1, \ldots, n$ dove n è il numero di blocchi.

2.7.2 Electronic Code Book (ECB)

$S(B_i, k) = C_i$

E' l'implementazione più semplice. Ogni blocco viene elaborato indipendentemente quindi blocchi di testo in chiaro simili daranno in output blocchi di testo cifrato simili.

2.7.3 Cipher Block Chaining (CBC)

$S((IV \oplus B_1), k) = C_1$

$S((C_{i-1} \oplus B_i), k) = C_i$

In questa implementazione si aggiunge un fattore di causalità, ovvero si effettua uno XOR tra il blocco del testo in chiaro i che si intende cifrare e il blocco C_{i-1} precedentemente cifrato. Per cifrare il primo blocco si richiede un vettore di inizializzazione (IV) per cifrare il primo blocco.

2.7.4 Cipher Feed-Back (CFB)

$S(IV, k) \oplus B_1 = C_1$

$S(C_{i-1}, k) \oplus B_i = C_i$

Simile al CBC, ma viene cifrato prima il blocco C_{i-1} con la chiave k e poi viene eseguito uno XOR con il blocco B_i ottenendo così C_i. Anche in questo caso viene usato un vettore di inizializzazione (IV) per la costruzione del primo blocco cifrato.

2.7.5 Counter Mode (CRT)

$S(nonce||counter_i, k) \oplus m_i = C_i$

Il counter mode è un esempio di cifratura a flusso. Tale metodo utilizza due valori: nonce e counter. La nonce (number once) è un numero random o pseudo random il quale viene concatenato con il valore counter che rappresenta lo stato interno del metodo. Il counter normalmente incrementa di una unità il suo valore, ovvero $Counter_i = i$ il che rende l'operazione paralizzabile.

2.8 Crittografia asimmetrica

La crittografia asimmetrica utilizza una coppia di chiavi nei suoi algoritmi di cifratura. Prende il nome di asimmetrica in quanto, per decifrare un testo cifrato tramite una delle due chiavi, servirà l'altra chiave della coppia.
Questo tipo di crittografia è recente, idealizzata nel periodo del 1970 e implementata nel 1977 da Ron Rivest, Adi Shamir e Leonard Adleman che realizzarono l'algoritmo di crittografia RSA. Un sistema equivalente fu sviluppato segretamente dal matematico

inglese Clifford Cocks nel 1973 ma, per vari motivi, Cocks non divulgo il suo lavoro ed esso rimase segreto fino al 1997 quando la costruzione venne declassificata dal governo inglese.
Inoltre nel 1974 viene implementato il protocollo Diffie-Hellman che consente di scambiare una chiave condivisa, senza che i due utenti della comunicazione si siano scambiati informazioni in precedenza. Questo protocollo consente quindi di scambiare una chiave che verrà usata per cifrare tramite crittografia simmetrica, risolvendo il problema dello scambio delle chiavi.

2.8.1 Chiave pubblica e chiave privata

Il principio che sta dietro alla crittografia asimmetrica è quello di avere due tipi di chiave:

- **Chiave pubblica** ovvero una chiave che è non segreta, che è ottenibile da tutti gli utenti della comunicazione (sia "nemici" che destinatari etc..);
- **Chiave privata** cioè una chiave segreta, la cui conoscenza è ristretta al proprietario della stessa;

La chiave pubblica si ottiene con l'utilizzo di problemi matematici che attualmente non ammettono soluzioni particolarmente efficienti come trovare i fattori primi di un numero intero.
In matematica è una operazione semplice moltiplicare tra loro due numeri primi, definiamo:
a un numero primo
b un numero primo diverso da a
essi comporranno la chiave privata;
Il loro prodotto, ovvero:
$c = a \times b$
diventa la chiave pubblica.
Quindi solo c sarà pubblica e risalire ai due fattori primi il cui prodotto da proprio la chiave pubblica è un lavoro oneroso attualmente per a e b sufficientemente grandi.

Il logaritmo discreto e le relazioni con le curve ellittiche sono altri problemi matematici che non ammettono soluzioni particolarmente efficienti attualmente e che vengono usati per la creazione di chiavi pubbliche e private.

2.8.1.1 Usi delle chiavi private e pubbliche

Un testo in chiaro, che viene cifrato utilizzando una delle chiavi di una coppia di chiavi, può essere decifrato solo tramite l'utilizzo dell'altra chiave della coppia.
Quindi un testo cifrato tramite una chiave privata, può essere decifrato solo tramite l'uso della chiave pubblica e viceversa.
Definiamo:

- M il mittente;
- D il destinatario;
- K la chiave pubblica
- k la chiave privata
- P il testo in chiaro
- AS l'algoritmo di cifratura asimmetrica
- AD l'algoritmo di decifratura asimmetrica
- C il testo cifrato

M ottiene la chiave K_D, con la quale computa $AS(P, K_D) = C$. Quindi M invia C, D riceve C e computa $AD(C, k_d) = P$.

Usando questo metodo M è sicuro che D sia l'unico che possa decifrare P se l'algoritmo è sicuro, ma ciò non assicura che il mittente sia proprio M, in quanto il messaggio è soggetto ad attacchi di sniffing, ovvero il pacchetto può essere intercettato e sostituito, cambiandone il contenuto in quanto anche l'attaccante conosce la chiave K_D e quindi, sostituendosi ad M, riesce ad eseguire tutti i passaggi.
Per risolvere questo problema, il problema dell'autenticazione, si può usare lo standard DSA (Digital Signature Algorithm) il quale è uno dei possibili schemi di firma. DSA si avvale di due operazioni: *generazione della firma* e *verifica della firma*.
Per la creazione della firma la funzione prende in input quattro parametri:

- il digest ottenuto passando in input il messaggio da firmare ad una funzione hash predefinita;
- la Global Public Key che è a sua volta composta da tre parametri $p, q\ e\ g$ la quale è uguale per tutti gli utenti della comunicazione;
- la chiave privata del mittente, il cui valore è un numero randomico compreso tra $0\ e\ q$;
- un numero randomico, che varia per ogni messaggio diverso da firmare. Nel caso in cui si utilizzasse un valore uguale per messaggi diversi, è possibile ricostruire la chiave privata.

Questa funzione ritornerà in output due parametri s, r i quali compongono la firma e verranno inviati insieme al messaggio. Il destinatario riceve il messaggio ed esegue l'operazione di verifica firma passando in input alla funzione:

- il digest ottenuto applicando la funzione hash predefinita al messaggio ricevuto.
- la Global Public Key;
- la chiave pubblica del mittente che lo stesso ha generato in precedenza utilizzando la chiave privata;
- i valori s e r ricevuti;

La funzione confronterà il valore v, ottenuto applicando le funzioni di verifica, con il parametro r ricevuto insieme al messaggio. Se i due valori sono uguali, allora la funzione ritornerà 1, ovvero la verifica è andata a buon fine, altrimenti ritornerà 0.
Usando questo schema è ancora possibile per un terzo individuo intercettare il messaggio, cambiarne il contenuto e poi presentarsi come il mittente al destinatario in quanto non è vi è un agente della comunicazione che certifica il possesso delle chiavi.
Per bloccare questo tipo di attacchi viene utilizzata una terza parte, che prende posto di garante, chiamata Certification Authority (CA). Questo ente, tramite una catena di certificazioni, rilascia dei certificati i quali attestano e garantiscono l'autenticità delle chiavi. In questo modo l'attaccante non può ingannare il destinatario se la CA non è compromessa.
Per alto peso computazionale, la crittografia asimmetrica non è usata per scambiare informazioni nelle sessioni: in questi casi è preferibile usare la crittografia simmetrica in quanto computazionalmente veloce e immediata. La crittografia asimmetrica viene applicata per scambiare le chiavi di sessione in modo sicuro per poi instaurare una comunicazione con crittografia simmetrica.

2.9 Bluetooth

Il Bluetooth è una tecnologia che consente a due o più dispositivi di comunicare tramite l'ausilio delle onde radio, quindi senza l'utilizzo di cavi.
Tramite le onde radio è possibile inviare flussi di dati su 79 canali sulla frequenza di banda 2.4 GHz. Supporta la comunicazione punto a punto ed è oggi usata maggiormente per lo streaming di dati su dispositivi quali cuffie, casse, ed anche stampa wireless sui dispositivi mobile. Fu formalizzato nel 1999 dalla SIG (Bluetooth Special Interest Group) e sviluppata dalla Ericsson. Prende il suo nome da Sant'Aroldo, Harald Blàtand, re di Danimarca il quale era soprannominato "Dente Blu" per vari motivi non meglio confermati come il colorarsi i denti di blu per incutere timore agli avversari.
Sant'Aroldo unì i popoli della scandinavia con l'ausilio della religione, scopo simile a quello posto in essere dai suoi realizzatori che volevano unire dispositivi tramite onde radio.
La tecnologia del Bluetooth si basa sulla creazione di una rete di dimensioni limitate chiamata Personal Area Network (PAN) che possono arrivare fino a 100 metri.
Le prime versioni, 1.0 e 1.1, non consentivano una elevata interoperabilità e la velocità massima teorica è di 1 Mbit/s.
La versione 2 introduce la crittografia per garantire l'anonimato e le trasmissioni multicast/broadcast. Con questa versione la velocità massima teorica è di 3 Mbit/s.
Con la versione 3 viene data la possibilità di inviare grandi moli di dati con l'ausilio della connessione High Speed Wi-Fi IEEE 802.11 con la specifica Alternate MAC/PHY: tramite questa specifica si instaura prima una connessione bluetooth; se devono essere inviati dati di grandi dimensioni, il controllo passa alla connessione Wi-Fi e una volta completato il trasferimento il controllo della comunicazione passa al Bluetooth.
La versione 4 introduce il BLE (Bluetooth Low Energy) usato per aggregare dati provenienti da sensori come quelli medici o termometri, usando meno energia a discapito della velocità che raggiunge un massimo teorico di 1 Mbit/s.
La 5° versione aumenta la potenza della connessione, mantenendo gli stessi consumi della versione precedente. BLE raggiunge una velocità massima di 2 Mbit/s. Inoltre da questa versione è possibile usare la tecnologia Bluetooth per localizzare i dispositivi con

la precisione di qualche metro attraverso l'angolo d'arrivo e l'angolo di partenza possibili grazie all'uso di un certo numero di antenne per inviare o ricevere dati.

2.9.1 Bluetooth Low Energy

Questa versione Bluetooth è pensata per le operazioni a bassa potenza ed è maggiormente utilizzata sui piccoli dispositivi quali sensori e microcontrollori.
BLE utilizza la frequenza di banda 2.4 GHz e trasmette su 40 canali. Supporta diversi tipi di topologie di comunicazione: dalla comunicazione punto-punto a quella broadcast e la mesh che consente la creazione di una rete di dispositivi. Tramite BLE è possibile conoscere presenza, distanza e posizione dei dispositivi.
GAP e GATT sono i protocolli di connessioni complementari che lavorano in parallelo nei dispositivi BLE.

2.9.1.1 Generic Access Profile

Generic Access Profile (**GAP**) è il protocollo di comunicazione usato per connettere e scoprire dispositivi bluetooth.
In questo protocollo i dispositivi possono ricoprire due ruoli chiave:

- **centrale** è il dispositivo con potenza di calcolo maggiore, raccoglie le informazioni dalle periferiche e le elabora. Un esempio sono gli smartphone che raccolgono le informazioni da dispositivi quali smartband etc.
- **periferica** sono solitamente dispositivi a bassa potenza che inviano informazioni al dispositivo centrale.

I dispositivi possono rendere pubblici dati tramite due modi: tramite Advertising Data Payload o tramite Scan Response Payload. Sono entrambi payload di 31 bit, ma solo l'Advertising Data è obbligatorio in quanto è quello che viene costantemente inviato al dispositivo centrale. Lo Scan Response è un payload opzionale che viene inviato nel caso in cui la centrale lo richieda. Per questo motivo quest'ultimo viene usato per inviare informazioni più specifiche in caso di necessità.

La periferica invia l'Advertising Data payload in un intervallo di tempo prefissato e al termine dello stesso rinvia nuovamente il payload. Se in questo intervallo di tempo riceve una richiesta di Scan Response, invia il relativo payload se disponibile.
Questa configurazione è utile nel caso in cui una periferica vuole comunicare i suoi dati a più dispositivi: l'Advertising Data payload è ricevuto da tutti questi ultimi che sono nel raggio di ricezione e non vi è la necessità di instaurare una connessione.

2.9.1.2 Generic Attribute Profile

GATT (Generic ATTribute Profile) è un protocollo di comunicazione che consente di trasferire informazioni utilizzando due nuovi concetti: Service e Characteristics.
Questo protocollo entra in esecuzione quando una connessione viene stabilita tra due dispositivi, cioè dopo il protocollo GAP. La connessione stabilità tra una periferica e una centrale è esclusiva, ovvero una periferica può connettersi ad una sola centrale per volta.GATT fa uso di un altro protocollo per i dati che è l'ATT (Attribute Protocol).
Questo protocollo dati viene usato per conservare i dati dei Service e dei Characteristic usando una lookup table in cui è possibile accedere alle entry tramite un ID di 16 bit.
Il GATT Server è il dispositivo centrale che contiene le informazioni di cui la periferica necessita. La singola informazione (come il valore della temperatura) è incapsulata nel concetto di Characteristic. Ogni Characteristic viene identificata attraverso un identificatore di 16 bit o di 128 bit chiamato UUID che è un valore predefinito definito dal SIG ma è anche possibile usare valori personali che sono però riconosciuti solo dalle applicazioni in cui sono definite.
Le Characteristic sono incapsulate in un livello concettuale più alto, il Service, il quale è anch'esso identificato con un UUID. I Service raccolgono Characteristic simili tra loro, ovvero i Service sono specifici per un determinato tema come può essere la misurazione del battito cardiaco. A loro volta i Service vengono raggruppati in Profiles, i quali non sono effettivamente implementati nelle periferiche, ma a livello concettuale sono stati ideati da SIG e servono a raggruppare i Service i quali, combinati insieme, forniscono un determinato servizio.

3 Protocolli per il tracciamento

3.1 Decentralized Privacy-Preserving Proximity Tracing

Decentralized Privacy-Preserving Proximity Tracing (DP3T) è un protocollo nato per avere uno strumento di tracciamento decentralizzato. Utilizza la tecnologia BLE per il tracciamento dei contatti, rilevando pacchetti di Advertising che contengono numeri pseudo casuali i quali identificano l'utente con cui si viene in contatto.
Il protocollo viene sviluppato all'inizio della pandemia da Covid-19 proprio per fornire uno strumento utile per combattere la pandemia ed è stato ideato da:
Scuola politecnica federale di Losanna: Prof. Carmela Troncoso, Prof. Mathias Payer, Prof. Jean-Pierre Hubaux, Prof. Marcel Salathé, Prof. James Larus, Prof. Edouard Bugnion, Dr. Wouter Lueks, Theresa Stadler, Dr. Apostolos Pyrgelis, Dr. Daniele Antonioli, Ludovic Barman, Sylvain Chatel.
ETHZ: Prof. Kenneth Paterson, Prof. Srdjan Capkun, Prof. David Basin, Dr. Jan Beutel, Dennis Jackson.
KU Leuven: Prof. Bart Preneel, Prof. Nigel Smart, Dr. Dave Singelee, Dr. Aysajan Abidin.
TU Delft: Prof. Seda Gürses.
University College London: Dr. Michael Veale.
CISPA Helmholtz Center for Information Security: Prof. Cas Cremers, Prof. Michael Backes, Dr. Nils Ole Tippenhauer.
University of Oxford: Dr. Reuben Binns.
University of Torino / ISI Foundation: Prof. Ciro Cattuto.

Aix Marseille Univ, Université de Toulon, CNRS, CPT: Dr. Alain Barrat.
University of Salerno: Prof. Giuseppe Persiano.
IMDEA Software: Prof. Dario Fiore.
University of Porto (FCUP) and INESC TEC: Prof. Manuel Barbosa.
Stanford University: Prof. Dan Boneh.
Sono state sviluppate diverse versioni di questo protocollo per soddisfare diverse esigenze, ma tutti i design hanno la stessa proprietà fondamentale: i dati sono immagazzinati solo sul dispositivo personale, sia i dati usati per la creazione del proprio identificativo pseudo randomico sia gli identificativi pseudo randomici degli utenti con cui si è entrato in contatto. Le informazioni verranno inviate ad un server centrale solo nel caso in cui l'utente risultasse infetto da Covid-19, ma anche in questo caso viene mantenuto l'anonimato. Quindi DP3T è pensato in modo tale da garantire e rispettare la privacy degli individui e della comunità ed è conforme con le linee guida della General Data Protection Regulation (GDPR), ovvero il regolamento Europeo il quale obiettivo è quello di rafforzare la protezione dei dati personali dei cittadini dell'Unione Europea o dei residenti dell'UE, sia all'interno che all'esterno dei confini dell'UE, restituendo ai cittadini il controllo dei dati personali.
Le diverse versioni nascono per garantire diversi tipi di protezione all'utente ma a discapito dei dati elaborati ed inviati.
I passi principali sono:

- Definito un certo intervallo di tempo, per ognuno di esso viene generato un numero pseudo casuale definito ephid (Ephemeral Identifiers) che viene generato tramite un algoritmo che varia in base alla versione scelta;
- L'ephid viene inserito in un pacchetto Advertising Data e viene inviato tramite il protocollo GAP di BLE;
- Gli utenti partecipanti che sono nell'area di azione del protocollo, area definita dalle autorità sanitarie, rilevano il pacchetto di Advertising e lo immagazzinano in memoria secondaria per un determinato periodo di tempo, anch'esso definito dalla comunità sanitaria;
- Periodicamente i dispositivi si connettono al server centrale a cui richiedono una lista contenente l'Ephid degli utenti partecipanti infetti, i quali sono caricati sul server esplicitamente dall'utente infetto da Covid-19. Questa lista viene elaborata e confrontata con la lista degli ephid conservati in memoria secondaria.

Se vi è una corrispondenza, avviene la notifica del rischio opportunamente calcolato.

3.1.1 Backend

Prima di parlare dei vari design, è doveroso parlare, seppur brevemente, del concetto di server inteso in essi.
Infatti, a dispetto della parola "decentralizzato" presente in tutti i protocolli, questi fanno effettivamente uso di uno o più server di appoggio
Bisogna far notare però che il ruolo del server è quello di raccogliere le chiavi recenti generate da dispositivi in uso da utenti risultati positivi al COVID-19 e renderle disponibili a tutti gli altri dispositivi.
Quindi è bene sottolineare che tale server non contiene alcuna informazione personale sugli utenti positivi e nemmeno informazioni sugli utenti che usano il protocollo.

3.1.2 Low-cost decentralized proximity tracing

Questa versione del protocollo è quella ideata per rispondere alle esigenze di basso consumo dei dispositivi in quanto richiede minor elaborazioni e un consumo di banda inferiore alle altre versioni ma mantenendo delle buone proprietà di privacy.
Per la generazione degli Ephid si usa un numero randomico definito $seed$.

Primo avvio

La prima volta che il protocollo viene eseguito:

- definiamo t il giorno corrente prendendo come riferimento i giorni UTC;
- generiamo un valore randomico e assegniamolo a $seed_t$;

Generazione Ephid

Ogni giorno il dispositivo cambierà il seed per la generazione degli $Ephid$, utilizzando il seed del giorno precedente nel seguente modo:

$seed_t = H(seed_{t-1})$

dove $H()$ è una funzione di Hash crittografico.
$seed_t$ servirà da chiave per la generazione dell'identificativo e quindi dovrà essere conservata in caso di positività dell'utente al Covid-19.
Per la creazione degli ephid è necessario introdurre il concetto di $epoch$.
Un'$epoch$ è il periodo di tempo in cui l'Ephid è inviato tramite BLE e la sua durata è indicata in minuti e rappresentata da L. Infatti da un singola chiave vengono creati molteplici identificatori ognuno dei quali è inviato solo in una determinata $epoch$. Alla fine del proprio periodo di invio, tale identificatore non viene più inviato per il resto della giornata. Alla fine della giornata t (tenendo sempre come riferimento UTC) tutti gli $Ephid$ vengono rigenerati usando la chiave del nuovo giorno.
Per L minuti invieremo un $Ephid$ quindi ne serviranno $n = \frac{24 \cdot 60}{L}$ per ogni giorno i quali devono derivare dalla stessa chiave. Per farlo usiamo una composizione di funzioni come segue:
$Ephid_1 || \ldots || Ephid_n = PRG(PRF(seed_t, "broadcast\ key"))$;
dove:

- PRF è una funzione pseudo-random come per esempio ***HMAC-SHA256***;
- PRG è un generatore pseudo random come per esempio ***AES in counter mode***;

Inoltre "$broadcast\ key$" è una stringa fissa e pubblica.
Questa composizione di due funzioni genera $n \cdot 16$ bytes, i quali vengono divisi in 16 bytes per ottenere gli n $Ephid$ per il giorno corrente.
Per ogni $epoch$ verrà scelto un ephid randomico che sarà inviato come già sopradetto.

3.1.2.1 Memorizzazione

Lo scopo del protocollo è quello di tenere traccia dei contatti che l'utente ha avuto nell'arco della giornata.
Sappiamo già che il sistema, in presenza di altri utenti, riceve dei pacchetti dal protocollo GAP di BLE.
Quello che il sistema deve memorizzare è:

- l'$Ephid$ ricevuto, il quale viene estratto dal pacchetto BLE ricevuto;
- la misura di esposizione;

- il giorno e l'ora nella quale il pacchetto BLE è stato ricevuto;

Inoltre, ogni giorno il dispositivo conserva $Seed_t$ dopo averlo generato. Queste informazioni sono conservate nella memoria secondaria e il periodo di memorizzazione viene deciso tramite le informazioni ottenute dal personale sanitario o dagli esperti. Questo periodo normalmente è il periodo di incubazione del virus, ovvero il periodo in cui esso è presente all'interno dell'utente rendendolo potenzialmente contagioso senza che i sintomi si siano manifestati.

3.1.2.2 Utente positivo

Una volta che l'utente viene riconosciuto come positivo al COVID-19, viene chiesto a quest'ultimo di inviare i dati necessari per notificare agli altri utenti partecipanti la propria positività anonimamente. L'utente invia al server centrale il $seed_t$ dove t corrisponde al primo giorno in cui il paziente viene considerato contagioso.
Dopo aver fatto ciò il dispositivo rimuove $seed_t$ e resetta il protocollo creando un nuovo seed iniziale randomico, cioè riparte dal primo avvio, in modo tale che l'utente non diventi tracciabile.
Infatti, tramite il $seed_t$ che il server manda a tutti i partecipanti, l'attaccante può sia spacciarsi per la vittima in quanto è in grado di generare i $seed_t$ dei giorni successivi e quindi ricreare gli ephid della vittima con le conseguenze del caso.
Gli altri utenti richiedono la lista dei $seed_t$ infetti al server, generano le chiavi fino a generare la chiave del giorno corrente, ricreano gli $ephid$ tramite queste chiavi e controllano se esiste una corrispondenza in memoria secondaria tra quelli memorizzati nei giorni antecedenti, fino ad arrivare a t, e quelli elaborati. Se esiste una corrispondenza questa viene notificata all'utente mostrando anche il grado di rischio elaborato utilizzando l'esposizione memorizzata con l'$Ephid$.

3.1.3 Unlinkable decentralized proximity tracing

Questa versione richiede delle capacità computazionali maggiori rispetto al precedente protocollo e un maggiore uso della banda ma offre delle proprietà di privacy superiori. L'unlinkable utilizza dei filtri specifici, i filtri Cuckoo. Un filtro Cuckoo è una struttura dati probabilistica efficiente in termini di spazio. Esso sfrutta il Cuckoo hashing, ovvero un tipo di hashing che risolve il problema delle collisioni con l'uso di un'altra funzione hash ed una tabella hash associata a tale funzione. Se, inserendo l'elemento x, avviene una collisione, cioè è già presente un elemento y nella posizione $i = h_1(x)$, l'elemento y viene eliminato da questa posizione e viene inserito nella posizione $j = h_2(y)$. Se in questa nuova posizione avviene un'altra collisione, si ripete il processo eliminando l'elemento nella posizione j e calcolando il suo indice tramite h_2. Si procede così fin quando non avvengono più collisioni. L'elemento x viene inserito nella posizione i. I filtri usano il Cuckoo hashing usando come funzioni hash le seguenti funzioni:

$i = h_1(x) = hash(x)$

$j = h_2(x) = h_1(x) \oplus hash(fingerprint(x))$

dove $fingerprint()$ è una funzione che mappa un dato di una lunghezza arbitraria in una sequenza di bit di lunghezza minore.

Usando questo tipo di funzioni le operazioni di ricerca e di cancellazione hanno complessità $O(1)$.

L'Unlinkable utilizza come riferimento temporale uno starting point, in comune tra tutti i dispositivi che lo utilizzano. Tramite questo riferimento vengono create le varie $epoch\ i$.

Per ogni $epoch\ i$ il dispositivo genera un numero casuale di 32 byte che denominiamo con $seed_i$ dal quale creiamo l'ephid come segue:

$$Ephid_i = LEFTMOST128(H(seed_i))$$

dove :

- H è una funzione hash crittografica;

- $LEFTMOST128()$è una funzione che ritorna in output i 128 bit più significativi della variabile passata in input;

L'$Ephid_i$ così generato verrà inviato per l'intera epoca i tramite il protocollo GAP da BLE.

Al termine dell'epoca i, ovvero dopo che sono passati L minuti dall'inizio dell'epoca, si passa all'epoca $i + 1$ creando l'$Ephid_{i+1}$ come sopra descritto.

Bisogna notare come in questo design non vi è una chiave giornaliera, ma una chiave randomica per ogni singola epoca. Questo rafforza la non tracciabilità dell'utente in quanto non è possibile collegare insieme i vari $Ephid$ essendo tutti indipendenti.

3.1.3.1 Memorizzazione

A differenza del Low-Cost, la memorizzazione è un parte onerosa del protocollo. Dovremmo memorizzare per ogni epoca l'$Ephid$ generato, in quanto non abbiamo una chiave giornaliera ma esso varia per ogni epoca.

Inoltre, quando riceviamo il pacchetto dal protocollo GAP di BLE dobbiamo memorizzare le seguenti informazioni in memoria secondaria:

- $H(Ephid \mid\mid i)$ cioè memorizziamo una elaborazione di $Ephid$ ottenuta concatenandolo a i , ovvero l'epoca in cui è stato ricevuto, per poi passare la stringa così ottenuta ad una funzione di hash crittografico H.
- la misura di esposizione;
- l'epoca i nella quale il pacchetto BLE è stato ricevuto.

3.1.3.2 Utente positivo

Una volta che l'utente viene certificato come infetto, esso ha la possibilità di scegliere l'insieme delle epoche i di cui vuole condividere l'$Ephid$. Questa scelta garantisce una miglior privacy rispetto al design Low-Cost. Le epoche scelte vengono inserite in un insieme che chiameremo ***I***. Il dispositivo crea periodicamente, per esempio ogni 2 ore) un filtro Cuckoo in cui inserisce la coppia $(seed_i, i)$ di tutte le ***i*** facenti parte dell'insieme ***I***. I dati però non sono inseriti in chiaro nel filtro ma vengono elaborati nel

seguente modo:

$H(\ LEFTMOST128(\ H(seed_i)\ )\ ||\ i\ )$

Ricordiamo che $Ephid\ =\ LEFTMOST128(H(seed_i))$ di conseguenza il server inserisce nel filtro F la stringa $H(Ephid_i||i)$ che è la stessa stringa che il dispositivo memorizza.

Periodicamente il dispositivo scarica il nuovo filtro F dal server centrale, così facendo l'utente comune non vedrà mai i singoli $Ephid$ ma una struttura dati contenente delle stringhe hash. Successivamente il dispositivo farà delle query alla struttura dati, passando in input le stringhe memorizzate nel periodo a cui il filtro fa riferimento. Se vi è una corrispondenza all'utente proprietario verrà notificato il rischio.

I filtri Cuckoo hanno una bassa, ma non equivalente a zero, probabilità che si verifichi un falso positivo.

3.1.4 Hybrid decentralized proximity tracing

Questo design è stato ideato unendo i sopradescritti protocolli ed è quello più simile al protocollo GAEN di Apple/Google, da cui le due società si sono fortemente ispirate.

Anche in questa versione esiste uno starting point condiviso da tutti i dispositivi, dalla quale vengono codificate le epoche $\boldsymbol{i}$. Facendo riferimento al Low-Cost, viene suddiviso il tempo tramite delle finestre temporali w, le quali sono una composizione di epoche i di L minuti.

La dimensione di w è quindi un multiplo di L e il suo valore è determinato dal numero di epoche che essa comprende: può assumere come valore minimo L (ciò ci riconduce al protocollo Unlinkable), oppure l'intera giornata (come nel protocollo Low-Cost).

Per ogni finestra temporale w viene generato un seed randomico, $seed_w$ di 16 byte che assume il ruolo di chiave per la generazione degli $Ephid$ relativi a w.

Quindi tali identificativi sono dipendenti da una chiave come nel primo protocollo, ma per periodo ridotto (normalmente si tende a scegliere come dimensione di w 2 o 4 ore).

Utilizzando $seed_w$ come chiave nel periodo w il dispositivo genererà $n\ Ephid$ dove n è il numero di epoche che contiene w cioè:

$n = \frac{|w|}{L}$ dove $|w| = k * L \, con \, k \in \{1, 2,..., \frac{24*60}{L}\} \, ed \, L \, intero \, positivo;$.

Gli ephid sono generati come segue:

$Ephid_{w,1}||Ephid_{w,2}|| \ldots || Ephid_{w,n} = PRG(PRF(seed_w, "DP3T - HYBRID"));$

Dove:

- PRF è una funzione pseudo-random (es. HMAC-256);
- "$DP3T - HYBRID$" è una stringa fissa e pubblica;
- PRG è un generatore pseudo random (es. AES in counter mode);

Tale combinazione di funzioni produce una stringa di $n \times 16 \, bytes$ che verrà suddivisa in modo da creare $n \, Ephid$.

Ottenuti gli $Ephid$ ne verrà preso uno randomicamente e verrà inviato per L minuti, cioè per tutta l'epoca. Alla fine della finestra temporale w si rigenera il $seed$ randomicamente e si ripete il processo per $w + 1$.

3.1.4.1 Memorizzazione

Si ritorna a conservare l'$Ephid$ in chiaro come nella prima versione consentendo un minor uso di memoria, ma i $seed$ che dovranno essere salvati sono in numero maggiore in quanto è necessario tenere in memoria quelli degli ultimi x giorni (normalmente 14 giorni) secondo le linee guida degli esperti.

Quello che il dispositivo memorizzerà sara:

- l'$Ephid$ ricevuto, il quale viene estratto dal pacchetto BLE ricevuto;
- la misura di esposizione;
- la finestra w nella quale l'$Ephid$ è stato ricevuto;

3.1.4.2 Utente positivo

Nel caso in cui l'utente risulti positivo al COVID-19, esso invierà al server i $seed_w$ ritenuti rilevanti nel periodo contagioso: se il dispositivo durante la finestra temporale w non ha ricevuto $Ephid$ o comunque ritiene che il periodo di esposizione non è

sufficiente a creare una situazione di pericolo, il $seed_w$ non verrà considerato e quindi non inoltrato al server centrale. Inoltre, come nel design Unlinkable, l'utente può selezionare le finestre w di cui vuol far conoscere la chiave. L'insieme dei w così selezionati viene definito W. Il dispositivo quindi inoltrerà al server centrale la coppia $(seed_w, w)$ per tutti i $w \in W$. Tutti gli altri dispositivi scaricheranno questa lista del server, calcoleranno l'insieme degli $Ephid$ relativi a w e cercheranno una corrispondenza nella memoria secondaria tenendo come riferimento temporale la finestra relativa agli identificatori. Se vi è una corrispondenza, l'utente viene notificato con relativa misura di rischio.

3.2 Google Apple Exposure Notification

Google Apple Exposure Notification (GAEN) è il protocollo creato da Apple e Google fortemente inspirato all'hybrid decentralized proximity tracing del protocollo DP-3T come già accennato precedentemente.
GAEN prevede non solo l'invio di un identificativo per tracciare gli utenti con cui si viene in contatto (come lo era l'$Ephid$ per DP-3T) ma anche dei metadati. Ci riferiamo all'identificativo con RPI (Rolling Proximity Identifiers) mentre ci riferiremo ai metadati con il termine AEM (Associated Encrypted Metadata).
Anche in questo caso si usa uno starting point temporale, condiviso da tutti i dispositivi, in questo caso l'Unix Epoch Time.
Per la creazione di RPI e AMK si utilizza una chiave qui definita con TEK (Temporary Exposure Key) e questa chiave ha una validità di 24 ore, ovvero per un periodo di tempo chiamato $TEKRollingPeriod$ che è il numero di epoche per cui la chiave ha validità. Considerando che in questo protocollo un'epoca dura 10 minuti, quindi $L = 10$ si avrà che $TEKRollingPeriod = \frac{24*60}{L} = 144$.
Ogni epoca è identificata attraverso un numero intero utilizzando lo starting point temporale. Questo identificativo prende il nome di ENIN (ENInterval Number) ed è calcolato attraverso la seguente funzione, passando in input il timestamp in Unix Epoch Time.

$ENIN(timestamp) = \frac{timestamp}{60 \times 10}$;

Per generare la chiave TEK il protocollo prende come riferimento la prima epoca del giorno, ottenuta tramite ENIN. Questa particolare epoca viene definita i ed è così calcolata:

$i = floor(\frac{ENIN(time\ at\ key\ generation)}{TEKRollingPeriod}) \times TEKRollingPeriod$;

La chiave TEK_i invece viene generata utilizzando un generatore di numeri randomici crittografico, che restituisce in output una chiave di 16 byte.

$TEK_i = CRNG(16)$;

La coppia (TEK_i, i) viene memorizzata per un determinato periodo di tempo definito dalle autorità sanitarie, di solito 14 giorni, e poi viene eliminata. Alla fine del TEKRollingPeriod, viene generata una nuova chiave TEK ripetendo i passaggi sopra mostrati.

Una volta ottenuta la chiave, essa viene utilizzata per trovare i due valori che saranno poi inviati dal protocollo GAP del BLE. Il protocollo prevede il passaggio attraverso due chiavi intermedie: $RPIK$ (Rolling Proximity Identifiers Key) per la generazione di RPI e $AEMK$ (Associated Encrypted Metadata Key) per la generazione di AEM.

Entrambi vengono generati tramite la funzione $HKDF(key, salt, info, outputLength)$ definita in IETF RFC 5869 la quale estrae una chiave pseudo randomica utilizzando una funzione hash (in questo caso SHA-256) su un "salt" opzionale che funziona da chiave e una qualsiasi chiave considerata debole che funziona da dato: la funzione darà in output una chiave più robusta e di una lunghezza predefinita.

In questo caso avremo:

$RPIK = HKDF(TEK_i, NULL, UTF8("EN - RPIK"), 16)$;

$AEMK = HKDF(TEK_i, NULL, UTF8("EN - AEMK"), 16)$;

Infine, ottenute queste chiavi possiamo generare i valori da inviare. Il valore di RPI varia al cambiare dell'epoca in cui viene generato, quindi ogni qualvolta l'epoca finisce se essa è ancora nel periodo di validità di TEK_i:

- j è l'Unix Epoch Time nel quale avviene il cambio o più in generale l'istante di tempo in cui si decide di cambiare RPI;
- $ENIN_J = ENIN(J)$;

- creiamo $PaddedData$ che è una sequenza di 16 bytes così composta
  - $PaddedData_j[0 \dots 5] = UTF8("EN - RPI");$
  - $PaddedData_j[6 \dots 11] = 0x000000000000;$
  - $PaddedData_j[12 \dots 15] = ENIN_J;$
- $RPI_{i,j} = AES_{128}(RPIK_i, PaddedData);$
- $AEM_{i,j} = AES_{128} - CRT(AEMK_i, RPI_{i,j}, Metadata);$

Con l'utilizzo di queste funzioni, i metadati non possono essere decifrati se prima non si è a conoscenza della TEK relativa, ovvero fino a quando l'utente non decide di caricare le proprie chiavi perchè affetto da Covid-19.
$RPI_{i,j}$ e $AEM_{i,j}$ hanno una dimensione totale di 20 byte, 16 e 4 byte rispettivamente.

Utente positivo

Quando un utente risulta positivo al Covid-19, esso invia l'insieme delle coppie (TEK_i, i) al server centrale. Verranno inviate le coppie inerenti al periodo in cui il paziente è entrato in contatto con altri utenti partecipanti al protocollo, ovvero le epoche i in cui sono stati memorizzati dei dati ricevuti dal protocollo GAP. Inoltre non si inoltrano le chiavi create 14 giorni prima dell'avvenuta diagnosi e se le chiavi non lasceranno mai il dispositivo fin quando l'utente non risulti positivo al Covid-19.
Gli altri utenti scaricano la lista delle coppie ed usando questa combinazione per ricreare gli RPI e confrontarli con quelli memorizzati, in ricerca di una corrispondenza. Inoltre viene considerata una tolleranza di 2 ore di differenza temporale da quando viene supposto che RPI è stato inviato a quando invece è stato effettivamente scansionato.

3.3 Confronto tra GAEN e hybrid decentralized proximity tracing

Come già esposto prima GAEN è fortemente ispirato dal design hybrid, però presenta alcune differenze rispetto a quest'ultimo. La soluzione di Apple e Google utilizza una finestra temporale di validità della chiave TEK di un giorno. Questa particolarità lo riconduce al protocollo Low Cost di DP3T generando un problema di correlazione tra il valore RPI e la sua chiave: nell'eventualità in cui un potenziale attaccante riesce, anche a posteriori, ad ottenere il valore TEK è in grado di tracciare tutti i contatti che ha avuto l'utente nell'intera giornata. Usando invece una finestra temporale di validità inferiore invece è possibile limitare il tracciamento per un periodo di tempo nettamente ridotto. Questa scelta però comporta un consumo inferiore di banda per GAEN sia per il download dei dati sia per l'upload. Infatti nel protocollo di Apple/Google l'utente positivo invierà un solo TEK per giorno fino ad un massimo di 14 TEK relative al periodo in cui il virus risulta in incubazione rendendo il paziente potenzialmente contagioso, mentre nel design di DP3T vengono inviati fino a 12 $seed$ per giorno, nel caso in cui si utilizzi una finestra temporale di 2 ore, che arrivano fino a 168 considerando sempre lo stesso periodo di contagiosità. Si noti che anche il server dovrà immagazzinare molti più dati rispetto al protocollo GAEN, richiedendo dunque un maggior uso di memoria.
Un'altra importante considerazione da fare è lo scopo perseguito dai due protocolli: DP3T si pone come unico obiettivo quello di tracciare i contatti in modo in modo sicuro e rispettando la privacy degli utenti, GAEN aggiunge a questo obiettivo anche quello di raccogliere informazioni statistiche che poi vengono inviate esclusivamente all'autorità di salute pubblica attraverso metodi sicuri ed anonimi. L'utente può anche decidere di non condividere tali informazioni, cioè di non partecipare al programma di raccolta informazioni che prende il nome di Exposure Notification Privacy-preserving Analytics (ENPA).

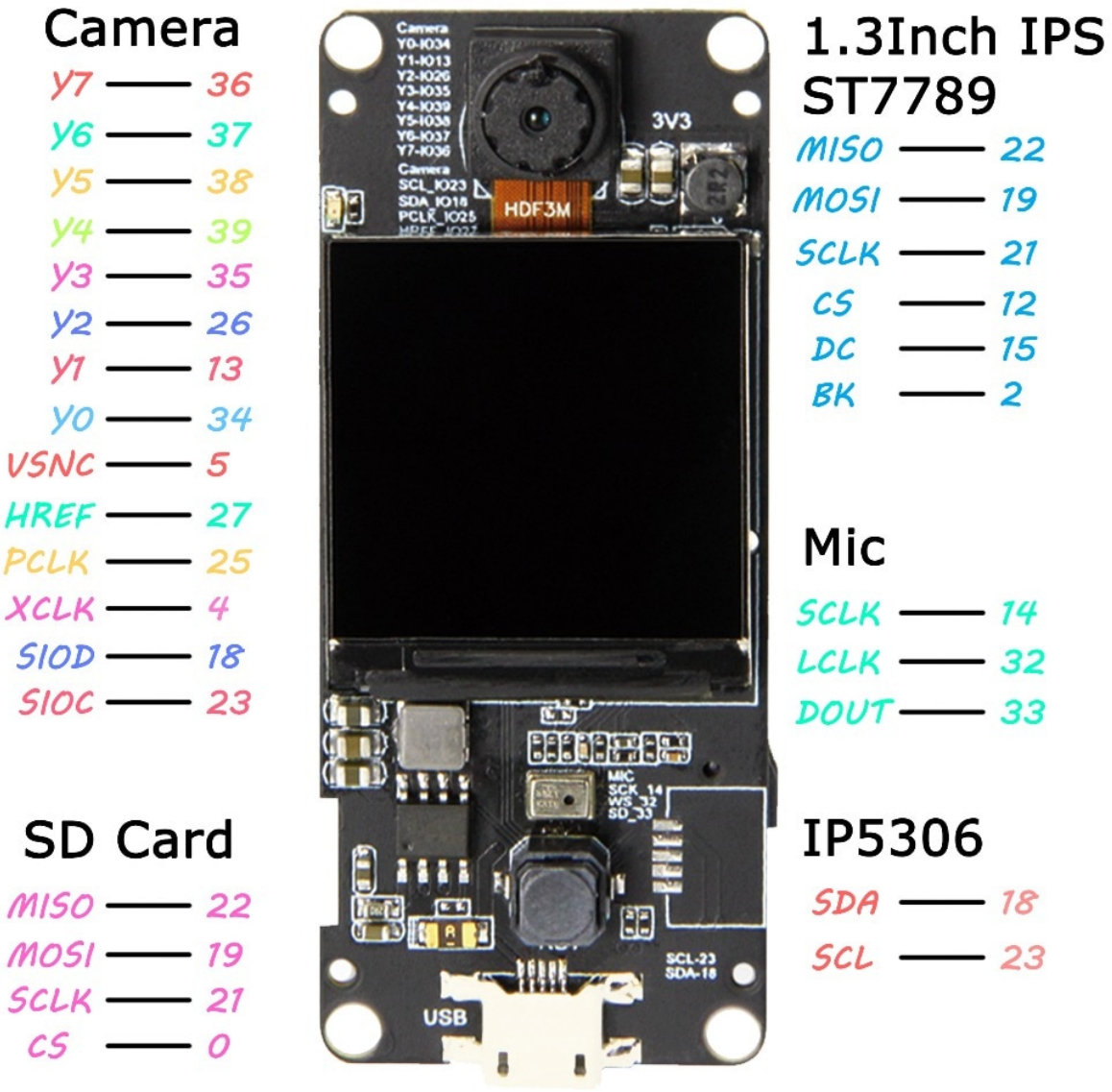
4 Descrizione dell'ambiente di sviluppo

4.1 Scheda

Per la realizzazione del progetto ho usato una scheda TTGO-Camera Plus che monta il chipset ESP32.
ESP32 è una serie di SoC (System-on-a-Chip) low-cost e low-power realizzata da Espressif systems.
Un Soc (System-on-a-chip), letteralmente "sistema su un circuito integrato", è un circuito stampato che include tutti o più comuni componenti di un computer o di altri sistemi elettronici. Tali componenti sono, ad esempio, la CPU, l'interfaccia di memoria o l'interfaccia di memoria secondaria. La differenza con la tradizionale architettura del PC basata sulla scheda madre è proprio quella di integrare questi strumenti su un unico circuito integrato, mentre le schede madri tendono a ospitare e collegare tali strumenti consentendone anche la sostituzione e riparazione.
Questa scheda è prodotta da **LILYGO®.**
Ecco i dettagli della scheda:

Chipset	**ESPRESSIF-ESP32-DOWDQ6 240MHz Xtensa® dual-core 32-bit LX6 microprocessor**
Flash	**QSPI 4MB flash / 8MB SRAM**
SRAM	**520kB SRAM + 8MB External SPRAM**
Button	**Bottone Reset**

Display	**IPS Panel ST7789 – 1.3 Inch**
USB to TTL	**CP2104**
Camera	**OV2640 2 Megapixels**
Microfono	**MSM261S4030H0**
SD Card	**MicroSD card slot**
Operating voltage	**2.3V – 3.6V**
Operating current	**~160mA**
Bluetooth	**bluetooth v4.2BR/EDR and BLE standard**



TTGO-Camera Plus

Wi–Fi Bluetooth Board
Pin Diagram

Si è optato per utilizzare tale tipo di scheda in quanto essa ha già nella sua board tutto quello che serve per il progetto: la tecnologia BLE, la principale funzione richiesta per progetto, la possibilità di un monitor per comunicare in maniera diretta con l'utente, il modulo Wireless per la connessione ad internet, una camera da poter utilizzare per apprendere informazioni e la possibilità di usufruire di una scheda microSD tramite lo slot SD.

4.2 Ambiente di Sviluppo

L'ambiente di sviluppo usato è **Arduino IDE v.1.8.16**.
Arduino IDE è un IDE (integrated development environment) multipiattaforma in Java e deriva dall'IDE creato per il linguaggio di programmazione Processing e per il progetto Wiring. Di tale progetto eredità la libreria C/C++ chiamata Wiring che consente di rendere molto più semplici le operazioni di I/O. Il programmi scritti tramite questo IDE sono scritti in un linguaggio derivato dal C/C++ ed all'utilizzatore viene richiesto di definire due funzioni:

- **void setup()** funzione che viene invocata una volta sola all'inizio del programma, viene usata per impostare i settaggi iniziali del progetto;
- **void loop()** funzione che viene invocata ripetutamente la cui esecuzione viene bloccata quando viene tolta l'alimentazione della scheda;

Anche se l'ambiente di sviluppo è concepito per programmare sulle schede Arduino, tramite il gestore delle schede fornito dello stesso è possibile programmare qualsiasi scheda, a patto che esistano pacchetti librerie per quest'ultima.
La libreria per ESP32 è creata dalla stessa Espressif System ed è facilmente installabile tramite lo strumento sopracitato cercando la parola chiave "ESP32" la cui repository è la seguente:
https://github.com/espressif/arduino-esp32
Si è usata la versione **1.0.6** di tale collezione di librerie.
In particolare usufruiamo delle seguenti librerie incluse nella repository:

- `ESP32_BLE_Arduino`: consente di usufruire delle funzionalità BLE della scheda; Inclusa nel pacchetto sopra citato e usata nel file "modulo_BT.cpp" del progetto. repo: https://github.com/nkolban/ESP32_BLE_Arduino

- `ESPTrueRandom`: libreria le cui funzioni generano numeri random utilizzando le funzioni di ESP32 per la generazioni di informazioni randomiche. Usata nel file “modulo_Key.cpp”. repo: https://github.com/sinricpro/ESPTrueRandom
- `mbedtls`: in particolare `mbedtls\aes` e `mbedtls\md`. Esse sono utili per implementare le funzioni di crittografia del design scelto.Sono già incluse nell’IDE;

5 Realizzazione del progetto

Nel mio progetto ho deciso di implementare il design “Hybrid Decentralized Proximity Tracing”. Si è scelto come valore di L 4 ore, in modo da non appesantire troppo l’applicazione con pesanti elaborazioni vicine temporalmente. L'obiettivo principale del progetto è l’implementazione del protocollo in un sistema embedded standalone che potesse in alcuni casi essere un sostituto valido dei dispositivi tuttora adoperati. Il progetto presenta una versione semplificata del protocollo ed è sviluppato in buona parte in linguaggio C e C++.

5.1 Primo avvio

Il sistema richiede una configurazione iniziale che avviene al momento del primo avvio. Esso attenderà una connessione ad internet necessaria per avviare l’intero protocollo e per usufruire del servizio NTP (Network Time Protocol) il quale consente di usare uno starting point uguale per tutti i dispositivi, in modo da poter ottenere dei timestamp validi per le varie funzione del progetto. Il protocollo necessita di una connessione alla rete Internet per molti suoi aspetti tra cui la più importante è la connessione al server centrale. La scheda presenta un modulo Wifi con cui è possibile connettersi ai vari access point che garantiscono un accesso ad internet. Come è noto per connettersi ad un access point è necessario conoscere SSID e password, due parametri che non hanno un valore standard ma sono personali e configurabili dai possessori degli stessi. È impensabile configurare tali parametri in fase di progettazione ma è necessario che siano gli utenti ad inserirli in quanto sono loro che ne possiedono la conoscenza.
Per implementare questa funzione, l’input da parte dell’utente di SSID e password, si è evidenziato il problema dell'inserimento dei dati in input: la scheda non presenta una tastiera, strumento più accessibile all’utenza o uno schermo touch; l’unico pulsante fisico presente è il pulsante di riavvio la cui unica funzione è quella appunto di riavviare la scheda ed è comunque uno strumento poco pratico per l’inserimento di questo tipo di dati. Per risolvere questo problema, si è deciso di impiegare la camera posteriore come

strumento di “Lettura QR Code", infatti tutti i moderni modem o access point presentano sulla loro parte posteriore un QR Code in cui sono codificati i dati di autenticazione di default mentre, nel caso in cui tali dati sono stati modificati dal possessore del dispositivo, è possibile ottenere un QR Code una volta conosciuti SSID e Password tramite lo strumento di condivisione password presente sulla maggior parte dei moderni dispositivi smartphone. Quindi l’applicazione rimarrà in attesa di un QR Code prima di avviare l’intero processo e una volta ottenuto l’accesso ad internet il dispositivo è in grado di funzionare e può generare chiavi o rilevare e immagazzinare ephid senza l’ausilio di essa. La connessione sarà richiesta in seguito solo per le operazioni di verifica del rischio.
Per la connessione ad internet è stata utilizzata la libreria “`WiFi.h`”. Il modulo WiFi di ESP32 ha tre diverse modalità: WiFi station la quale viene usata per connettersi ad un access point, Access Point e Access Point Station. Per specificare la modalità si usa la funzione `Wifi.mode()` e nel nostro caso passiamo come parametro la stringa “`WIFI_STA`” per usufruire della prima modalità. Tramite la funzione “`WiFi.begin(SSID,Password)`” instauriamo una connessione all’access point che ha accesso ad internet e usando “`WiFi.status()`” siamo in grado di conoscere lo stato della connessione.

5.2 Generazione degli Ephid

Una parte fondamentale del progetto è quella di generare gli Ephid. Per far ciò si è preso come riferimento l’implementazione fornita nella repository ufficiale di DP3T.
È stata quindi creata la classe “`Protocol`”, dichiarata all’interno del file “`Modulo_Key.h`” e definita nel file “`Modulo_Key.cpp`” la quale raccoglie i principali passaggi per la creazione degli ephid rispettando il protocollo Hybrid decentralized proximity tracing. Il primo passo da eseguire è quello di creare un seed randomico di 16 byte tramite il metodo della classe “`generate_Seed()`” la quale utilizza come generatore di numeri randomici la funzione “`ESPTrueRandom.random()`”, che sfrutta le funzionalità della scheda per la generazione di numeri randomici. Ogni chiamata a questa funzione restituisce in output

un byte che rappresenta un intero il cui valore varia randomicamente tra 0 e 255. Successivamente i singoli byte vengono aggregati, andando a formare una sequenza di 16 byte che formerà il seed. Dopo aver ottenuto il seed, è necessario applicare le funzioni PRF e PRG. Si è scelto di utilizzare come Pseudo Random Function Hmac, e come Pseudo Random Generator la funzione AES in counter mode; entrambi sono implementati come metodi di classe. Hmac è implementato nella funzione omonima "`Hmac`" la quale prende in input un seed e restituisce in output un digest di 32 byte. Al suo interno la funzione utilizza i metodi della libreria "`mbedtls/md`" per la creazione del digest.

```
byte* Protocol::Hmac(char* seed){
  char *payload="DP3T-HYBRID";
  byte digest[32];
  mbedtls_md_context_t ctx;
  mbedtls_md_type_t md_type = MBEDTLS_MD_SHA256;
  const size_t payloadLength = strlen(payload);
  const size_t keyLength = 16;
  mbedtls_md_init(&ctx);
  mbedtls_md_setup(&ctx, mbedtls_md_info_from_type(md_type), 1);
  mbedtls_md_hmac_starts(&ctx, (const unsigned char *) seed, keyLength);
  mbedtls_md_hmac_update(&ctx, (const unsigned char *) payload, payloadLength);
  mbedtls_md_hmac_finish(&ctx, digest);
  mbedtls_md_free(&ctx);
  return digest;
}
```

Il payload viene inizializzato con una stringa pubblica, nel nostro caso "DP3T-HYBRID". Il seed generato precedentemente viene usato come chiave della funzione.

Il digest creato viene infine usato come chiave per la funzione AES definita come metodo della classe con il nome "`aes_crt()`". Come per Hmac la libreria usata fa parte della libreria mbedtls nello specifico in "`mbedtls/aes.h`".

```
void Protocol::aes_crt(byte* digest){
      mbedtls_aes_context aes;
      unsigned int nc_off = 0;
      unsigned char nonce_counter[16] = {0};
      unsigned char stream_block[16] = {0};
      unsigned char input[16*16]={0};
      unsigned char output[16*16]={0};
      esp_aes_init(&aes);
      esp_aes_setkey(&aes,digest, 256);
      esp_aes_crypt_ctr(&aes, 16 * 16, &nc_off,nonce_counter , stream_block, input,output);
      esp_aes_free( &aes );
}
```

Viene inizializzato il vettore di nonce lo stream block a 0, i quali hanno una lunghezza di 16 byte e vengono creati dei vettori di dimensione 16×16 byte inizializzati

anch'essi a 0. Si utilizzano vettori di 16 byte in quanto desideriamo creare un numero randomico di questa lunghezza che sarà il nostro Ephid, inoltre necessitiamo di 16 Ephid in quanto vogliamo coprire una fascia temporale di 4 ore, dividendola in epoche di 15 minuti, per cui $\frac{4\times60}{15} = 16$. A questo punto settiamo la chiave, il digest, ed otterremo una sequenza di 256 byte che verranno suddivisi in una sequenza di 16 byte e inseriti in un vettore di Ephid la quale è una struct definita nel file `"Utility.h"`. L'ultima funzione della classe è `"getRandomEphid()"` la quale ritorna in output un Ephid casuale all'interno del vettore e dalla quale viene eliminato in modo tale che esso non possa essere selezionato più volte dalla funzione. Alla fine della finestra temporale di quattro ore il vettore sarà vuoto e verranno rieseguiti i passaggi sopra citati per generare i nuovi Ephid per la successiva finestra d'invio.

5.3 Bluetooth

Dopo aver creato gli Ephid adesso è necessario inviarli attraverso la tecnologia BLE. Per accedere alle funzionalità BLE della scheda si è fatto uso di diverse librerie: `"BLEDevice.h", "BLEServer.h", "BLEUtils.h" e "BLE2904.h".` Tutte le funzioni inerenti a BLE sono state raccolte in una classe, la classe `"MyBT"` dichiarata all'interno del file `"modulo_BT.h"` e definita in `"modulo_BT.cpp".` La classe raccoglie sia i metodi utili per l'invio dei pacchetti , che quelli utili per la scansione dell'area circostante e la rilevazione di dispositivi vicini. Il costruttore della classe richiama la funzione `BLEDevice::init()` che abilita il modulo BT.
Per inviare i dati si utilizza la funzione `"startAdv()"` alla quale viene passato in input l'Ephid che deve essere inserito all'interno del pacchetto. Per creare questo pacchetto si instanzia un oggetto "`BLEAdvertisementData`" in cui verranno definiti i parametri "`CompleteService`" e `"ServiceData"` tramite le apposite funzioni `set`. Il complete service è un valore esadecimale che identifica il tipo di servizio offerto, in questo caso $0xfd6f$ il quale è il codice esadecimale che identifica il servizio di contact tracing, mentre i service data sono la combinazione di due valori: il service UUID, che nel progetto assume lo stesso valore del complete service, e il dato da inviare, ovvero l'Ephid. Una volta che il pacchetto è stato creato, è necessario

impostare il servizio di Advertising. La libreria BLE di ESP32 permette di ottenere l'istanza dell'oggetto "BLEAdvertising", utilizzando la funzione BLEDevice::getAdvertising(), che gestisce tale servizio. Ci avvaliamo quindi di questa instanza per registrare il pacchetto sopra creato come "AdvertisementData" usando l'apposita funzione set e dopo di che è possibile invocare "BLEDevice::startAdvertising()" per rendere attivo il servizio.
Le funzioni che invece si occupano della parte di ricezione sono "Scan()" e "setScan()". Quest'ultima serve in particolare per settare la funzione di CallBack ovvero una funzione specifica che viene richiamata quando viene scansionato un dispositivo. È possibile creare una propria funzione di callbacks creando una nuova classe che eredita da "BLEAdvertisedDeviceCallbacks" e ridefinisce i suoi metodi tra cui il metodo "OnResult()". In questo progetto utilizziamo questo metodo per controllare se il dispositivo scansionato abbia come Service UUID il codice esadecimale sopra definito per il contact tracing. Se il controllo dà esito positivo si procede con la memorizzazione del dato estratto dal pacchetto Service Data ricevuto e del relativo timestamp nella quale il pacchetto è stato ricevuto.
Invece la funzione "Scan()" esegue una scansione invocando la funzione "start()" dell'oggetto "BLEScan" precedentemente settato. Le scansioni possono essere di due tipologie: sempre attive o attive per un determinato lasso di tempo. Per lo scopo del progetto si è optato per una ricerca attiva solo per un determinato lasso di tempo ma che viene ripetuta all'interno della funzione "loop()" di Arduino IDE.
È bene notare che alla fine di un'epoca l'ephid che fino a qualche istante prima era inviato all'interno del pacchetto non deve essere più inoltrato ed occorre invece inviare l'ephid scelto randomicamente per la nuova epoca. In questo caso si esegue la funzione "stopAdv()" che blocca l'operazione di invio e vengono ripetuti i passaggi sopra descritti per la generazione del pacchetto ed il relativo invio.

5.4 Gestione della memoria

La memorizzazione è stata una parte particolarmente delicata e problematica del progetto. Si è presentato da subito il problema del limite di scrittura su memoria: difatti una memoria a stato solido tende ad usurarsi nel tempo perché ogni blocco può essere scritto un determinato numero di volte; quando viene raggiunto il limite di scritture per blocco, le informazioni memorizzate su di esso si perdono, con conseguente possibile corruzione dei dati. Di conseguenza, se dovesse essere sempre utilizzata la stessa zona di memoria per memorizzare le nostre informazioni, ne conseguirebbe un'usura prematura del disco; occorre quindi trovare una metodologia che permetta di distribuire uniformemente i dati sul disco. Il problema si riconduce a trovare un modo alternativo per capire quale è l'ultima zona di memoria in cui si è scritto, per evitare la riscrittura prematura della stessa area. La procedura adottata si ispira dal protocollo GAEN e nello specifico alla funzione per la generazione di i. Si noti, avendo nel seguente progetto adottato una durata di epoca di 15 minuti, possiamo calcolare

$ENIN(timestamp) = \frac{timestamp}{60*15}$ in seguito applichiamo

$rawIndex = floor(\frac{ENIN(timestamp)}{16})$ dove 16 è il numero di epoche per cui un seed è valido (ovvero $\frac{L \times 60}{15} = \frac{4 \times 60}{15} = 16$). Questo $rawIndex$ cresce una unità ogni volta che da una finestra temporale i si passa alla finestra temporale $i + 1$ con i intero.
La memoria invece è stata divisa in blocchi di una dimensione prestabilita, stimando il numero massimo di contatti che un individuo può avere durante l'arco della giornata. Tramite rawIndex, è possibile ottenere un intero univoco per una determinata finestra temporale. In questo modo, eseguendo una funzione modulo ben strutturata è possibile ottenere l'indice della prima cella relativa al blocco riservato a quella finestra temporale. La funzione modulo è così definita:

$index = ResSec + (((rawIndex - 1) \times ResSec) \, mod \, (nSec - SecForWin - ResSec))$

dove $ResSec$ è il numero di settori di memoria riservati per altre operazioni, $nSec$ è il numero di settori della memoria secondaria, $SecForWin$ è il numero di settori stimato per immagazzinare i dati in una sola finestra temporale.

I dati ricevuti tramite il protocollo GAP vengono mantenuti per un lasso di tempo in memoria primaria aumentando un contatore ogni qual volta lo stesso Ephid viene scansionato. Dopo un breve periodo di tempo i dati in memoria principale vengono salvati in memoria secondaria oppure vengono aggiornati i dati già presenti aggiornando il valore del contatore. Se un Ephid non viene più scansionato per un intervallo di tempo prestabilito allora esso viene salvato o aggiornato in memoria secondaria, inoltre non viene eliminato dalla memoria primaria. Tutte queste operazioni vengono eseguite dalla classe `"Observation_Array"` definita nel file `"Observation_array.h".`
Per trovare l'ultima zona di memoria in cui sono stati scritti dei dati all'interno del settore e quindi trovare la prossima zona di memoria in cui è possibile scrivere è sufficiente scansionare il settore fino a trovare dei caratteri di fine scrittura opportunamente prestabiliti.
Inoltre un altro dato che è importante da memorizzare è il seed con i relativi timestamp. In caso di sopraggiunta positività al Covid-19 l'utente può scegliere di inviare i seed al server centrale. Questi seed sono memorizzati in determinati settori di memoria, per il progetto si è scelto il settore 2, e non presentano particolari problematiche di memorizzazione.

5.5 Connessione al server

Come già evidenziato, la connessione alla rete Internet svolge un ruolo fondamentale per il funzionamento del protocollo anche per le richieste che vengono inviate al server centrale. È bene evidenziare che, per quanto il protocollo sia definito decentralizzato, esso ha comunque bisogno di un server centrale da cui ottenere la lista dei seed degli utenti infetti o dove caricare i suoi seed nel medesimo caso. È inoltre bene ricordare che tale server non contiene informazioni personali sugli utenti né è in grado di determinare l'individuo, avendo a disposizione i seed. Esso si limita ad una funzione di raccolta e notifica di tali seed ed è possibile immaginarlo come una grande bacheca.
Si è deciso di usufruire del backend fornito dagli stessi autori di DP3T e contenuto nella repository ufficiale.
La comunicazione con il server avviene perlopiù tramite richieste GET e POST ed esse sono implementate nella libreria `"HTTPClient.h".` Per prima cosa è necessario

definire un oggetto "HTTPClient" il quale gestirà le varie richieste. Si deve inoltre specificare l'Url del server: in fase realizzativa si è implementato il server in locale quindi l'indirizzo usato è "localhost" o banalmente 127.0.0.1, cioè l'indirizzo di loopback; inoltre i servizi utili del server sono "/v1/exposed" e "/v1/exposed/dayDateStr".

Per ottenere la lista dei seed occorre eseguire una richiesta GET. Si definisce quindi un oggetto "HTTPClient" e tramite esso si invoca la funzione "begin" che prende in input una stringa: ovvero l'indirizzo del server; nel nostro caso "http://127.0.0.1:8080/v1/exposed/" a cui concateneremo la stringa "?dayDateStr="+timestamp dove "timestamp" è il timestamp relativo alla data di cui si vuole ottenere il seed. Dopo aver definito l'url della richiesta GET, si invoca tramite l'oggetto la funzione "GET()" la quale ritorna in output il "Response code" della richiesta appena effettuata: se tale codice è 200 la richiesta è andata a buon fine ed invocando la funzione "getString()" è possibile ottenere il payload della richiesta che conterrà i vari seed. Tramite i seed così ottenuti si calcolano i vari ephid utilizzando la classe "Protocol" e controlliamo se essi sono stati memorizzati nei relativi settori di memoria il cui "index" viene ottenuto passando alla funzione "getIndex()" il timestamp relativo al seed tramite cui si sono calcolati gli Ephid. Se vi è una corrispondenza, tramite il contatore memorizzato in memoria relativo a tale Ephid si stima il tempo in cui i due utenti sono stati in contatto e quindi il grado di rischio che viene notificato all'utente attraverso l'uso del display.

Eseguendo una richiesta POST invece è possibile pubblicare sul server i propri seed nel caso in cui l'utente risulti positivo al COVID-19. Per eseguire la richiesta POST istanziamo il medesimo oggetto usato per effettuare la richiesta GET ma questa volta alla funzione "begin()" passiamo in input solo la stringa che identifica l'API, ovvero "v1/exposed". Dopo aver eseguito questo passaggio è necessario creare il request body specificando al suo interno i seed che vogliamo pubblicare. Per la realizzazione del progetto abbiamo deciso di codificare le informazioni da mandare nel formato JSON, il quale rappresenta un array di "Exposee".

In accordanza con la documentazione ufficiale del backend, "Exposee" è un oggetto che presenta due attributi : "key" che è una stringa contenente la codifica in base64 del seed e "onset" che è anch'essa una stringa che rappresenta la data della creazione del seed. Nel nostro progetto si è deciso di rappresentare questo oggetto attraverso una

struct. Per creare quindi questo request body, si fa uso della funzione `serializeJson()` che ritorna in output una stringa codificata nel formato JSON passando in input un oggetto "`StaticJsonDocument`" opportunamente istanziato; inoltre essa prende in input anche un buffer dove verrà memorizzato il dato in output. Dopo aver creato il JSON è possibile inviarlo tramite il metodo "`POST()`" dell'oggetto "`HTTPClient`" al quale viene passata la stringa stessa. Come per il metodo che implementa la richiesta GET, "`POST()`" tornerà in output il codice di risposta della richiesta: se tale codice è 200 la richiesta è andata a buon fine.

6 Conclusione

In questo elaborato sono stati presentati diversi tipi di protocolli ideati per risolvere il problema del contact tracing in modo sicuro e rispettando la privacy degli utenti. Nello specifico sono state presentate tre versioni del protocollo DP3T e il protocollo GAEN creato dalle società Apple e Google e fortemente ispirato al primo, fornendo anche un confronto critico.
Inoltre è stata presentata una implementazione semplificata del protocollo Hybrid decentralized proximity tracing su una scheda ESP32.
Sviluppi futuri relativi a questo progetto possono essere l'integrazione di questo sistema nell'ecosistema già presente di GAEN, implementando l'omonimo protocollo.

7 Riferimenti bibliografici

1. Decentralized Privacy Preserving Proximity Tracing White Paper, DP3T team;
2. Exposure Notification Cryptography Specification, Apple & Google Team;
3. RFC 4634 Secure Hash Algorithms, July 2006;
4. MMH*32 from scratch: an introduction to hash functions, Spencer Miller;
5. RFC 3686 Using advanced encryption standard (AES) counter mode with IPsec encapsulating security payload (ESP), January 2004;
6. RFC 2104 HMAC: keyed-hashing for message authentication, February 1997;
7. Bluetooth low energy, https://developer.android.com/guide/topics/connectivity/bluetooth/ble-overview;
8. Introduction to bluetooth low energy, https://learn.adafruit.com/introduction-to-bluetooth-low-energy;
9. Lecture 4, Pseudo-Random Generators (PRGs) and Pseudo-Random Functions (PRFs), University of Illinois, Urbana Champaign CS 598DK Special Topics in Cryptography, Instructor Dakshita Khurana, Scribe Surya Bakshi, September 6, 2019;
10. Digital Signature Standard (DSS), FIPS pub 186-4;
11. Dispense del corso di Internet Security, Giampaolo Bella;
12. Symmetric-Key Cryptography, lecturer Tom Roeder, http://www.cs.cornell.edu/courses/cs5430/2010sp/TL03.symmetric.html
13. Crittografia e Criptoanalisi, http://www.di-srv.unisa.it/~ads/corso-security/www/CORSO-9900/crittografiaclassica/www.apogeonline.com/catalogo/allegati/483/doc/algoritmi/crypto.htm;
14. Lilygo, TTGO Camera plus, http://www.lilygo.cn/prod_view.aspx?TypeId=50030&Id=1272&FId=t3:50030:3dule OV2640 1.3 Inch Display Rear Camera(1) - T-Camera - Shenzhen Xin Yuan Electronic Technology Co., Ltd;
15. Modalità di funzionamento della cifratura a blocchi, https://it.upwiki.one/wiki/Block_cipher_mode_of_operation#Counter_(CTR);
16. Cifrario di Cesare, https://it.wikipedia.org/wiki/Cifrario_di_Cesare;