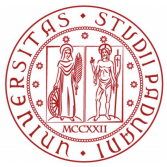

Università degli Studi di Padova

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

ACCELERAZIONE DI ALGORITMI DI POST-PROCESSING PER QUANTUM KEY DISTRIBUTION ATTRAVERSO SYSTEM-ON-A-CHIP

Relatore:
Dott. Andrea Stanco

Laureando:
Francesco Campigotto

Anno Accademico 2023/2024

23 Settembre 2024

Abstract

The purpose of the following thesis is the design and implementation of post-processing for Quantum Key Distribution(QKD) through the usage of the System-on-a-Chip (SoC) technology. The QKD is a method that allows the exchange of cryptographic keys using the properties of quantum mechanics that guarantee the security of the transmitted information. Usually, in QKD experiments, Field Programmable Gate Array (FPGA) boards with one or more CPUs integrated are used for the purpose of processing data and transforming a raw sequence of data into a secret key. A general introduction will be made on the cryptography, the QKD, the protocol BB84 and the FPGA, at the end of which the carried out activities are exposed discussed. Using programs written in C, the "sifting" of the data was implemented, validating the results with testing programs developed in C++ from which relevant statistics for the analysis of the experiments can be drawn.

Sommario

Lo scopo della seguente tesi è quello di ideare ed implementare l'algoritmo di post-processing per la Quantum Key Distribution (QKD) attraverso l'uso della tecnologia System-on-a-Chip (SoC). Quest'ultimo è un metodo che permette lo scambio di chiavi criptografiche utilizzando proprietà della meccanica quantistica che garantiscono la sicurezza delle informazioni trasmesse. Solitamente, negli esperimenti di QKD, vengono utilizzate delle Field Programmable Gate Array (FPGA) con integrate una o più CPUs allo scopo di processare i dati e trasformare una sequenza grezza di dati in una chiave segreta. Verrà fatta un'introduzione generale sulla criptografia, la QKD, il protocollo BB84 e l'FPGA, al termine della quale è stata esposta e discussa l'attività effettuata. Utilizzando dei programmi scritti in C è stato implementato il "sifting" dei dati, validando i risultati con programmi di testing sviluppati in C++ da cui si possono trarre statistiche rilevanti per l'analisi degli esperimenti.

Indice

Capitolo 1

Introduzione

Uno degli aspetti più importanti ma nascosti della nostra realtà, ormai dominata dalla tecnologia, è quello della crittografia. Nonostante la sua applicazione e sviluppo abbiano avuto una forte crescita in tempi recenti, una delle prime forme della crittografia è nata al tempo dei romani con il cifrario di Cesare. Da quel momento in poi sono state fatte innovazioni su innovazioni, introducendo supporti elettromeccanici per poi passare all'elettronica con i computer e arrivando, infine, ai tempi d'oggi, dove si è iniziato a parlare di crittografia quantistica.
Inizialmente, il primo metodo usato per la comunicazione consisteva nella crittografia simmetrica: questa però richiedeva l'invio e la ricezione della chiave segreta, cosa che rendeva la comunicazione facilmente esposta all'attacco di una terza persona. Si è perciò passati alla crittografia asimmetrica, dove la chiave per crittografare può essere nota a chiunque, mentre quella per decifrarla è in possesso solo del destinatario. Ciò ha tolto la necessità di doversi scambiare una chiave, ma ha dato la possibiltà a persone esterne di conoscere la chiave pubblica e ha aperto la strada a un nuovo modo di decifrare il messaggio.
Infatti, la necessità di sviluppare nuove metodologie per lo scambio di chiavi è stato reso necessario dalla controparte della crittografia che si occupa di ottenere le informazioni criptate senza conoscere l'informazione segreta: la crittoanalisi. In particolare, per quanto riguarda la crittografia asimmetrica, la sicurezza di tale comunicazione è garantita dalla complessità matematica e temporale necessaria alla decifrazione della chiave. Il criterio fondamentale che dà una stima della solidità di questa è il numero di cifre da cui è composta. Infatti, considerando la tecnologia odierna, le chiavi contenenti centinaia di cifre sono ancora considerati difficili da decifrare, ma, con l'arrivo dei computer quantici in grado di ridurre drasticamente i tempi computazionali, anche se sono ancora nelle loro prime fasi

di sviluppo, si è vista la necessità di introdurre e sviluppare nuovi metodi per la comunicazione criptata, arrivando così a introdurre la crittografia quantistica.
La crittografia quantistica usa proprietà della meccanica quantistica nella fase di scambio della chiave per evitare che questa possa essere intercettata da un agente esterno, senza che le parti in gioco se ne accorgano. Il principio fondamentale alla base della crittografia quantistica è il principio di indeterminazione di Heisenberg: dato un oggetto quantistico, non ci è possibile conoscere simultaneamente e in maniera completamente precisa due determinate proprietà di questo oggetto, come per esempio la posizione e la quantità di moto. Questo è di fondamentale importanza in quanto garantisce l'esistenza di un fenomeno fisico casuale e imprevedibile. Infatti, secondo la meccanica quantistica, per quanto si possano sviluppare tecniche sempre più nuove e precise, questa imprevedibilità non potrà mai essere eliminata ed è proprio questo ciò che sta alla base della Quantum Key Distribution.
Entrando in merito alla tesi, l'attività svolta consiste nell'implementazione di un codice in grado di effettuare il sifting all'interno delle schede FPGA sprovviste di sistema operativo, con successiva creazione di un codice di validazione per la raccolta dei dati. In seguito è stata implementata una comunicazione enthernet tra due schede distinte, in modo da simulare la fase di post-processing tra le due. Anche in questo caso è stato ideato un codice di validazione per la raccolta dei dati.
Nel secondo capitolo sono state introdotte le nozioni necessarie all'interpretazione della tesi, ovvero il protocollo BB84 e i teoremi che lo permettono. Il terzo capitolo parla di come funzionano le schede FPGA, con particolare riferimento ai SoC e del loro utilizzo nel post-processing. Nel quarto capitolo viene discusso il codice, il suo sviluppo, le scelte implementative e i risultati ottenuti. Il quinto capitolo offre una prospettiva su come il codice sviluppato nel capitolo precedente venga poi implementato in uno scenario simile a quelli degli esperimenti della QKD. Nel capitolo finale si discutono, in conclusione, i risultati ottenuti.

Capitolo 2

Quantum Key Distribution

La distribuzione a chiave quantistica (con sigla QKD dall'inglese Quantum Key Distribution) è un sistema della meccanica quantistica che garantisce una comunicazione sicura delle chiavi tra due parti. Per capirla a pieno è necessario spiegare il concetto di qubit in quanto base dell'informazione, il protocollo BB84 per la creazione delle chiavi e i teoremi che garantiscono la sicurezza della comunicazione.

2.1 Quantum bit

Un modo molto semplice per introdurre il concetto di quantum bit, da ora in poi abbreviato con qubit, è paragonarlo al classico bit. Infatti, quest'ultimo può assumere solo i valori (0) e (1). Allo stesso modo, possiamo chiamare gli stati che il qubit può assumere come $|0\rangle$ e $|1\rangle$, usando la notazione di Dirac. Questa viene usata per descrivere lo stato quantico del qubit, in quanto denota un vettore astratto risultante da una sovrapposizione degli stati $|0\rangle$ e $|1\rangle$ fino a quando non viene osservato quello effettivo, secondo il principio di sovrapposizione. In questo modo, il valore può essere espresso in forma lineare come:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.1}$$

Qui, α e β sono delle variabili complesse e il risultato della misurazione di un singolo qubit nello stato (2.1) sarà $|0\rangle$ con probabilità $|\alpha|^2$ o $|1\rangle$ con probabilità $|\beta|^2$. Questo porterebbe a pensare che dovrebbero esserci quattro gradi di libertà, in quanto nell'equazione sono presenti due numeri complessi, ognuno dei quali possiede due gradi di libertà. Dobbiamo però osservare che, avendo solo due

eventi possibili, questi sono l'uno il complementare dell'altro. Pertanto, possiamo eliminare uno dei gradi di libertà, scrivendo che:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.2}$$

In questo modo possiamo rappresentare l'equazione (2.1) con un grafico detto sfera di Bloch. Questo è necessario in quanto un classico bit, che può assumere solo i valori 0 e 1, può trovarsi senza problemi all'estremo superiore e inferiore della sfera, dove abbiamo anche $|0\rangle$ e $|1\rangle$, ma non può trovarsi in nessun altro punto della superficie, cosa invece necessaria alla rappresentazione dello stato quantico di un qubit [1].

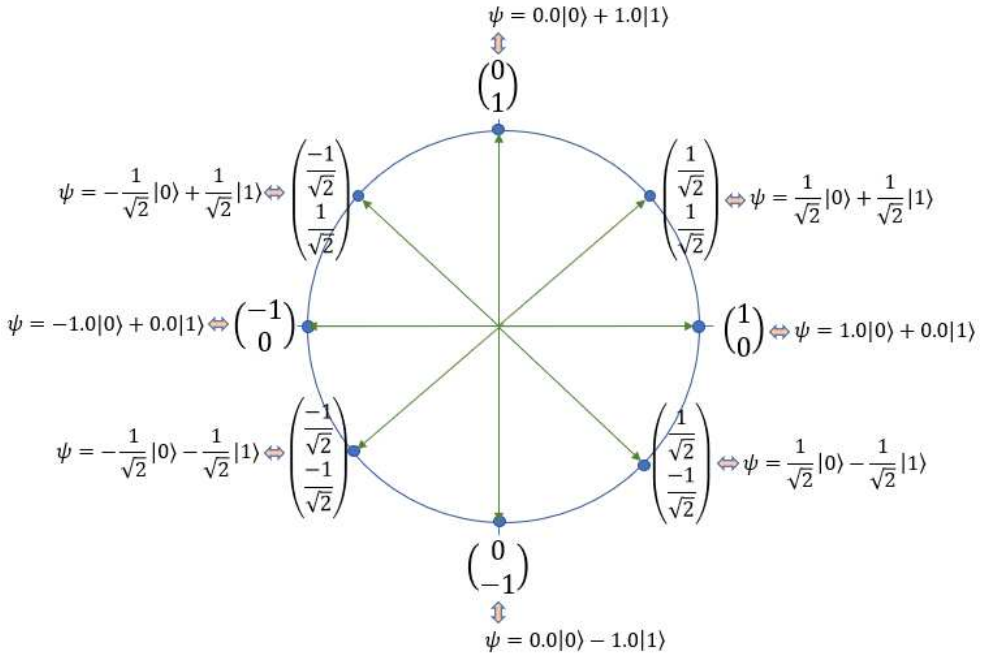

Figura 2.1: Rappresentazione circolare del qubit [2]

Parlando invece dell'effettiva implementazione fisica dei qubit, possiamo usare diverse tipologie di oggetti quantici per esprimere un'informazione e alcuni di essi hanno più supporti possibili. Per esempio, considerando il nucleo di un atomo, possiamo sfruttare la presenza di uno spin nucleare, che semplicisticamente si indica come "up" e "down". Allo stesso modo possiamo sfruttare lo spin degli elettroni, ma potremmo anche prendere come base dell'informazione la presenza di un elettrone in più o la sua assenza in un atomo. Nel nostro caso, sfrutteremo le proprietà dei fotoni, i quali possono esprimere informazione grazie alla loro presenza, il tempo di arrivo o, nel nostro caso, con la QKD e il protocollo BB84, la loro polarizzazione.

2.2 Protocollo BB84

Il protocollo BB84 è il primo protocollo della distribuzione a chiave quantistica e della crittografia quantistica, sviluppato da Charles Bennett e Gilles Brassard nel 1984. Questo è composto da tre attori: Alice, l'emittente della chiave; Bob, il ricevitore della chiave; Eve, un'entità esterna che cerca di intercettare la chiave. Supponiamo che Alice sia collegata a Bob tramite un canale quantistico, nel quale può inviare singoli fotoni, potendo decidere arbitrariamente la polarizzazione di questi. In questo caso, l'informazione data dal fotone non è solamente 0 e 1 come ci aspetteremmo: infatti, la polarizzazione può essere suddivisa in due basi, che definiremo come + e ×, non ortogonali tra di loro. La prima può essere suddivisa in orizzontale e verticale, mentre la seconda in diagonale e antidiagonale: in questo modo, con un solo fotone possiamo avere quattro possibilità diverse che esprimono un'informazione rappresentabile su due bit. Sapendo questo possiamo usare la notazione (2.1) e le nozioni sulla sfera Bloch per scrivere le possibili rappresentazioni dello stato quantistico del fotone:

1. $|\psi_{00}\rangle = |0\rangle$
2. $|\psi_{10}\rangle = |1\rangle$
3. $|\psi_{01}\rangle = |+\rangle \ = \frac{1}{\sqrt{2}}|0\rangle \ + \frac{1}{\sqrt{2}}|1\rangle$
4. $|\psi_{11}\rangle = |-\rangle \ = \frac{1}{\sqrt{2}}|0\rangle \ - \frac{1}{\sqrt{2}}|1\rangle$

Da notare come, nei primi due casi, uno dei coefficienti α o β assuma come valore zero, rendendo veritiera l'equazione (2.2) e certa l'informazione contenuta, mentre, nelle ultime due, la verifica di tale condizione è un semplice calcolo: elevando alla seconda i due coefficienti, otteniamo in entrambi i casi 0.5, che, sommati, risultano 1, confermando l'equazione (2.2). Andiamo ora a vedere passo passo quali sono i passaggi fondamentali del protocollo BB84:

- Alice sceglierà, in primo luogo, una base e, successivamente, uno dei due stati di questa in maniera casuale. Il fotone verrà spedito tramite canale di comunicazione ottico (fibra o spazio-libero) a Bob per le N volte necessarie alla creazione della chiave. Per ottenere l'informazione, Bob dovrà decidere una delle due basi: nel caso in cui quella del mittente e del ricevente siano la stessa, si avrà con certezza il valore $|0\rangle$ o $|1\rangle$, invece, nel caso in cui le due basi siano discordi, si registrerà nel 50% dei casi il valore $|0\rangle$ e nell'altro

50% il valore $|1\rangle$, in concordanza con le quattro equazioni precedenti. Tutto questo avviene non considerando possibili perdite o disturbi dati dal mezzo trasmissivo o interventi da parte di Eve, arrivando ad avere due chiavi grezze, una per Bob e una per Alice, che potrebbero non coincidere.

- In seguito, avviene la prima fase di post-processing, durante la quale viene effettuato il sifting. La sua implementazione verrà discussa più approfonditamente nei successivi capitoli in quanto oggetto principale della tesi. Sostanzialmente, Alice e Bob comunicano tramite un canale classico e scambiano informazioni riguardanti soltanto le basi. Nel caso in cui queste risultino uguali, Bob salva anche il valore del bit di Alice relativo a quella base, che viene poi utilizzato per poter creare una chiave finale.

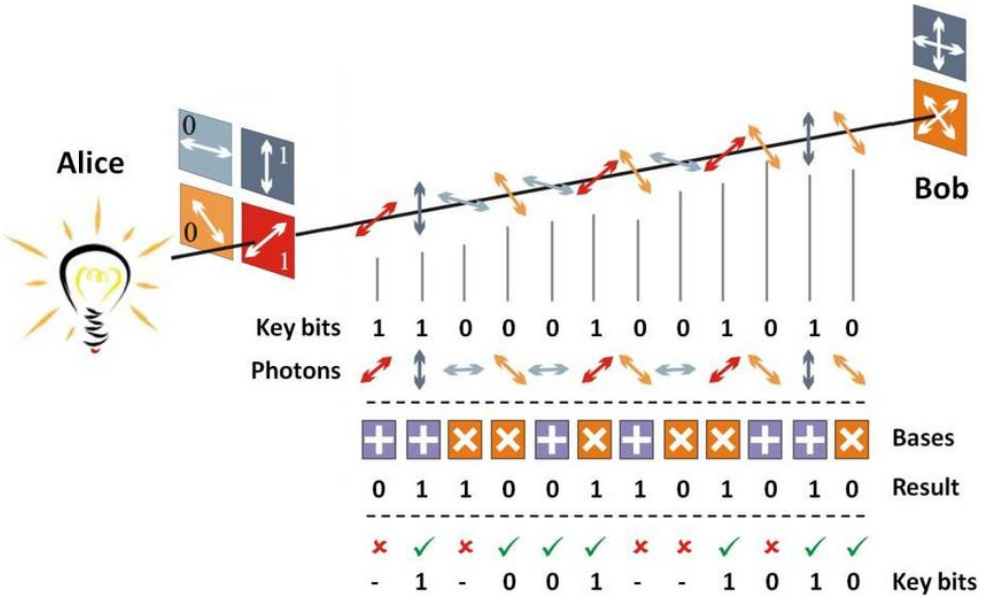

Figura 2.2: Schematizzazione del protocollo BB84 [3]

- Il passo successivo è quello di capire se la chiave è stata compromessa grazie alla parameter estimation. Durante questa fase, Alice e Bob comunicano sul canale classico parte della chiave ottenuta, la quale risulta in seguito inutilizzabile in quanto chiunque potrebbe intercettarla. Ciò serve a capire se sono avvenuti errori a causa dell'intervento di Eve. Non potendo distinguere errori dovuti da agenti esterni o dal rumore, se vengono rilevate discrepanze si suppone che siano dovute tutte a Eve e quella parte di informazione viene segnata come potenzialmente a rischio, così da essere eliminata in seguito. Chiaramente, più lungo è il segmento di chiave scambiato, maggiore è la certezza del risultato, ma si avrà una chiave finale più

corta. Viceversa, usare segmenti troppo corti porterebbe gli errori statistici a rendere il giudizio meno accurato [4].

- Se la chiave non viene scartata a causa della presenza di troppi errori, si procede con la fase di error correction. Viene preso un determinato set di bit della rimanente chiave e vengono effettuate le necessarie modifiche in modo che, alla fine del processo, Alice e Bob siano in possesso della stessa chiave grezza.

- In conclusione, su questa viene effettuata la privacy amplification che, grazie ad un'opportuna funzione di hashing, permette di eliminare alcuni bit in modo da ridurre al minimo qualsiasi informazione in possesso di Eve. Generalmente, se durante la parameter estimation si sono calcolati T bit compromessi sugli N utilizzati, è necessario che venga tolta la stessa percentuale di $\frac{T}{N}$ dalla chiave grezza. Ciò non toglie che ne possano essere eliminati di più, anzi: maggiore il numero S di bit extra rimossi, maggiore risulterà la segretezza della chiave condivisa, non volendo chiaramente compromettere eccessivamente la lunghezza e complessità della chiave stessa, fattore controproducente [5].

Certamente, questi passaggi sono solo una forma sintetizzata di quello che descrive il protocollo BB84 [6] [7].

2.3 Comunicazione sicura

Fino a ora abbiamo parlato del funzionamento e dei passaggi del protocollo BB84, ma non abbiamo ancora visto i motivi che rendono questo, a livello teorico, uno dei metodi di comunicazione più sicuri esistenti. Il teorema che sta alla base di tutto questo è quello di non clonazione quantistica, il quale afferma che, dati i postulati della meccanica quantistica, non è possibile duplicare esattamente e in maniera indipendente uno stato quantistico a priori. Ciò si verifica solo nel caso in cui si abbiano stati non ortogonali tra di loro, in quanto, dato un insieme ortogonale di stati, è sempre possibile effettuare la duplicazione senza errori se questo è conosciuto a priori, fallendo nel caso in cui lo stato non appartenga all'insieme dato. Inoltre, è sempre possibile duplicare lo stato di un oggetto quantico se questo è già conosciuto. In pratica, un generico processo di duplicazione consisterebbe nell'identificazione di un operatore unitario U tale che, per ogni stato

$|\phi\rangle$, vale la seguente relazione:

$$U\,|E\rangle \otimes |\phi\rangle \otimes |\eta_1\rangle \otimes |\eta_2\rangle \otimes \cdots \otimes |\eta_n\rangle = |E_\phi\rangle \otimes |\phi\rangle \otimes |\phi\rangle \otimes \cdots \otimes |\phi\rangle \qquad (2.3)$$

Il teorema di non clonazione quantistica afferma che tale operatore unitario, con queste caratteristiche, non può esistere.
Ciò è di fondamentale importanza quando abbiamo a che fare con agenti esterni come Eve, che possono osservare e interagire con il canale quantistico. In particolare, nel caso in cui Eve sia in grado di intercettare il fotone e osservarne lo stato quantico, il teorema di non clonazione farà sì che, se viene scelta la base giusta, a Bob verrà inviata la stessa informazione spedita da Alice, ma, nel caso in cui venga scelta la base sbagliata, a Bob verrà inviata un'informazione completamente errata. In questo caso, la fase di parameter estimation del protocollo BB84 provvederà a decidere se tenere o scartare completamente la chiave grezza.

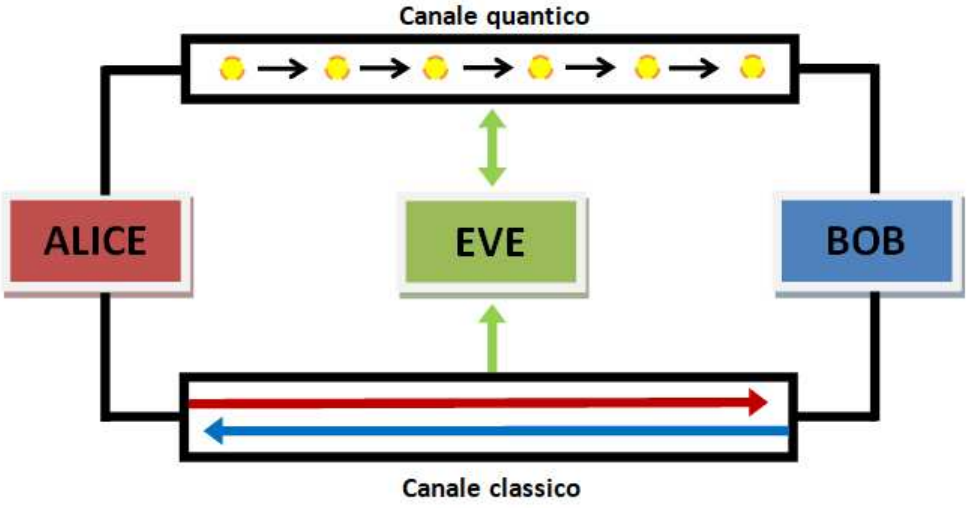

Figura 2.3: Schematizzazione della comunicazione tra Bob e Alice con Eve

Un altro aspetto molto importante da considerare, che fino a ora abbiamo ignorato, è la difficoltà nell'inviare un singolo fotone nel canale quantistico. Infatti, idealmente, il protocollo BB84 prevede che venga inviato un solo fotone, ma, realisticamente parlando, i laser attenuati utilizzati nella distribuzione a chiave quantistica inviano impulsi a singoli fotoni con una determinata probabilità, che non può risultare mai esattamente 1. Pertanto, quando Alice invierà più di un fotone, Eve potrebbe intercettare quelli in eccesso, ottenendo informazioni sulla chiave e lasciare altri fotoni nel canale senza lasciare tracce. Per ovviare a ciò,

possono essere utilizzati degli stati fantoccio: essenzialmente, Alice invierà degli impulsi con un numero di fotoni minore rispetto alla media. In questo modo, in caso di un attacco PNS (photon-number splitting), Eve non sarà in grado di distinguere tra un segnale effettivo e uno fantoccio. Così, durante la fase di post-processing, Alice e Bob si possono confrontare e dedurre se c'è stata intromissione nel segnale. Inoltre, si stanno sviluppando sorgenti che rilascino effettivamente un solo fotone, così da eliminare alla radice il problema del PNS [8].
Invece, nel caso di un attacco DOS (denial of service), non c'è molto che si possa fare: la comunicazione tra Alice e Bob viene bloccata momentaneamente o in maniera definitiva. Pertanto, non si può fare altro che ricominciare la costruzione della chiave da capo o trovare un altro tragitto per la comunicazione. In ogni caso, questo tipo di attacco non è proprio del protocollo BB84 in quanto ogni tipo di connessione può esserne affetta. Inoltre, è da sottolineare che ciò non mette a rischio la segretezza in sé della chiave, ma la sua stessa creazione.
Infine, uno degli attacchi più efficaci da parte di Eve è quello che viene detto Trojan-horse. Sostanzialmente, è possibile sondare un sistema a QKD inviando segnali luminosi nel canale e, analizzando l'effetto della riflessione, ottenere informazioni sulla chiave. In recenti studi si è dimostrato che Eve è in grado di osservare la base segreta scelta da Bob con un successo del 90%, minando alla sicurezza del sistema [6].

Capitolo 3

Strumentazione

Nel seguente capitolo è spiegato in cosa consiste una Field Programmable Gate Array (FPGA) e si descrive la scheda sulla quale è montata. Inoltre, si delinea una panoramica dell'ambiente di sviluppo utilizzato per la simulazione del funzionamento della scheda.

3.1 Field Programmable Gate Array

Una scheda FPGA è un tipo di circuito integrato che può essere configurato molteplici volte grazie a opportuni linguaggi di descrizione hardware. Tra quelli a oggi più utilizzati individuiamo il linguaggio VHDL, simile ai linguaggi utilizzati per circuiti integrati specifici, il quale costituisce uno degli scogli più grande della programmazione delle FPGA poiché molto più complesso dei linguaggi C. Nonostante questo aspetto, al contrario delle schede integrate ASIC ("Application Specific Integreted Circuit"), le schede FPGA permettono molta flessibilità a discapito di inefficienze dal punto di vista energetico e delle risorse. Per questo motivo, l'ambito in cui le FPGA spiccano particolarmente è la creazione di prototipi che possono poi essere usati come base per gli ASIC. La caratteristica fondamentale che rende l'utilizzo delle FPGA necessario in questo ambito sta nella natura del linguaggio utilizzato. Infatti, il linguaggio VHDL permette in ogni momento, sapendo le condizioni iniziali, di stabilire in un deteminato instante di tempo cosa stia facendo il programma. Ciò è necessario in quanto si va a lavorare con milioni di dati al secondo e non si vuole che ottimizzazioni fatte da sistemi operativi o CPU facciano perdere il ritmo al programma, rendendolo inefficiente o addirittura inutilizzabile [9].

3.2 Scheda PYNQ-Z2

Nello specifico, la scheda su cui è montata l'FPGA è la PYNQ-Z2. Il modello in sé non è importante, ma una delle caratteristiche fondamentali da tenere in considerazione è la presenza di una CPU, anche se possono essercene più di una, che lavora in parallelo all'FPGA. Questo permette di poter ricevere e analizzare i dati in maniera deterministica e poi gestire, grazie al processore, le fasi successive, che non richiedono tale precisione. In particolare, la CPU si occupa delle fasi di sifting, confronto dei dati e creazione della chiave grezza. Inoltre, la comunicazione tra la scheda e il computer avviene tramite cavo, mentre, nel caso in cui si voglia instaurare un collegamente tra due schede diverse, questo può essere fatto collegandole tramite ethernet e creando appositi programmi per accedere e utilizzare questo protocollo [10]. Quest'ultimo punto è stato sviluppato in seguito (cfr. § 5.1).

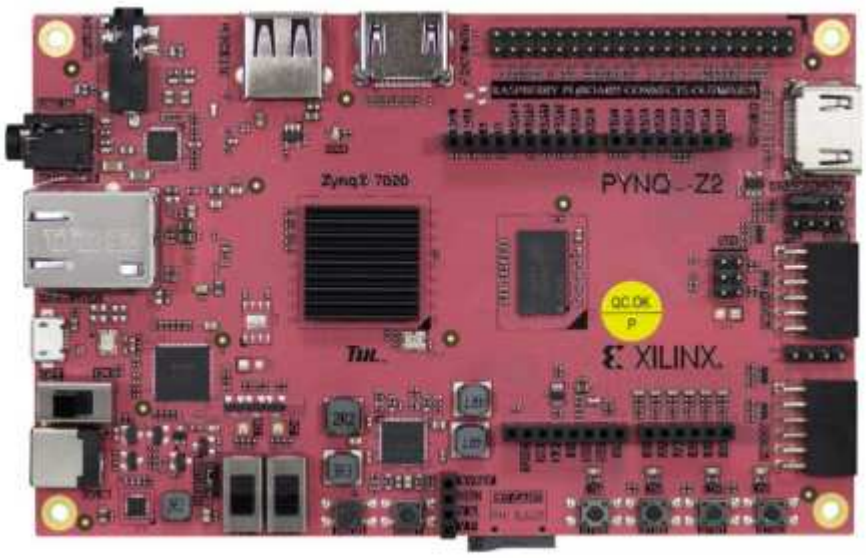

Figura 3.1: Immagine della scheda PYNQ-Z2 [10]

3.3 System-on-a-chip

La scheda PYNQ-Z2 è pensata per lo sviluppo di codice ed è costruita attorno al Xilinx ZYNQ 7020 SoC (System on Chip). Si tratta di un tipo di circuito integrato che è composto dalla maggior parte delle componenti che si trovano in un computer: una o più CPUs, un'interfaccia di memoria, dispositivi di Input/Output e un'eventuale memoria secondaria. Questi sono ampiamente utlizzati

soprattutto nella produzione di dispositivi elettronici, come i cellulari: a questi SoC viene aggiunta una GPU e dispositivi per la gestione di segnali Wi-Fi e radio. Più in generale, i SoCs possono essere applicati in qualsiasi ambito in cui è richiesto di avere grande potenza di calcoli in spazi ristetti o per sistemi embedded specifici.

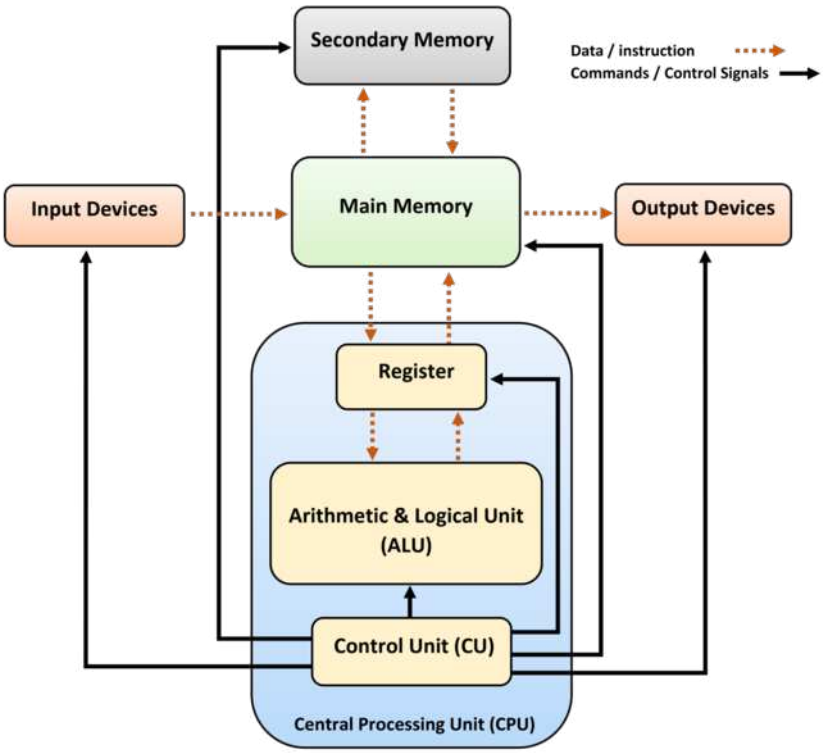

Figura 3.2: Schematizzazione delle componenti di un SoC generico [11]

La loro progettazione richiede una creazione simultanea delle parti hardware e software, in quanto queste devono lavorare in maniera precisa e consistente. Prima della produzione in massa di un determinato SoC, questo deve passare un processo di validazione. In questo ambito torna in gioco l'adattabilità delle FPGA, che, data la loro natura riprogrammabile, permette una più efficente creazione di prototipi e verifica tramite debug rispetto alle ASICs.
Durante la produzione del codice, fulcro della tesi, è stata usata la parte della CPU del SoC montato sulla scheda. Questo ha permesso di sviluppare un codice standalone che non richiede alcun tipo di sistema operativo. Ciò è chiaramente più efficiente rispetto a programmi soggetti a interrupt del computer e permette una gestione molto più flessibile delle risorse a disposizione, come ad esempio la memoria o la comunicazione tramite dispositivi di Input/Output [12].

3.4 Vivado e Vitis

Per il funzionamento della scheda e il suo utilizzo è necessario impiegare un apposito ambiente di sviluppo chiamato Vivado. Questo presenta strumenti per la progettazione a livello di sistema elettronico e permette la costruzione dei blocchi che formano il sistema e la loro verifica. Ciò dà la possibilità di eseguire programmi all'interno delle schede, le quali hanno proprie limitazioni relative a memoria, output e, in generale, utilizzo delle risorse disponibili. Affiancato a Vivado troviame l'ambiente di sviluppo che è invece utilizzato per programmare ed eseguire i programmi in C sulla scheda, ovvero Vitis. Questo, oltre a permettere le comuni funzioni necessarie alla programmazione, permette l'utilizzo di specifici comandi e librerie legate alle SoC e schede FPGA in generale. Inoltre, offre una serie di programmi tameplate preconfigurati che possono essere utilizzati per impostare programmi piu complessi. In particolare, durante lo svolgimento di questa tesi sono stati utilizzati le librerie lwIP TCP Perf Client e lwIP TCP Perf Server come base per il client e server della connessione ethernet. Questi aspetti verranno meglio esposti nei capitoli seguenti (cfr. § 5.1), dove verranno analizzati il codice, le ottimizzazione che sono state necessarie e i risultati ottenuti.

Capitolo 4

L'algoritmo di sifting

In questo capitolo sarà discusso quello che è stato l'effettivo obiettivo della tesi, ovvero l'implementazione di un codice in grado di effettuare il sifting delle basi di Alice e Bob allo scopo di individuare i bit utili alla creazione della chiave.

4.1 Sviluppo del codice

Prima di parlare del codice in se è necessario sottolineare alcune approssimazioni fatte nel corso della sua scrittura. Infatti, il valore delle basi e bit durante la comunicazione normalmente è ricavato da canale quantistico e proprietà dei fotoni, mentre, in questo caso, è stata usata una funzione che genera in maniera casuale il valore di questi, impostando un "seed". Inoltre, essendo un unico codice, le informazioni tra Bob ed Alice sono condivise, cosa non ammissibile nel protocollo BB84: questo verrebbe risolto instaurando una comunicazione tra due schede diverse. Nonostante l'implementazione tramite ethernet non rispetti comunque gli standard di sicurezza, ciò viene utilizzato in quanto si è ancora nella fase di testing.
Le variabili più importanti sono: la dimensione, che indica il numero di elementi che si vogliono ottenere; il seed da utilizzare; i diversi counter, necessari per tenere traccia di quanti valori validi sono stati trovati, quanti risultati sono stati generati e il numero di casi in cui base e bit combaciavano perfettamente. Inoltre, sono stati creati due array distinti, contenenti tutte le informazioni di Alice e Bob, aventi come dimensione il doppio dei valori che si vogliono ottenere. È stato poi creato un array per salvare la posizione dei dati validi, ovvero dati dove le basi combaciano, e un altro contenente la base e bit risultanti dal sifting.

Nel main vengono generati gli elementi di Alice e Bob di cui viene fatto successivamente il sifting, usando come base il seguente pseudocodice:

Algorithm 1 Sifting

```
Inizializza un contatore a 0
Per ogni indice i da 0 a DIM - 1:
Se la base di Bob[i] è uguale alla base di Alice[i]:
        Imposta validi[i] a 1
        Incrementa il contatore di 1
Altrimenti:
        Imposta validi[i] a 0
```

Successivamente, in un altro ciclo for, si riempie il vettore dei risultati con le basi e bit di Alice dove il valore di validi[i] corrisponde a 1. Durante questo si conta anche il numero di volte che i bit di Alice e Bob corrispondono, per avere un dato più attendibile sulla correttezza dell'informazione ricevuta idealmente da Bob. Questo passaggio potrebbe risultare superfluo, in quanto potremmo direttamente riempire il vettore dei risultati, ma bisogna tenere conto che questo codice dovrà poi essere implementato su due schede diverse e sul canale classico non vogliamo comunicare l'informazione stessa, ma la posizione nell'array di questa.
La restante parte di codice è necessaria alla creazione di un log in cui sono mostrate tutte le informazioni utili: questo sarà poi usato per la validazione del risultato su un programma separato, fatto eseguire sul computer di cui parleremo nei risultati (cfr. § 4.3).

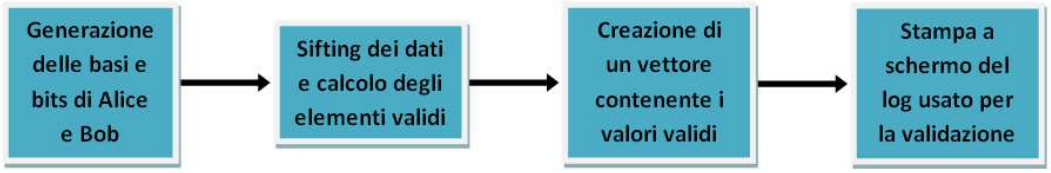

Figura 4.1: Schema a blocchi dell'esecuzione del programma

4.2 Scelte implementative

Durante lo sviluppo del codice sono state rilevate problematicità che hanno porta il risultato finale a dover implementare certi metodi o optare per alcune scelte

rispetto ad altre.
Durante lo sviluppo si è considerato più volte di optare per una gestione della memoria dinamica piuttosto che per l'uso di semplici array statici. Ciò, in linea teorica, renderebbe più efficiente il programma, ma, vista la natura stessa del programma, che non richiede vettori di dimensioni variabili o la necessità di copiare segmenti di memoria, non si è visto alcun miglioramento dell'efficenza che giustificasse l'implementazione di una memoria gestita in maniera dinamica. Altro dettaglio tecnico è la presenza di funzioni sleep(N) della libreria "sleep.h" a seguito dell'output di ogni vettore, le quali fermano il programma per un determinato numero N di secondi. Questo è stato necessario in quanto, nonostante il corretto funzionamento del programma e l'ottenimento dei risultati, la creazione del log poteva risultare errata, superato il milione di elementi. Ciò è stato ricondotto a limiti del buffer dell'output, cosa risolvibile mettendo in pausa il programma e lasciando quest'ultimo il tempo di liberarsi.
Infine, il programma ha un vero e proprio limite riguardante la grandezza possibile dei vettori. Infatti, superati circa gli 80 milioni di elementi, le funzioni di debug iniziano a segnalare errori che, teoricamente, non dovrebbero sussistere. Questo accade in quanto in C non esiste il tipo bool, pertanto ogni elemento occupa minimo 8 bit. Contando che si hanno due vettori per le basi e i bit di Alice e Bob, entrambi grandi due volte la dimensione, gli elementi validi, con tanti elementi quanti la dimesione, e i risultati, che deve avere la stessa dimensione di Alice, si arriva a poco meno di 4,5 Gbit, che supera il limite di 4 Gbit della scheda. Nonostante ciò, non si tratta di una problematica particolarmente grave, dal momento che il programma non lavorerà mai con più di una decina di milioni di elementi.

4.3 Validazione dei risultati

Parallelamente all'implementazione del sifting è stato creato un programma che permette l'effettiva verifica del risultato. Il codice in questione legge da un file di testo il log generato dal programma principale e, utilizzando gli elementi generati casualmente da Alice e Bob, svolge nuovamente il procedimento. Questo potrebbe sembrare ridondante, ma vogliamo assicurarci che i risultati ottenuti dalla scheda siano gli stessi di quelli ottenuti da un normale computer. Pertanto, il programma crea i suoi vettori di elementi validi e risultati, copia dal log quelli calcolati dalla scheda e procede a confrontare uno a uno ogni elemento nella stessa

posizione. Durante questo processo viene anche eseguito un debug che controlla se sono presenti numeri anomali diversi da 0 e 1.
Sono state fatte numerose prove, andando a modificare il numero di dati ed il seed utilizzato per generarli. In questa parte non verranno mostrate le prove fatte con pochi elementi, ma solo quelli con un numero abbastanza significativo di campioni, in modo da avere dati statisticamente più affidabili possibili.

SEED[]	1	2	3	4	5	6	7	8	9	10	AVERAGE
VALID[bit]	49925	50010	49932	49774	49974	49867	49971	50000	50130	49991	49957
RESULT[bit]	49925	50010	49932	49774	49974	49867	49971	50000	50130	49991	49957
MATCH[bit]	24992	24977	25049	24955	25154	24894	25064	25021	25206	24973	25028
RESULT[%]	49,93%	50,01%	49,93%	49,77%	49,97%	49,87%	49,97%	50%	50,13%	49,99%	49,96%
MATCH[%]	24,99%	24,98%	25,05%	24,96%	25,15%	24,89%	25,06%	25,02%	25,21%	24,97%	25,0285%
SIFT TIME[ms]	0,832	0,832	0,832	0,831	0,832	0,832	0,832	0,832	0,832	0,832	0,832

Tabella 4.1: Per i primi dieci seed è riportato il numero di elementi validi, risultati e coppie uguali, le loro percentuali e i colpi clock necessari per 100.000 elementi

SEED[]	1	2	3	4	5	6	7	8	9	10	AVERAGE
VALID[bit]	500365	500488	499731	500639	500799	500245	499700	500529	501014	500072	500358
RESULT[bit]	500365	500488	499731	500639	500799	500245	499700	500529	501014	500072	500358
MATCH[bit]	250071	25032	249581	249885	250324	249944	249698	249817	250804	250025	227518
RESULT[%]	50,04%	50,05%	49,97%	50,06%	50,08%	50,02%	49,97%	50%	50,10%	50,01%	50,04%
MATCH[%]	25,01%	25,04%	24,96%	24,99%	25,03%	24,99%	24,97%	24,98%	25,08%	25,00%	25,00%
SIFT TIME[ms]	8,322	8,323	8,322	8,322	8,322	8,322	8,321	8,322	8,322	8,321	8,322

Tabella 4.2: Per i primi dieci seed è riportato il numero di elementi validi, risultati e coppie uguali, le loro percentuali e i colpi clock necessari per 1.000.000 elementi

Questi sono i risultati ottenuti dai primi dieci seed, osservando vettori contenenti centomila e un milione di elementi. Come si può notare, prendendo numeri pseudo-casuali, si ha circa il 50% di dati validi. Abbiamo, dunque, una coppia identica in circa metà di questi, ovvero il 25% del totale. Inoltre, osserviamo che la quantità di tempo nei due casi risulta direttamente proporzionale: moltiplicando il numero di elementi di una costante K, il tempo impiegato è K-volte più grande. VALID indica il numero di basi uguali, medesimo valore di RESULT che indica il numero di risultati, ma, in fase di testing, questi non sempre combaciavano, e MATCH indica il numero di volte in cui sia base che bit sono uguali.

SEED[]	1	2	3	4	5	6	7	8	9	10	AVERAGE
VALID[bit]	1251017	1249416	0	1250552	0	0	1251087	1249296	1250885	0	1250375
RESULT[bit]	1251017	1249416	0	1250552	0	0	1251087	1249296	1250885	0	1250375
MATCH[bit]	625132	624888	0	624642	0	0	625021	624168	625765	0	624936
RESULT[%]	50,04%	49,98%	0,00%	50,02%	0,00%	0,00%	50,04%	50%	50,04%	0,00%	50,02%
MATCH[%]	25,01%	25,00%	0,00%	24,99%	0,00%	0,00%	25,00%	24,97%	25,03%	0,00%	24,99%
SIFT TIME[ms]	20,405288	20,40422892	0	20,42889846	0	0	20,43011585	20,430794	20,43157615	0	20,4218169

Tabella 4.3: Per alcuni dei primi dieci seed è riportato il numero di elementi validi, risultati e coppie uguali, le loro percentuali e i colpi clock necessari per 2.500.000 elementi

SEED[]	1	2	3	4	5	6	7	8	9	10	AVERAGE
VALID[bit]	1500126	1499799	1500117	1501971	1500119	0	0	0	0	0	1500426
RESULT[bit]	1500126	1499799	1500117	1501971	1500119	0	0	0	0	0	1500426
MATCH[bit]	749763	749990	749624	751901	749871	0	0	0	0	0	750229
RESULT [%]	50,04%	49,98%	0,00%	50,02%	0,00%	0,00%	50,04%	50%	50,04%	0,00%	50,02%
MATCH[%]	50,00%	49,99%	50,00%	50,07%	50,00%	0,00%	0,00%	0,00%	0,00%	0,00%	50,01%
SIFT TIME[ms]	29,877	29,859	29,877	29,881	29,878	0,000	0,000	0,000	0,000	0,000	29,874

Tabella 4.4: Per i primi cinque seed è riportato il numero di elementi validi, risultati e coppie uguali, le loro percentuali e i colpi clock necessari per 3.000.000 elementi

Sono stati successivamente effettuati gli stessi test aumentando la dimensione degli elementi da confrontare e non sono state riscontrate problematiche. Questo conclude l'implementazione del codice di sifting, che risulta perfettamente funzionante. Il codice sarà riutilizzato poi con l'introduzione della comunicazione ethernet (cfr. § 5.1).

Capitolo 5

Design e implementazione con connessione Ethernet

Nell'ultima parte della tesi è stata ricreata, nel modo più realistico possibile, una situazione simile al sifting nel protocollo BB84. Nello specifico, sono state utilizzate due schede distinte, che non condividono nessuna area di memoria, e l'unico mezzo di comunicazione presente è un cavo ethernet collegato tra le due. Questo simulerebbe, sempre con le approssimazioni citate nel capitolo precedente, come Alice comunichi le proprie informazioni a Bob in modo che questo possa poi eseguire il sifting. Per una questione di semplicità, Bob riceverà, oltre alla base di Alice anche il valore del bit. Benché l'instaurazione di questo tipo di comunicazione non rispecchi a pieno il protocollo, essa può comunque essere modificata senza difficoltà e aiuta ad analizzare meglio l'efficenza e i risultati del programma.

5.1 Implementazione della comunicazione

Si è deciso di adottare una gestione della memoria dinamica mediante puntatori. Infatti, per tenere conto dei dati inviati e copiare segmenti degli array, risulta molto più comodo ed efficiente questo approccio.
Vista la presenza di due schede, è stata necessaria la creazione di due programmi distinti con comportamenti molto diversi. La situazione che si viene perciò a creare può essere descritta come un sistema di comunicazione Client/Server. Difatti, possiamo considerare Alice il client, che si mette in contatto con Bob, passandogli le informazioni sui dati che ha generato. Successivamente, Bob processa le informazioni ricevute e manda la propria risposta al client.

Passiamo ora a un'analisi più dettagliata, in cui si decriverà passo passo quello che viene eseguito dal programma e il codice che lo permette:

- Il primo passaggio è quello di instaurare e confermare l'avvenuta connessione tra le due schede. Questo avviene grazie a strutture e metodi simili a quelli utilizzate per la programmazione mediante socket. Infatti, devono essere specificati l'indirizzo di arrivo e la porta da utilizzare, mentre le risposte avvengono principalmente mediante la comunicazione di errori. Se tutto va a buon fine, viene stampato a schermo la conferma che la connessione si è verificata senza imprevisti e il programma procede con il suo effettivo obiettivo.

- Il trasferimento dei dati in sé avviene attraverso due funzioni: la send_data() e my_tcp_sent_callback(). Nella pratica, viene usato il metodo tcp_sent, che richiama la callback, la quale, a sua volta, usa il metodo send_data() per inviare i dati desiderati. Questi processi innestati protrebbero sembrare confusionari e superflui, ma permettono in maniera molto efficiente e controllata di inizializzare un loop che ripete la chiamata a queste funzioni fino a che ogni elemento non è stato inviato.

- La funzione principale è la send_data(), in quanto è stata adattata alle esigenze del protocollo e ai limiti dell'ethernet. Nello specifico, lavorando con milioni di dati, questi non possono essere spediti in una singola volta, ma bisogna suddividere il vettore da inviare in blocchi che saranno poi spediti singolarmente. Pertanto, la prima parte della funzione si occupa di verificare quanta informazione è rimasta da inviare o, se necessario, usare la dimensione standard di 500 elementi: questa è arbitraria e modificabile, ma, per limitazioni dovute alla comunicazione ethernet, non può essere superiore a circa 1400 elementi. Il messaggio è salvato dentro a un'apposita variabile chiamata c_pcb tramite una funzione che, nel caso in cui ritorni un errore, permette di stamparlo a schermo. In caso di successo, l'informazione viene spedita ed è incrementato il puntatore ad Alice per non ripetere i dati.

- Ogni volta che il server riceve un pacchetto, questo viene processato e, nel nostro caso, il segmento di elementi di Alice viene copiato nell'apposito vettore. Il puntatore deve poi essere spostanto per l'equivalente della lunghezza del blocco ricevuto per evitare di sovvrascrivere i dati. Quando sono stati ricevuti tutti i blocchi, il server, ovvero Bob, realizza il sifting per ottenere il vettore contenente la posizione degli elementi validi e creare il suo array

di risultati. Questo avviene mediante il programma di cui si è discusso in precedenza (cfr. § 4.1), rivisitato in modo da adattarsi alle esigente della comunicazione ethernet.

- Infine, Bob invia il vettore degli elementi validi in modo analogo a quanto avvenuto in Alice tramite la funzione send_data() che spedisce blocchi di informazione. Alice, come avvenuto nel server, riceve i singoli segmenti e, una volta accertata la completa riuscita della comicazione, utilizza le posizioni valide per creare il suo array di risultati.

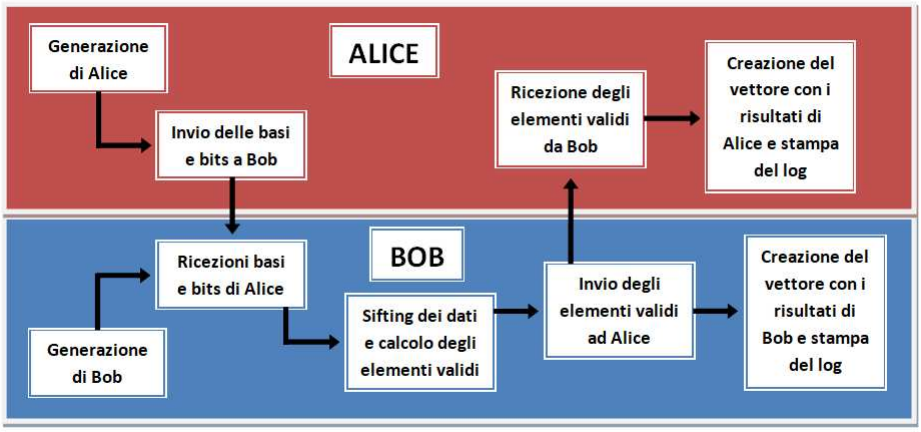

Figura 5.1: Schema a blocchi della comunicazione ed esecuzione dei programmi ethernet

Questi passaggi permettono una comunicazione efficiente e funzionante tra le due schede. Non sono stati riscontrati errori inaspettati o problematicità che non siano state risolte.

5.2 Risultati

Appurato il perfetto funzionamento del codice, sono stati fatti vari test di validazione con diverse dimensioni.

SEED[]	1	2	3	4	5	6	7	8	9	10	AVERAGE
VALID[bit]	50131	49911	49834	50029	100000	50278	50123	49974	50073	50044	50044
RESULT[bit]	50131	49911	49834	50029	100000	50278	50123	49974	50073	50044	50044
MATCH[bit]	24870	24973	24979	24931	100000	25022	25172	25039	24991	25096	25008
RESULT[%]	50,13%	49,91%	49,83%	50,03%	100,00%	50,28%	50,12%	49,97%	50,07%	50,04%	50,04%
MATCH[%]	24,87%	24,97%	24,98%	24,93%	100,00%	25,02%	25,17%	25,04%	24,99%	25,10%	25,01%
SIFT TIME[ms]	0,942	0,940	0,998	0,942	1,248	1,004	1,002	1,000	0,942	1,000	0,974
RESULT TIME[ms]	0,897	0,896	0,896	0,897	0,805	0,902	0,903	0,896	0,896	0,902	0,898

Tabella 5.1: Per i primi dieci seed è riportato il numero di elementi validi, risultati e coppie uguali, le loro percentuali e i colpi clock necessari per l'esecuzione del sifting e creazione dei risultati per 100.000 elementi

SEED[]	1	2	3	4	5	6	7	8	9	10	AVERAGE
VALID[bit]	125200	124973	125126	125070	250000	125215	125305	125117	124963	124867	125093
RESULT[bit]	125200	124973	125126	125070	250000	125215	125305	125117	124963	124867	125093
MATCH[bit]	62612	62400	62615	62529	250000	62580	62764	62335	62610	62586	62559
RESULT[%]	50,08%	49,99%	50,05%	50,03%	100,00%	50,09%	50,12%	50,05%	49,99%	49,95%	50,04%
MATCH[%]	25,04%	24,96%	25,05%	25,01%	100,00%	25,03%	25,11%	24,93%	25,04%	25,03%	25,02%
SIFT TIME[ms]	2,504	2,351	2,352	2,353	3,121	2,354	2,355	2,503	2,501	2,352	2,403
RESULT TIME[ms]	2,247	2,259	2,258	2,246	2,018	2,259	2,247	2,259	2,260	2,246	2,253

Tabella 5.2: Per i primi dieci seed è riportato il numero di elementi validi, risultati e coppie uguali, le loro percentuali e i colpi clock necessari per l'esecuzione del sifting e creazione dei risultati per 250.000 elementi

Nelle tabelle sono stati riportati, oltre al "seed" utilizzato, il numero di elementi validi, il numero di risultati ottenuti e le volte in cui sia la base che il bit corrispondevano. Inoltre, è stato riportato il tempo necessario alla generazione del vettore contenente la posizione degli elementi validi o, nel caso in cui si volesse conoscere solo il risultato, il tempo impiegato per quest'ultimo. Un'ultima nota riguarda il "seed" 5: infatti, come si può notare, si ha un successo del 100% sia nella generazione del risultato che della coppia base e bit. Questo non è un errore, ma la prova del corretto funzionamento del programma. Infatti, server e client usano due "seed" diversi e, nell'eventualità in cui questi usino il medesimo, come avviene per il numero 5, si andranno a generare le stesse identiche basi per Alice e Bob. Pertanto, arrivati al calcolo degli elementi validi, il programma dovrà confrontare due vettori perfettamente identici, portando in quello specifico caso a un successo del 100%. Essendo un dato "anomalo" che nel protocollo BB84 non può avvenire, questo è stato escluso dalla media dei valori a destra delle tabelle.

SEED[]	1	2	3	4	5	6	7	8	9	10	AVERAGE
VALID[bit]	249923	249881	250274	249994	500000	250191	250105	0	0	0	250061
RESULT[bit]	249923	249881	250274	249994	500000	250191	250105	0	0	0	250061
MATCH[bit]	125138	124832	125187	125011	500000	125119	125040	0	0	0	125055
RESULT[%]	49,98%	49,98%	50,05%	50,00%	100,00%	50,04%	50,02%	0,00%	0,00%	0,00%	50,01%
MATCH[%]	25,03%	24,97%	25,04%	25,00%	100,00%	25,02%	25,01%	0,00%	0,00%	0,00%	25,01%
SIFT TIME[ms]	5,004	4,703	5,006	5,003	6,242	4,708	5,007	0,000	0,000	0,000	4,905
RESULT TIME[ms]	4,521	4,493	4,521	4,494	4,038	4,521	4,496	0,000	0,000	0,000	4,508

Tabella 5.3: Per i primi dieci seed è riportato il numero di elementi validi, risultati e coppie uguali, le loro percentuali e i giri clock necessari per l'esecuzione del sifting e creazione dei risultati per 500.000 elementi

SEED[]	1	2	3	4	5	6	7	8	9	10	AVERAGE
VALID[bit]	500103	499950	499809	499841	1000000	0	0	0	0	0	499926
RESULT[bit]	500103	499950	499809	499841	1000000	0	0	0	0	0	499926
MATCH[bit]	249796	249931	250137	250012	1000000	0	0	0	0	0	249969
RESULT[%]	50,01%	50,00%	49,98%	49,98%	100,00%	0,00%	0,00%	0,00%	0,00%	0,00%	49,99%
MATCH[%]	24,98%	24,99%	25,01%	25,00%	100,00%	0,00%	0,00%	0,00%	0,00%	0,00%	25,00%
SIFT TIME[ms]	10,011	10,006	10,003	10,006	12,497	0,000	0,000	0,000	0,000	0,000	10,006
RESULT TIME[ms]	8,991	9,039	8,987	9,040	8,078	0,000	0,000	0,000	0,000	0,000	9,014

Tabella 5.4: Per i primi dieci seed è riportato il numero di elementi validi, risultati e coppie uguali, le loro percentuali e i giri clock necessari per l'esecuzione del sifting e creazione dei risultati per 1.000.000 elementi

Questi sono altri dati ricavati utilizzando dimensioni ben più grandi e simili a quelle

che ci si aspetta di trovare lavorando con il protocollo BB84. Va sottolineato che questi dati sono stati raccorti tramite un log generato da sia il client Alice e il server Bob, questi sono stati utilizzati in una versione modificata del tester di validazione in modo da processare i risultati di due diversi log e confrontarli.

SEED	1	2	3	4	5	6	7	8	9	10	AVERAGE
1°	11,970	11,967	12,965	12,965	13,987	12,967	13,972	13,961	13,962	11,968	13,068
2°	14,961	13,964	12,970	12,998	11,968	14,960	11,969	13,966	13,931	11,969	13,366
3°	13,964	11,969	11,969	14,960	14,959	13,003	13,960	13,964	11,970	12,966	13,368
4°	12,973	14,952	13,964	12,960	12,962	14,960	11,968	11,967	12,965	13,963	13,363
5°	12,973	14,952	13,964	12,960	12,962	14,960	11,968	11,967	12,965	13,963	13,363
AVERAGE	13,366	13,363	13,166	13,569	13,374	13,572	13,166	13,165	13,558	12,567	13,287

Tabella 5.5: Tempi di sifting e la loro media per i primi dieci seed eseguito su computer

Infine, nella tabella qui sopra riportata è stata segnata la quantità necessaria al completamento del sifting in millisecondi. Questi dati sono stati raccolti usando un programma simile al primo sifting, ma in grado di essere eseguito su un qualsiasi computer. Essendo quest'ultimo soggetto al sistema operativo, i tempi posso variare di molto, pertanto, per ognuno dei primi dieci "seed", sono stati raccolti cinque campioni. Nella riga più in basso si ha una media in base al "seed" utilizzato, mentre nella colonna più a destra si ha una media per il numero del tentativo. Infine, nella cella in comune in basso a destra vi è la media delle due sopracitate come dato ultimo. Si nota che il programma fatto funzionare sulla scheda richiede tempi decisamente inferiori rispetto al computer, che, oltre a essere più lento, risulta inconsistente coi tempi. Pertanto, l'uso di un SoC nel processo di sifting, nel protocollo BB84, migliora di circa il 25% l'efficienza in termini di tempo del programma.

Capitolo 6

Conclusioni

Con la presente tesi sono stati portati a termine tutti gli obiettivi prefissati. Innanzitutto, è stato implementato con successo il sifting dei dati attraverso un SoC. Grazie ai dati raccolti, è stato inoltre possibile confermare che il sifting eseguito su queste schede risulta nettamente più efficiente, in termini di tempo, rispetto alla sua controparte, eseguita invece su un comune computer.
Oltre a questo, è stata ricreata anche una situazione simile a quella che si verifica nel protocollo BB84 dove, al posto del canale quantistico, viene instaurata una comunicazione ethernet. Nonostante ciò non sia completamente accurato, questo ha permesso di verificare l'effettivo funzionamente del codice creato in presenza dei due attori Alice e Bob come entità distinte.
Al termine di ogni fase di testing non sono state riscontrate anomalie che possano intaccare il funzionamento del programma. Tutti i dati raccolti rispecchiano alla perfezione le aspettative, mentre i limiti fisici dettati dalla scheda, al di fuori del nostro controllo, sono stati discussi e illustrati.
Quanto ottenuto non è altro che un tassello dell'intero protocollo BB84. Si ricorda che il sifting è solo la prima fase del post-processing dei dati e che, oltre a questa, sarà necessaria una profonda e continua ricerca per permettere alla crittografia quantistica di migliorare ed espandersi fino a diventare uno strumento standard per la comunicazione moderna.

Bibliografia

[1] Colin P. Williams. *Explorations in Quantum Computing*, page 9–13. 2011.

[2] Quantum computing. https://www.sharetechnote.com/html/QC/Quantum Computing_Qbit.html. Visitato il: 11/09/2024.

[3] Alberto Carrasco-Casado, Veronica Marmol, and Natalia Denisenko. *Free-Space Quantum Key Distribution*, pages 589–607. 08 2016.

[4] Panagiotis Papanastasiou Stefano Pirandola Cosmo Lupo, Carlo Ottaviani. Parameter estimation with almost no public communication for continuous-variable quantum key distribution. *Reviews of Modern Physics*, 120(22):1–2, Jun 2018.

[5] Ahmed Abbas, Amr Goneid, and Sherif El-Kassas. Privacy amplification in quantum cryptography bb84 using combined univarsal2-truly random hashing. *International Journal of Information Network Security (IJINS)*, 3:98–115, 04 2014.

[6] Valerio Scarani, Helle Bechmann-Pasquinucci, Nicolas J. Cerf, Miloslav Duš ek, Norbert Lütkenhaus, and Momtchil Peev. The security of practical quantum key distribution. *Reviews of Modern Physics*, 81(3):1301–1350, sep 2009.

[7] SujayKumar Reddy M and Chandra Mohan B. Comprehensive analysis of bb84, a quantum key distribution protocol, 2023.

[8] Mateusz D. Zych Vasileios Mavroeidis, Kamer Vishi and Audun Jøsang. The impact of quantum computing on present cryptography. *The Science and Information Organization*, 9(3):6, Mar 2018.

[9] Soteris Kalogirou Adel Mellit. *Handbook of Artificial Intelligence Techniques in Photovoltaic Systems*, pages 267–329. 2022.

[10] Pynq-z2 reference manual v1.0. https://www.mouser.com/datasheet/2/744 /pynqz2_user_manual_v1_0-1525725.pdf. Visitato il: 12/08/2024.

[11] File:computer architecture block diagram.png. https://en.wikipedia.org /wiki/File:Computer_architecture_block_diagram.png. Visitato il: 27/08/2024.

[12] Peng Zhang. Chapter 5 - microprocessors. In Peng Zhang, editor, *Advanced Industrial Control Technology*, pages 155–214. William Andrew Publishing, Oxford, 2010.

Ringraziamenti

Ringrazio la mia famiglia per avermi sempre dato supporto in questo percorso di studio.
Ringrazio Tau e Sofia per essere stati il mio più grande sostegno nel corso di questi anni.
Ringrazio Gonzalo per la pazienza e l'aiuto dati durante lo sviluppo di questo progetto.