UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

"PROGRAMMAZIONE DI DRIVER DI RETE: RUST VS. LINGUAGGIO C"

Relatore: Prof. /Dott. Nicola Zingirian

Laureando: Tommaso Leoni

ANNO ACCADEMICO: 2023 / 2024
Data di laurea 16/11/2023

Abstract

La tesi affronta una discussione tecnica sull'analisi comparativa tra i linguaggi di programmazione C e Rust nel contesto dello sviluppo di driver per sistemi operativi. Attraverso l'implementazione di un programma generatore di pacchetti Ethernet IP e ICMP a basso livello, la tesi si concentra sull'analisi dell'uso della memoria e confronta le prestazioni dei due linguaggi. I risultati indicano che il linguaggio C, privo di alcuni costrutti di controllo presenti in Rust, presenta un vantaggio significativo in termini di istruzioni macchina e tempi di esecuzione, evidenziando il trade-off tra efficienza e sicurezza nella programmazione di basso livello. La discussione sottolinea la rilevanza di considerazioni pratiche nella scelta del linguaggio per lo sviluppo di driver di sistema operativo.

1. Introduzione

Nel contesto dello sviluppo di driver di sistema operativo, la scelta tra linguaggi di programmazione come C e Rust può influire notevolmente sulle prestazioni e sulla sicurezza. Questa tesi si propone di confrontare le prestazioni dei due linguaggi attraverso l'implementazione di un programma che genera pacchetti Ethernet IP e ICMP a basso livello per implementare il programma "ping" , con particolare attenzione all'analisi dell'uso della memoria. L'obiettivo è fornire una visione pratica delle differenze tra C e Rust nella gestione della memoria, con implicazioni dirette sulle prestazioni dei driver di sistema operativo.
I risultati della nostra analisi evidenziano chiaramente che il linguaggio C, a differenza di Rust, non implementa specifici costrutti di controllo che incidono significativamente sulla generazione di istruzioni macchina. Questa mancanza di costrutti aggiuntivi comporta un numero inferiore di istruzioni macchina necessarie per eseguire determinate operazioni rispetto a Rust.
Di conseguenza, si osserva un notevole vantaggio in termini di tempi di esecuzione per il linguaggio C. I test condotti hanno indicato una riduzione approssimativa del 30% nei tempi di esecuzione rispetto a Rust, evidenziando l'efficienza intrinseca di C nel contesto della programmazione a basso livello.
Questa constatazione può essere attribuita alla progettazione C, focalizzata sull'offrire un controllo diretto sull'hardware senza l'aggiunta di astrazioni di alto livello, mentre Rust, pur offrendo livelli di sicurezza superiori, introduce costrutti aggiuntivi che possono comportare un sovraccarico in termini di istruzioni macchina e tempi di esecuzione.
In definitiva, questi risultati sottolineano il trade-off tra sicurezza e efficienza nel contesto della programmazione di basso livello, indicando che C, per la sua semplicità e controllo diretto, può essere una scelta più rapida in termini di esecuzione rispetto a Rust, a discapito di alcune misure di sicurezza aggiuntive che Rust offre.

Linguaggio C

Il linguaggio C è stato sviluppato dai Bell Labs tra il 1969 e il 1973 per scrivere il sistema operativo UNIX ed è ancora oggi uno dei più utilizzati per le sue eccellenti performance e la sua capacità di interfacciarsi con configurazioni hardware a basso livello. (*WikipediaC*)

Rust

Rust è un linguaggio di programmazione che nasce come progetto di Mozilla Research nel 2006. Fin dalle sue prime versioni Rust è stato sviluppato con l'obiettivo di divenire un competitor di C/C++.
È attualmente un prodotto della Rust Foundation[1], nata all'indomani della crisi di Mozilla causata dalla pandemia di COVID-19, con l'obiettivo di portare avanti lo sviluppo del linguaggio assieme alla comunità open-source Rust . (*WikipediaRust*)

[1] La Rust Foundation è un'organizzazione non-profit finanziata da AWS, Huawei, Google, Microsoft e Mozilla

Feature di Rust

Di seguito vengono presentate alcune delle principali caratteristiche di Rust:

- **Zero cost abstraction**: le ottimizzazioni realizzate dal compilatore di Rust permettono la scrittura di codice che utilizzi astrazioni senza che questa scelta abbia un impatto significativo sulle prestazioni.
- **Ownership**: Rust impone delle rigide regole per la gestione della proprietà di una zona di memoria associata ad un valore. Ogni valore può essere posseduto da una singola variabile alla volta, che ne è responsabile; quando una variabile esce dallo scope la memoria associata al valore viene liberata automaticamente; le variabili possono condividere riferimenti, è il linguaggio a garantire che non ci siano riferimenti non validi o accessi concorrenti.
- **Algebraic data types**: in Rust è possibile definire enumerazioni, strutture, tuple e utilizzare il pattern matching.
- **Polymorphism**: Rust supporta la programmazione generica basata sui tratti, che sono simili alle interfacce e consentono di definire comportamenti comuni implementati da tipi diversi.

2. Ping

Una delle funzionalità base richieste da un driver di rete è quella di effettuare un ping.
La procedura richiede la creazione di un pacchetto ICMP di tipo *echo request* che viene inviato ad un dispositivo, attendendo poi in risposta un pacchetto ICMP di tipo *echo reply*.
Abbiamo quindi deciso di prendere la scrittura e lettura di questi pacchetti come base per il confronto di performance.
I pacchetti non vengono inviati, poiché risulta chiaro che andando ad inviare grosse quantità di dati attraverso la rete si avrebbe un effetto "collo di bottiglia" che falserebbe i risultati dell'esperimento.

Generazione e lettura pacchetti

Per poter attraversare la rete ethernet i pacchetti ICMP vengono incapsulati in un **pacchetto ip**, che viene a sua volta inserito in un **pacchetto ethernet**.
La costruzione del pacchetto avviene attraverso la definizione di tre strutture, che contengono campi mostrati in *Figura 1*, *Figura 2* e *Figura 3*.
Vengono prese in considerazione sia la **generazione**, ossia il tempo impiegato per la scrittura dei campi all'interno di un buffer, che la **lettura**, ossia il tempo impiegato ad estrarre in variabili distinte il contenuto del buffer stesso.

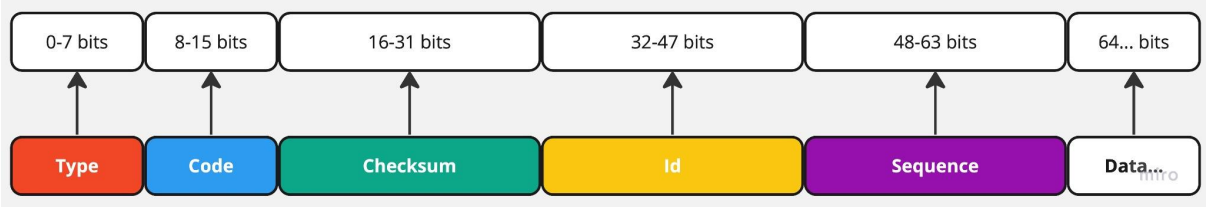


Figura 1: Campi pacchetto ICMP

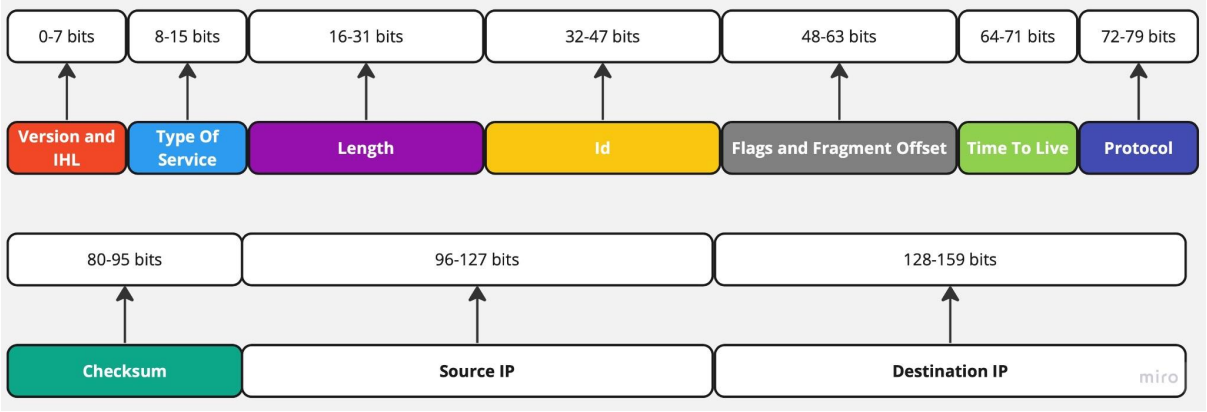


Figura 2: Campi pacchetto IP

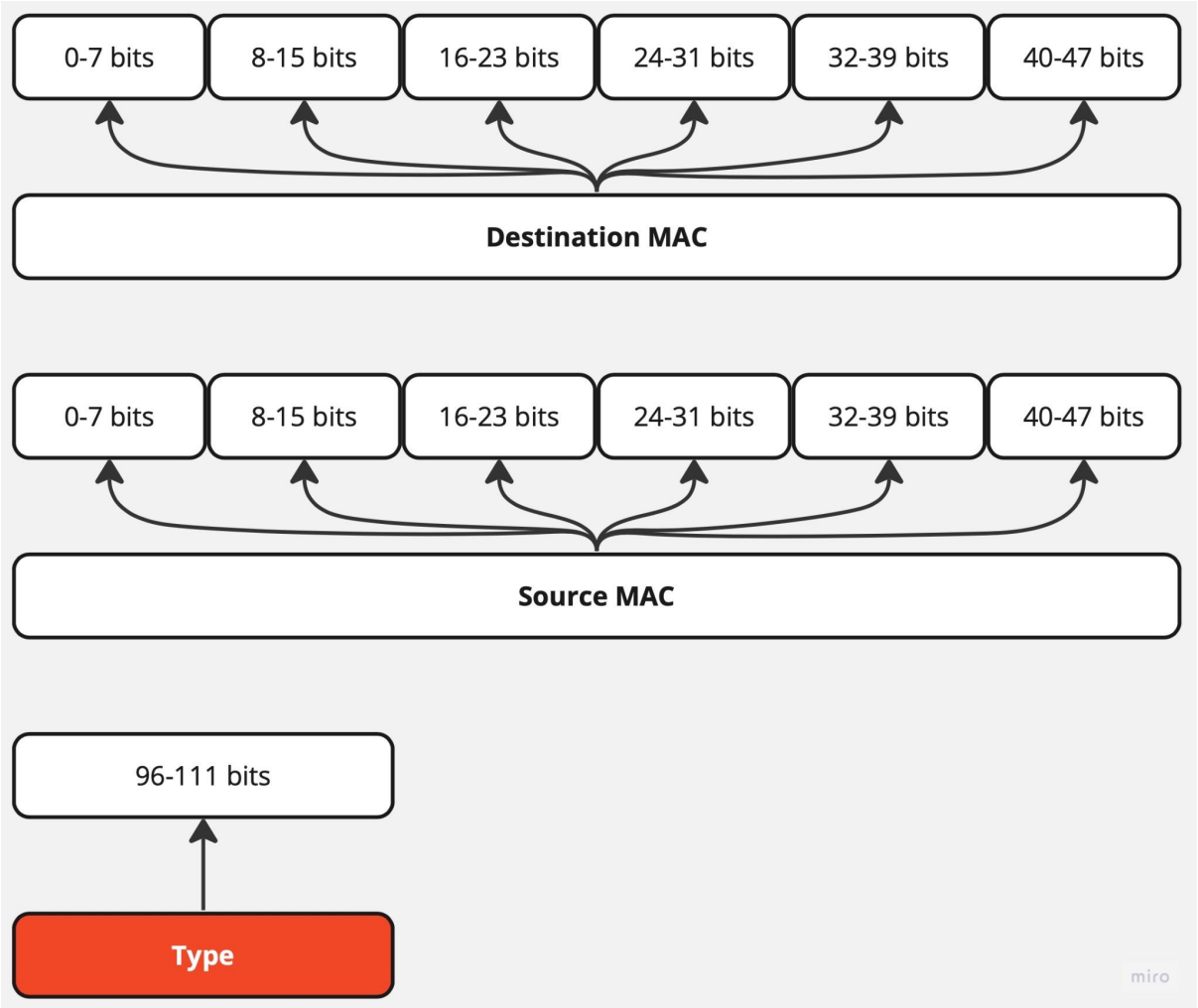


Figura 3: Campi pacchetto Ethernet

3. Programma C

In questa sezione verrà presentato il programma realizzato in linguaggio C. Verranno prima presentate la strutture e i metodi realizzati per il loro riempimento e poi la funzione main contenente il codice necessario alla creazione e lettura dei pacchetti.
Per il codice completo fare riferimento a Appendice A - Programma completo in C.

Librerie

Le librerie standard utilizzate nel programma scritto in linguaggio C sono:

- **<arpa/inet.h>**: contiene i metodi *htons()* e *ntohs()*, che si occupano di convertire valori a 16 bit (short) dall'ordine dell'host all'ordine di rete e viceversa.

Checksum

Prima di iniziare la presentazione vera e propria delle strutture è importante presentare l'algoritmo utilizzato per il calcolo della *checksum*. Questa è il complemento a uno della somma dei byte di ciascun pacchetto, raggruppati in interi da 16 bit, ed è usata per effettuare un controllo di integrità dei pacchetti.
La funzione presentata calcola il checksum a partire dal parametro *b*, che è un puntatore generico, per una lunghezza pari al parametro *s*.

```
unsigned short checksum ( void * b, int s)
{
    unsigned short * p;
    int i;
    unsigned int tot=0;
    p = (unsigned short *) b;
    for (i=0; i<s/2; i++){
            tot +=ntohs(p[i]);
            if ( tot & 0x10000) tot = (tot&0xFFFF) + 1;
    }
    if ( i*2 != s ){
            tot+=ntohs(p[i])&0xFF00;
            if ( tot & 0x10000) tot = (tot&0xFFFF) + 1;
    }
    return  (0xFFFF-(unsigned short)tot);
}
```

Codice 1: Funzione checksum in C

Strutture

Pacchetto ICMP

La struttura utilizzata per la realizzazione di un pacchetto ICMP è denominata *icmp_packet*:

```
struct icmp_packet {
    unsigned char type;
    unsigned char code;
    unsigned short checksum;
    unsigned short id;
    unsigned short seq;
    unsigned char data[1];
};
```

Codice 2: Struttura pacchetto ICMP in C

Per riempire la struttura è stata definita una funzione chiamata *forge_icmp* che prende come parametro un puntatore ad una struttura di tipo *icmp_packet* (*icmp*) e va a scrivere direttamente all'interno dei suoi campi, come mostrato di seguito:

```
int forge_icmp(struct icmp_packet * icmp){
int i;
icmp->type = 8;
icmp->code = 0;
icmp->checksum = 0;
icmp->id = 0x1234;
icmp->seq = 0;
for(i=0;i<32;i++) icmp->data[i]=i;
icmp->checksum = htons(checksum(icmp,40));
return 40;
}
```

Codice 3: Funzione creazione pacchetto ICMP in C

È importante notare che il campo *type* viene impostato ad 8, che corrisponde ad un pacchetto di tipo *echo request*, mentre la sezione dati viene riempita con i primi 32 interi a partire da 0. Viene ritornata la lunghezza in byte del pacchetto creato, ossia 40 (8 di intestazione e 32 di dati).

Pacchetto IP

La struttura utilizzata per la realizzazione di un pacchetto IP è denominata *ip_datagram*:

```
struct ip_datagram {
    unsigned char ver_ihl;
```

```
    unsigned char tos;
    unsigned short len;
    unsigned short id;
    unsigned short flags_offs;
    unsigned char ttl;
    unsigned char proto;
    unsigned short checksum;
    unsigned int src;
    unsigned int dst;
    unsigned char payload[1];
};
```

Codice 4: Struttura pacchetto IP in C

Per riempire la struttura è stata definita una funzione chiamata *forge_ip* che prende come parametro un puntatore ad una struttura di tipo *ip_datagram* (*ip*) e va a scrivere direttamente all'interno dei suoi campi. Oltre a questo primo parametro vengono richiesti anche un puntatore all'ip di destinazione (*dst*), la lunghezza del payload (*payloadlen*) e il protocollo (*proto*).

```
void forge_ip(struct ip_datagram * ip, unsigned char* dst,  unsigned
short payloadlen, unsigned char proto  )
{
    ip->ver_ihl = 0x45;
    ip->tos = 0;
    ip->len = htons(payloadlen+20);
    ip->id = 0xABCD;
    ip->flags_offs = 0;
    ip->ttl = 128;
    ip->proto = proto;
    ip->checksum = 0;
    ip->src = *(unsigned int *)myip;
    ip->dst = *(unsigned int *)dst;
    ip->checksum = htons(checksum(ip,20));
}
```

Codice 5: Funzione creazione pacchetto IP in C

È importante notare che il campo *ver_ihl*, contiene nel bit alto il valore 4, per indicare un pacchetto ipv4, mentre nel bit basso contiene il valore 5, che indica un'header del pacchetto di lunghezza pari a 20 byte.

Pacchetto Ethernet

La struttura utilizzata per la realizzazione di un pacchetto Ethernet è denominata *ethernet_frame*:

```c
struct ethernet_frame {
    unsigned char dst[6];
    unsigned char src[6];
    unsigned short type;
    unsigned char payload[1];
};
```

Codice 6: Struttura pacchetto Ethernet in C

Per riempire la struttura è stata definita una funzione chiamata *forge_eth* che prende come parametro un puntatore ad una struttura di tipo *ethernet_frame* e va a scrivere direttamente all'interno dei suoi campi. Oltre a questo primo parametro vengono richiesti anche un puntatore al MAC address di destinazione (*dst*) e il tipo di frame (*type*).

```c
void forge_eth( struct ethernet_frame * eth, unsigned char * dst,
unsigned short type)
{
    for(int i=0;i<6;i++) eth->dst[i] = dst[i];
    for(int i=0;i<6;i++) eth->src[i] = mymac[i];
    eth->type = htons(type);
}
```

Codice 7: Funzione creazione pacchetto Ethernet in C

Funzione Main

```c
unsigned char mymac[6] = { 0xf2,0x3c,0x91,0xdb,0xc2,0x98} ;
unsigned char myip[4] = { 88,80,187,84 };
unsigned char buf[1500];

int main(){
    int i,len;
    unsigned char destip[4] = {147,162,2,100};
    unsigned char destmac[6] = {0x00, 0x1A, 0x2B, 0x3C, 0x4D, 0x5E};
    for(int j=0; j<1; j++) {
        struct icmp_packet *icmp;
        struct ip_datagram * ip;
        struct ethernet_frame * eth;

        eth = (struct ethernet_frame *) buf;
        ip = (struct ip_datagram *) (eth->payload);
        icmp = (struct icmp_packet *) (ip->payload);

        len = forge_icmp(icmp);
        forge_ip(ip,destip,len, 1);
        forge_eth(eth,destmac,0x0800);
```

```
        unsigned char eth_packet_dst[6];
        for(i=0; i<6; i++) {
            eth_packet_dst[i] = eth->dst[i];
        }
        unsigned char eth_packet_src[6];
        for(i=0; i<6; i++) {
            eth_packet_src[i] = eth->src[i];
        }
        unsigned short eth_packet_type = eth->type;
        unsigned char ip_packet_ver_ihl = ip->ver_ihl;
        unsigned char ip_packet_tos = ip->tos;
        unsigned short ip_packet_len = ip->len;
        unsigned short ip_packet_id = ip->id;
        unsigned short ip_packet_flags_off = ip->flags_offs;
        unsigned char ip_packet_ttl = ip->ttl;
        unsigned char ip_packet_proto = ip->proto;
        unsigned short ip_packet_checksum = ip->checksum;
        unsigned int ip_packet_src = ip->src;
        unsigned int ip_packet_dst = ip->dst;
        unsigned char icmp_packet_type = icmp->type;
        unsigned char icmp_packet_code = icmp->code;
        unsigned short icmp_packet_checksum = icmp->checksum;
        unsigned short icmp_packet_id = icmp->id;
        unsigned short icmp_packet_seq = icmp->seq;
        unsigned char icmp_packet_data[32];
        for(i=0; i<32; i++) {
            icmp_packet_data[i] = icmp->data[i];
        }
    }
}
```

Codice 8: Funzione main in C

Le prime tre variabili statiche rappresentano rispettivamente:

- **mymac**: è un indirizzo MAC generato casualmente che rappresenta l'indirizzo della macchina su cui viene eseguito il programma.
- **myip**: è un indirizzo ip generato casualmente che rappresenta l'indirizzo della macchina su cui viene eseguito il programma.
- **buf**: è l'area di memoria dove vengono scritti i pacchetti

All'interno dellan funzione main sono dichiarate altre due variabili:

- **destip**: è un indirizzo ip generato casualmente che rappresenta l'indirizzo della macchina di destinazione
- **destmac**: è un indirizzo MAC generato casualmente che rappresenta l'indirizzo della macchina di destinazione

Queste ultime variabili sono state create all'interno della funzione main per rappresentare il fatto in un'applicazione reale il loro valore potrebbe cambiare a tempo di esecuzione.

4. Programma Rust

In questa sezione verrà presentato il programma realizzato in Rust. Verranno prima presentate la strutture e i metodi realizzati per il loro riempimento e poi la funzione main contenente il codice necessario alla creazione e lettura dei pacchetti.
È importante tenere presente la distinzione tra codice **safe** e codice **unsafe**. Mentre nel codice safe la sicurezza dei riferimenti e degli accessi in memoria è garantita dal linguaggio, nelle sezioni di codice all'interno della sezione *unsafe* è compito del programmatore garantire che non ci siano errori nella gestione delle risorse.
Nel codice presentato in questa sezione ci saranno diverse istruzioni che necessitano di essere inserite in una sezione delimitata dalla keyword *unsafe*, che vengono utilizzate per la manipolazione dei puntatori grezzi.
Per il codice completo fare riferimento a Appendice B - Programma completo in Rust.

Librerie

Le librerie standard utilizzate nel programma Rust sono:

- **std::ptr**: necessaria per l'utilizzo di puntatori grezzi all'interno del programma.
- **std::mem**: necessaria per il metodo *size_of()*, che misura la lunghezza in byte di una struttura.

Checksum

Prima di iniziare la presentazione vera e propria delle strutture è importante presentare l'algoritmo utilizzato per il calcolo della *checksum*. Questa è il complemento a uno della somma dei byte di ciascun pacchetto, raggruppati in interi da 16 bit, ed è usata per effettuare un controllo di integrità dei pacchetti inviati e ricevuti e viene calcolata per ciascuna struttura.
La funzione presentata calcola il checksum a partire dal parametro *reference*, che è un puntatore ad un unsigned char, per una lunghezza pari al parametro *byte_size*.

```
fn checksum(reference: *const u8, byte_size: u32) -> u16 {
    unsafe { asm!("nop");}
    let tmp: *const u16 = reference as *const u16;
    let mut tot = 0u32;
    let mut last_index: u32 = 0;
    for i in 0..(byte_size/2) {
        tot += unsafe { u32::from(tmp.wrapping_add(i as
usize).read()).to_le()};
        if(tot & 0x10000) != 0 { tot = (tot & 0xffff) + 1;}
        last_index += 1;
    }
    if last_index*2 != byte_size {
```

```
        tot += unsafe{ u32::from(tmp.wrapping_add(last_index as
usize).read()).to_le() & 0xff00};
        if(tot & 0x10000) == 1 { tot = (tot & 0xffff) + 1;}
    }
    return 0xffff - (tot as u16);
    unsafe { asm!("nop");}
}
```

Codice 9: Funzione checksum in Rust

Strutture

Prima di ciascuna delle strutture che verranno presentate è stata dichiarata la direttiva *#[repr(C, packed)]*. Questa direttiva indica che l'organizzazione in memoria delle strutture è identica a quella di C e che non vengono inseriti dal linguaggio byte di padding per allineare i loro campi. La direttiva è necessaria per poter avere una corretta serializzazione delle strutture nel buffer di byte.

Pacchetto ICMP

La struttura utilizzata per la realizzazione di un pacchetto ICMP è denominata *IcmpPacket*:

```
#[repr(C, packed)]
struct IcmpPacket<T> {
    r#type:u8,
    code:u8,
    checksum:u16,
    id:u16,
    seq:u16,
    data: T,
}
```

Codice 10: Struttura pacchetto ICMP in Rust

Possiamo notare come, per poter realizzare un pacchetto con lunghezza variabile, sia stato definito un tipo generico *T* per il campo *data*.
Per riempire la struttura è stata definita una funzione chiamata *forge_icmp* che prende come parametro un riferimento mutabile ad una struttura di tipo *icmp_packet* il cui tipo generico è un array di 32 byte. La funzione va a scrivere direttamente all'interno dei campi della struttura, come mostrato di seguito:

```
fn forge_icmp(reference: & mut IcmpPacket<[u8;32]>) {
    reference.r#type=8;
    reference.code=0;
    reference.checksum=0;
    reference.id=0x1234;
    reference.seq=0;
    for i in 0..32 {
        reference.data[i] = i as u8;
    }
    reference.checksum=checksum(ptr::addr_of!(reference.r#type),
40).to_be();
}
```

Codice 11: Funzione creazione pacchetto ICMP in Rust

È importante notare che il campo *type* viene impostato ad 8, che corrisponde al tipo *echo request*, mentre la sezione dati viene riempita con i primi 32 interi a partire da 0.

Pacchetto IP

La struttura utilizzata per la realizzazione di un pacchetto IP è denominata *IpDatagram*:

```
#[repr(C, packed)]
struct IpDatagram<T> {
    ver_ihl:u8,
    tos:u8,
    len:u16,
    id:u16,
    flags_offs:u16,
    ttl:u8,
    proto:u8,
    checksum:u16,
    src:u32,
    dst:u32,
    payload: T,
}
```

Codice 12: Struttura pacchetto IP in Rust

Anche in questo caso viene definito un tipo generico *T* per permettere di avere un *payload* di tipo variabile.
Per riempire la struttura è stata definita una funzione chiamata *forge_ip* che prende come parametro un riferimento mutabile ad una struttura di tipo *ip_datagram* e va a scrivere direttamente all'interno dei suoi campi. Oltre a questo primo parametro vengono richiesti anche un puntatore all'ip di destinazione (*dst*), la lunghezza del payload (*payloadlen*) e il protocollo (*proto*).

```
fn forge_ip<T>(reference: & mut IpDatagram<T>, dst: &[u8;4], payloadlen:
u16, proto: u8) {
    reference.ver_ihl = 0x45;
    reference.tos = 0;
    reference.len = (payloadlen + 20).to_be();
    reference.id = 0xABCD;
    reference.flags_offs = 0;
    reference.ttl = 128;
    reference.proto = proto;
    reference.checksum = 0;
    reference.src = u32::from_le_bytes(MYIP);
    reference.dst = u32::from_le_bytes(*dst);
    reference.checksum=checksum(ptr::addr_of!(reference.ver_ihl),
20).to_be();
```

```
}
```

Codice 13: Funzione creazione pacchetto IP in Rust

È importante notare che il campo *ver_ihl*, contiene nel bit alto 4, per indicare un pacchetto ipv4, mentre nel bit basso ha un 5, che indica un'header di lunghezza pari a 20 byte.

Pacchetto Ethernet

La struttura utilizzata per la realizzazione di un pacchetto Ethernet è denominata *EthernetFrame*:

```
#[repr(C, packed)]
struct EthernetFrame<T> {
    dst:[u8;6],
    src:[u8;6],
    r#type:u16,
    payload: T,
}
```

Codice 14: Struttura pacchetto Ethernet in Rust

Anche in quest'ultimo caso viene definito un tipo generico *T* per permettere di avere un *payload* di tipo variabile.
Per riempire la struttura è stata definita una funzione chiamata *forge_eth* che prende come parametro un riferimento mutabile ad una struttura di tipo *ethernet_frame* e va a scrivere direttamente all'interno dei suoi campi. Oltre a questo primo parametro vengono richiesti anche un puntatore al MAC address di destinazione (*dst*) e il tipo di frame (*type*).

```
fn forge_eth<T>(reference: & mut EthernetFrame<T>, dst: &[u8;6], r#type:
u16) {
    reference.dst = *dst;
    reference.src = MYMAC;
    reference.r#type = r#type.to_be();
}
```

Codice 15: Funzione creazione pacchetto Ethernet in Rust

Funzione Main

```
static MYMAC: [u8;6] = [0xF2, 0x3C, 0x91, 0xDB, 0xC2, 0x98];
static MYIP: [u8;4] = [88, 80, 187, 84];
static mut BUFFER: [u8;1500] = [0;1500];

fn main() {
    let dstip: [u8;4] = [147,162,2,100];
    let dstmac: [u8;6] = [0x00, 0x1A, 0x2B, 0x3C, 0x4D, 0x5E];
```

```
    let len =
size_of::<EthernetFrame<IpDatagram<IcmpPacket<[u8;32]>>>>();
    for i in 0..1 {
        let ptr = unsafe {
BUFFER[0..len].align_to_mut::<EthernetFrame<IpDatagram<IcmpPacket<[u8;32
]>>>>()};
        let ethernet_frame = &mut ptr.1[0];
        forge_eth::<IpDatagram<IcmpPacket<[u8;32]>>>(ethernet_frame,
&dstmac, 0x0800);
        forge_ip::<IcmpPacket<[u8;32]>>(&mut ethernet_frame.payload,
&dstip, 40, 1);
        forge_icmp(&mut ethernet_frame.payload.payload);

        let eth_packet_dst = ethernet_frame.dst;
        let eth_packt_src = ethernet_frame.src;
        let eth_packet_type = ethernet_frame.r#type;
        let ip_packet_ver_ihl = ethernet_frame.payload.ver_ihl;
        let ip_packet_tos = ethernet_frame.payload.tos;
        let ip_packet_len = ethernet_frame.payload.len;
        let ip_packet_id = ethernet_frame.payload.id;
        let ip_packet_flags_off = ethernet_frame.payload.flags_offs;
        let ip_packet_ttl = ethernet_frame.payload.ttl;
        let ip_packet_proto = ethernet_frame.payload.proto;
        let ip_packet_checksum = ethernet_frame.payload.checksum;
        let ip_packet_src = ethernet_frame.payload.src;
        let ip_packet_dst = ethernet_frame.payload.dst;
        let icmp_packet_type = ethernet_frame.payload.payload.r#type;
        let icmp_packet_code = ethernet_frame.payload.payload.code;
        let icmp_packet_checksum =
ethernet_frame.payload.payload.checksum;
        let icmp_packet_id = ethernet_frame.payload.payload.id;
        let icmp_packet_seq = ethernet_frame.payload.payload.seq;
        let icmp_packet_data = ethernet_frame.payload.payload.data;
    }
}
```

Codice 16: Funzione main in Rust

Le prime tre variabili statiche rappresentano rispettivamente:

- **MYMAC**: è un indirizzo MAC generato casualmente che rappresenta l'indirizzo della macchina su cui viene eseguito il programma.
- **MYIP**: è un indirizzo ip generato casualmente che rappresenta l'indirizzo della macchina su cui viene eseguito il programma.
- **BUFFER**: è l'area di memoria dove vengono scritti i pacchetti.

All'interno dellan funzione main sono dichiarate altre due variabili:

- **destip**: è un indirizzo ip generato casualmente che rappresenta l'indirizzo della macchina di destinazione

- **destmac**: è un indirizzo MAC generato casualmente che rappresenta l'indirizzo della macchina di destinazione

Queste ultime variabili sono state create all'interno della funzione main per rappresentare il fatto in un'applicazione reale il loro valore potrebbe cambiare a tempo di esecuzione.

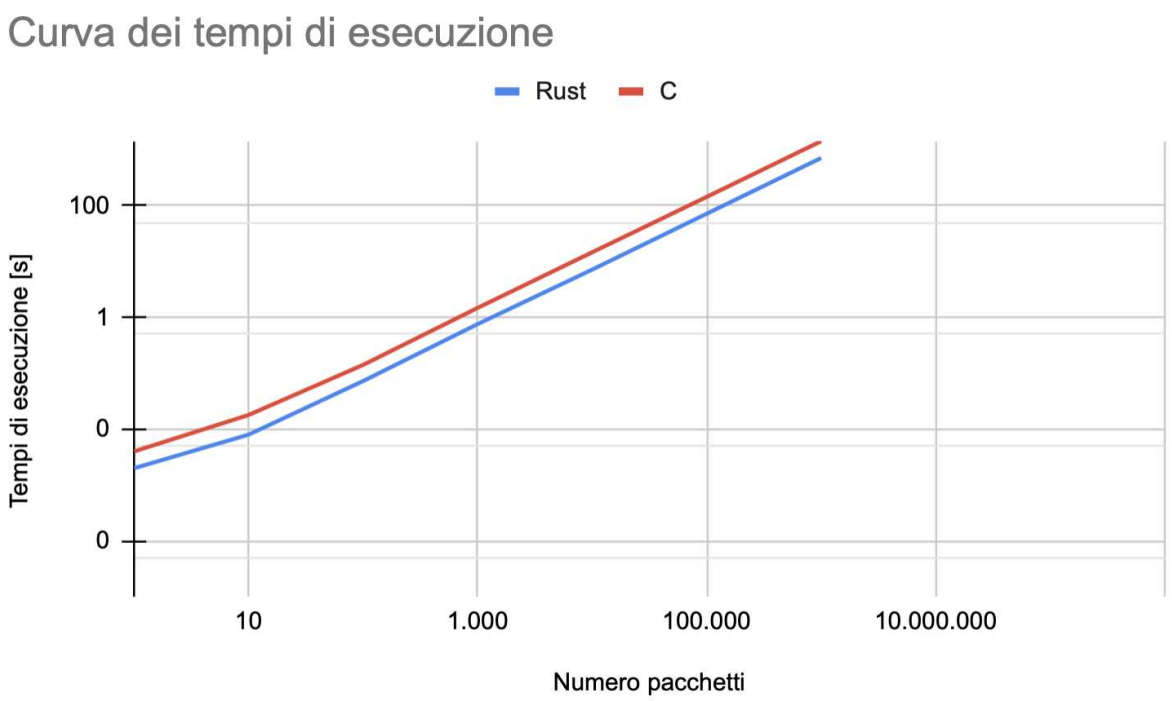
5. Risultati

Tempi di esecuzione

Le performance dei programmi presentati ai capitoli Programma C e Programma Rust sono state eseguite su una macchina virtuale di un sistema operativo Ubuntu 23.04 dopo essere stati compilati usando il comando *gcc file.c -o file* per C e *rustc file.rs -o file* per Rust.
Per misurare i tempi di esecuzione è stato utilizzato il comando *time*, disponibile nei sistemi Linux, che fornisce tre misurazioni: real, user e sys. Di seguito vengono riportati i risultati ottenuti nel campo **real**, che rappresenta il tempo totale trascorso per l'esecuzione dell'istruzione fornita come argomento, variando il numero di pacchetti in maniera logaritmica.

Numero pacchetti	Linguaggio C [s]	Rust [s]
1	0,001	0,003
10	0,001	0,002
100	0,002	0,003
1.000	0,002	0,004
10.000	0,008	0,018
100.000	0,073	0,142
1.000.000	0,764	1,491
10.000.000	7,225	14,763
100.000.000	72,462	145,000
1.000.000.000	722,664	1.444,057

Il grafico sottostante mostra le curve dei tempi di esecuzione in funzione del numero di pacchetti:



Come si può evincere dalla tabella e dalle curve del grafico il tempo di esecuzione di Rust si attesta a circa il 50% di quello del linguaggio C. È quindi chiaro che la sicurezza fornita da Rust abbia un costo in termini di performance.

Analisi del codice Assembly

Per ricercare le motivazioni della differenza in termini di performance tra i due linguaggi è stato analizzato un segmento dell'output in linguaggio **assembly arm** dei due programmi.
Nello specifico si è presa in esame la funzione *forge_icmp* presentata in C e in Rust rispettivamente nei segmenti di codice *Codice 3: Funzione creazione pacchetto ICMP in C* e *Codice 11: Funzione creazione pacchetto ICMP in Rust*, andando ad estrarla dal listato assembly ottenuto usando i comandi *gcc -S file.c* e *rustc –emit asm file.rs*.

Assembly prodotto da C

L'assembly prodotto da C è il seguente:

```
forge_icmp:
.LFB7:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
```

```
        subq    $32, %rsp
        movq    %rdi, -24(%rbp)
        movq    -24(%rbp), %rax
        movb    $8, (%rax)
        movq    -24(%rbp), %rax
        movb    $0, 1(%rax)
        movq    -24(%rbp), %rax
        movw    $0, 2(%rax)
        movq    -24(%rbp), %rax
        movw    $4660, 4(%rax)
        movq    -24(%rbp), %rax
        movw    $0, 6(%rax)
        movl    $0, -4(%rbp)
        jmp     .L8
.L9:
        movl    -4(%rbp), %eax
        movl    %eax, %ecx
        movq    -24(%rbp), %rdx
        movl    -4(%rbp), %eax
        cltq
        movb    %cl, 8(%rdx,%rax)
        addl    $1, -4(%rbp)
.L8:
        cmpl    $31, -4(%rbp)
        jle     .L9
        movq    -24(%rbp), %rax
        movl    $40, %esi
        movq    %rax, %rdi
        call    checksum
        movzwl  %ax, %eax
        movl    %eax, %edi
        call    htons@PLT
        movq    -24(%rbp), %rdx
        movw    %ax, 2(%rdx)
        movl    $40, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE7:
        .size   forge_icmp, .-forge_icmp
        .globl  forge_ip
        .type   forge_ip, @function
```

Codice 17: Funzione forge_icmp in assembly prodotto da C

Le istruzioni presenti sono quelle strettamente necessarie a scrivere nel riferimento alla

struttura *icmp_packet* (vedi *Codice 2: Struttura pacchetto ICMP in C*), passato come parametro attraverso il registro *rdi*. Vengono chiamate anche le funzioni *checksum* e *htons* per la scrittura del campo *checksum* del pacchetto.

Assembly prodotto da Rust

L'assembly prodotto da Rust è il seguente:

```
.section
.text._ZN13packet_reader10forge_icmp17hff8d298e596e55a7E,"ax",@progbits
        .p2align        4, 0x90
        .type
_ZN13packet_reader10forge_icmp17hff8d298e596e55a7E,@function
_ZN13packet_reader10forge_icmp17hff8d298e596e55a7E:
        .cfi_startproc
        subq    $72, %rsp
        .cfi_def_cfa_offset 80
        movq    %rdi, 8(%rsp)
        movb    $8, (%rdi)
        movb    $0, 1(%rdi)
        movw    $0, 2(%rdi)
        movw    $4660, 4(%rdi)
        movw    $0, 6(%rdi)
        movq    $0, 16(%rsp)
        movq    $32, 24(%rsp)
        movq    16(%rsp), %rdi
        movq    24(%rsp), %rsi
        callq
_ZN63_$LT$I$u20$as$u20$core..iter..traits..collect..IntoIterator$GT$9int
o_iter17h6c507eedb0449aa2E
        movq    %rax, 32(%rsp)
        movq    %rdx, 40(%rsp)

.LBB48_1:
        leaq    32(%rsp), %rdi
        callq
_ZN4core4iter5range101_$LT$impl$u20$core..iter..traits..iterator..Iterat
or$u20$for$u20$core..ops..range..Range$LT$A$GT$$GT$4next17ha7a6c8d330376
f52E
        movq    %rdx, 56(%rsp)
        movq    %rax, 48(%rsp)
        cmpq    $0, 48(%rsp)
        jne     .LBB48_3
        movq    8(%rsp), %rdi
        movl    $40, %esi
        callq   _ZN13packet_reader8checksum17ha1c4aebbda7b3c35E
```

```
        movw    %ax, %cx
        movq    8(%rsp), %rax
        rolw    $8, %cx
        movw    %cx, 70(%rsp)
        movw    70(%rsp), %cx
        movw    %cx, 2(%rax)
        addq    $72, %rsp
        .cfi_def_cfa_offset 8
        retq
.LBB48_3:
        .cfi_def_cfa_offset 80
        movq    56(%rsp), %rax
        movq    %rax, (%rsp)
        cmpq    $32, %rax
        setb    %al
        testb   $1, %al
        jne     .LBB48_4
        jmp     .LBB48_5
.LBB48_4:
        movq    8(%rsp), %rax
        movq    (%rsp), %rcx
        movb    %cl, %dl
        movb    %dl, 8(%rax,%rcx)
        jmp     .LBB48_1
.LBB48_5:
        movq    (%rsp), %rdi
        leaq    .L__unnamed_11(%rip), %rdx
        movq
_ZN4core9panicking18panic_bounds_check17h871dd1e68fdb74d2E@GOTPCREL(%rip
), %rax
        movl    $32, %esi
        callq   *%rax
        ud2
.Lfunc_end48:
        .size   _ZN13packet_reader10forge_icmp17hff8d298e596e55a7E,
.Lfunc_end48-_ZN13packet_reader10forge_icmp17hff8d298e596e55a7E
        .cfi_endproc
```

Codice 18: Funzione forge_icmp in assembly prodotto da Rust

Anche in questo caso abbiamo la scrittura dei campi nel riferimento alla struttura (vedi *Codice 10: Struttura pacchetto ICMP in Rust*) passato attraverso il registro *rdi*.
Quando si arriva alla scrittura del campo *data* l'assembly di Rust però procede alla creazione di un iteratore. Questo iteratore è composto dagli indici degli estremi (nel nostro caso 0 e 32) e dall'indice corrente e viene utilizzato per scorrere l'array attraverso il metodo *next()*, che verifica se il prossimo indice sia entro gli estremi prima di ritornare. Questi confronti,

soprattutto quando il numero di pacchetti aumenta esponenzialmente, hanno un impatto sulle performance.
Un ulteriore confronto viene fatto chiamando *panic_bounds_check*, che verifica che gli accessi siano entro gli estremi.

Conclusioni

Nonostante Rust abbia un controllo più serrato a livello di sicurezza del linguaggio C, garantendo conversioni sicure e accessi in memoria validi, queste verifiche comportano un'inevitabile riduzione delle sue performance.
Inoltre, per poter eseguire operazioni più mirate sulla memoria, risulta spesso necessario bypassare la sicurezza attraverso l'utilizzo della keyword *unsafe*.

6. Altre implementazioni

Al tempo di redazione di questo documento Rust non dispone di un'implementazione standard delle strutture dei pacchetti di rete. Esistono però delle **crate** (il nome che viene dato alle librerie per questo linguaggio) che forniscono le funzioni necessarie.

Crate pnet

La più utilizzata è la crate **pnet**, che fornisce un'API cross-platform per la gestione del networking di basso livello. (*PnetRust*)
All'interno della crate si possono trovare delle strutture che realizzano pacchetti ICMP, IP e frame Ethernet.

Pacchetto ICMP

Di seguito viene presentato il codice sorgente che definisce la struttura *Icmp* in *pnet::packet::icmp::IcmpPacket*:

```
/// Represents a generic ICMP packet.
#[packet]
pub struct Icmp {
    #[construct_with(u8)]
    pub icmp_type: IcmpType,
    #[construct_with(u8)]
    pub icmp_code: IcmpCode,
    pub checksum: u16be,
    // theoretically, the header is 64 bytes long, but since the "Rest
Of Header" part depends on
    // the ICMP type and ICMP code, we consider it's part of the
payload.
    // rest_of_header: u32be,
    #[payload]
    pub payload: Vec<u8>,
}
```

Codice 19: Struttura pacchetto ICMP della crate pnet

Possiamo notare che in questo caso il payload è rappresentato usando un vettore di byte, invece di un tipo generico.
Sono stati definiti anche due tipi che rappresentano i campi tipo e codice dell'intestazione del pacchetto ICMP: *IcmpType* e *IcmpCode*. Questi tipi sono usati per avvolgere valori interi di 8 bit senza segno, come mostrato nel codice sorgente proveniente dallo stesso file:

```
/// Represents the "ICMP type" header field.
```

```rust
#[derive(Copy, Clone, Debug, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct IcmpType(pub u8);

impl IcmpType {
    /// Create a new `IcmpType` instance.
    pub fn new(val: u8) -> IcmpType {
        IcmpType(val)
    }
}

impl PrimitiveValues for IcmpType {
    type T = (u8,);
    fn to_primitive_values(&self) -> (u8,) {
        (self.0,)
    }
}

/// Represents the "ICMP code" header field.
#[derive(Copy, Clone, Debug, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct IcmpCode(pub u8);

impl IcmpCode {
    /// Create a new `IcmpCode` instance.
    pub fn new(val: u8) -> IcmpCode {
        IcmpCode(val)
    }
}
```

Codice 20: Tipi ausiliari per il pacchetto ICMP della crate pnet

Pacchetto IP

Di seguito viene presentato il codice sorgente che definisce la struttura *Ipv4* in *pnet::packe::ipv4::Ipv4Packet*:

```rust
/// Represents an IPv4 Packet.
#[packet]
pub struct Ipv4 {
    pub version: u4,
    pub header_length: u4,
    pub dscp: u6,
    pub ecn: u2,
    pub total_length: u16be,
    pub identification: u16be,
    pub flags: u3,
    pub fragment_offset: u13be,
    pub ttl: u8,
```

```
    #[construct_with(u8)]
    pub next_level_protocol: IpNextHeaderProtocol,
    pub checksum: u16be,
    #[construct_with(u8, u8, u8, u8)]
    pub source: Ipv4Addr,
    #[construct_with(u8, u8, u8, u8)]
    pub destination: Ipv4Addr,
    #[length_fn = "ipv4_options_length"]
    pub options: Vec<Ipv4Option>,
    #[length_fn = "ipv4_payload_length"]
    #[payload]
    pub payload: Vec<u8>,
}
```

Codice 21: Struttura pacchetto IP della crate pnet

Possiamo notare che in questo caso il payload è rappresentato usando un vettore di byte, invece di un tipo generico. Inoltre l'indirizzo della sorgente e della destinazione sono definiti come *Ipv4Addr*, che è la struttura presente nella crate **core::net** della libreria standard di Rust.

Pacchetto Ethernet

Di seguito viene presentato il codice che definisce la struttura *Ethernet* in *pnet::packet::ethernet::EthernetPacket*:

```
/// Represents an Ethernet packet.
#[packet]
pub struct Ethernet {
    #[construct_with(u8, u8, u8, u8, u8, u8)]
    pub destination: MacAddr,
    #[construct_with(u8, u8, u8, u8, u8, u8)]
    pub source: MacAddr,
    #[construct_with(u16)]
    pub ethertype: EtherType,
    #[payload]
    pub payload: Vec<u8>,
}
```

Codice 22: Struttura pacchetto Ethernet della crate pnet

Possiamo notare che in questo caso il payload è rappresentato usando un vettore di byte, invece di un tipo generico. Inoltre l'indirizzo della sorgente e della destinazione sono definiti come *MacAddr*, un tipo definito in *pnet::util::MacAddr*:

```
/// A MAC address.
#[derive(PartialEq, Eq, Clone, Copy, Default, Hash, Ord, PartialOrd)]
pub struct MacAddr(pub u8, pub u8, pub u8, pub u8, pub u8, pub u8);
```

Codice 23: Tipi ausiliari per il pacchetto Ethernet della crate pnet

Bibliografia

- ManualeRust, and Carol Nichols. “The Rust Programming Language - The Rust Programming Language.” *Learn Rust*, https://doc.rust-lang.org/book.
- PnetRust. “pnet - Rust.” *Docs.rs*, https://docs.rs/pnet/latest/pnet/.
- WikipediaC. “C (linguaggio di programmazione).” *Wikipedia*, https://it.wikipedia.org/wiki/C_(linguaggio_di_programmazione).
- WikipediaRust. “Rust (linguaggio di programmazione).” *Wikipedia*, https://it.wikipedia.org/wiki/Rust_(linguaggio_di_programmazione).

Appendice A - Programma completo in C

```
#include <arpa/inet.h>

unsigned char mymac[6] = { 0xf2,0x3c,0x91,0xdb,0xc2,0x98} ;
unsigned char myip[4] = { 88,80,187,84 };

struct icmp_packet {
unsigned char type;
unsigned char code;
unsigned short checksum;
unsigned short id;
unsigned short seq;
unsigned char data[1];
};

struct ip_datagram {
unsigned char ver_ihl;
unsigned char tos;
unsigned short len;
unsigned short id;
unsigned short flags_offs;
unsigned char ttl;
unsigned char proto;
unsigned short checksum;
unsigned int src;
unsigned int dst;
unsigned char payload[1];
};

struct ethernet_frame {
unsigned char dst[6];
unsigned char src[6];
unsigned short type;
unsigned char payload[1];
};

unsigned short checksum ( void * b, int s)
{
unsigned short * p;
int i;
unsigned int tot=0;
p = (unsigned short *) b;
for (i=0; i<s/2; i++){
tot +=ntohs(p[i]);
if ( tot & 0x10000) tot = (tot&0xFFFF) + 1;
}
if ( i*2 != s ){
tot+=ntohs(p[i])&0xFF00;
if ( tot & 0x10000) tot = (tot&0xFFFF) + 1;
}
return (0xFFFF-(unsigned short)tot);
}
```

```
int forge_icmp(struct icmp_packet * icmp){
int i;
icmp->type = 8;
icmp->code = 0;
icmp->checksum = 0;
icmp->id = 0x1234;
icmp->seq = 0;
for(i=0;i<32;i++) icmp->data[i]=i;
icmp->checksum = htons(checksum(icmp,40));
return 40;
}

void forge_ip(struct ip_datagram * ip, unsigned char* dst, unsigned short payloadlen,
unsigned char proto )
{
ip->ver_ihl = 0x45;
ip->tos = 0;
ip->len = htons(payloadlen+20);
ip->id = 0xABCD;
ip->flags_offs = 0;
ip->ttl = 128;
ip->proto = proto;
ip->checksum = 0;
ip->src = *(unsigned int *)myip;
ip->dst = *(unsigned int *)dst;
ip->checksum = htons(checksum(ip,20));
}

void forge_eth( struct ethernet_frame * eth, unsigned char *dst, unsigned short type )
{
for(int i=0;i<6;i++) eth->dst[i] = dst[i];
for(int i=0;i<6;i++) eth->src[i] = mymac[i];
eth->type = htons(type);
}

unsigned char buf[1500];

int main(){
int i,len;
unsigned char destip[4] = {147,162,2,100};
unsigned char destmac[6] = {0x00, 0x1A, 0x2B, 0x3C, 0x4D, 0x5E};
for(int j=0; j<1; j++) {
struct icmp_packet *icmp;
struct ip_datagram * ip;
struct ethernet_frame * eth;

eth = (struct ethernet_frame *) buf;
ip = (struct ip_datagram *) (eth->payload);
icmp = (struct icmp_packet *) (ip->payload);

len = forge_icmp(icmp);
forge_ip(ip,destip,len, 1);
forge_eth(eth,destmac,0x0800);
unsigned char eth_packet_dst[6];
for(i=0; i<6; i++) {
eth_packet_dst[i] = eth->dst[i];
```

```
}
unsigned char eth_packet_src[6];
for(i=0; i<6; i++) {
eth_packet_src[i] = eth->src[i];
}
unsigned short eth_packet_type = eth->type;
unsigned char ip_packet_ver_ihl = ip->ver_ihl;
unsigned char ip_packet_tos = ip->tos;
unsigned short ip_packet_len = ip->len;
unsigned short ip_packet_id = ip->id;
unsigned short ip_packet_flags_off = ip->flags_offs;
unsigned char ip_packet_ttl = ip->ttl;
unsigned char ip_packet_proto = ip->proto;
unsigned short ip_packet_checksum = ip->checksum;
unsigned int ip_packet_src = ip->src;
unsigned int ip_packet_dst = ip->dst;
unsigned char icmp_packet_type = icmp->type;
unsigned char icmp_packet_code = icmp->code;
unsigned short icmp_packet_checksum = icmp->checksum;
unsigned short icmp_packet_id = icmp->id;
unsigned short icmp_packet_seq = icmp->seq;
unsigned char icmp_packet_data[32];
for(i=0; i<32; i++) {
icmp_packet_data[i] = icmp->data[i];
}
}
}
```

Appendice B - Programma completo in Rust

```
use std::ptr;
use std::mem::size_of;
use std::arch::asm;

static MYMAC: [u8;6] = [0xF2, 0x3C, 0x91, 0xDB, 0xC2, 0x98];
static MYIP: [u8;4] = [88, 80, 187, 84];

#[repr(C, packed)]
struct IcmpPacket<T> {
r#type:u8,
code:u8,
checksum:u16,
id:u16,
seq:u16,
data: T,
}
#[repr(C, packed)]
struct IpDatagram<T> {
ver_ihl:u8,
tos:u8,
len:u16,
id:u16,
flags_offs:u16,
ttl:u8,
proto:u8,
checksum:u16,
src:u32,
dst:u32,
payload: T,
}

#[repr(C, packed)]
struct EthernetFrame<T> {
dst:[u8;6],
src:[u8;6],
r#type:u16,
payload: T,
}

fn forge_icmp(reference: & mut IcmpPacket<[u8;32]>) {
reference.r#type=8;
reference.code=0;
reference.checksum=0;
reference.id=0x1234;
reference.seq=0;
for i in 0..32 {
reference.data[i] = i as u8;
}
reference.checksum=checksum(ptr::addr_of!(reference.r#type), 40).to_be();
}

fn forge_ip<T>(reference: & mut IpDatagram<T>, dst: &[u8;4], payloadlen: u16, proto: u8)
```

```
{
reference.ver_ihl = 0x45;
reference.tos = 0;
reference.len = (payloadlen + 20).to_be();
reference.id = 0xABCD;
reference.flags_offs = 0;
reference.ttl = 128;
reference.proto = proto;
reference.checksum = 0;
reference.src = u32::from_le_bytes(MYIP);
reference.dst = u32::from_le_bytes(*dst);
reference.checksum=checksum(ptr::addr_of!(reference.ver_ihl), 20).to_be();
}

fn forge_eth<T>(reference: & mut EthernetFrame<T>, dst: &[u8;6], r#type: u16) {
reference.dst = *dst;
reference.src = MYMAC;
reference.r#type = r#type.to_be();
}


fn checksum(reference: *const u8, byte_size: u32) -> u16 {
let tmp: *const u16 = reference as *const u16;
let mut tot = 0u32;
let mut last_index: u32 = 0;
for i in 0..(byte_size/2) {
tot += unsafe { u32::from(tmp.wrapping_add(i as usize).read()).to_le()};
if(tot & 0x10000) != 0 { tot = (tot & 0xffff) + 1;}
last_index += 1;
}
if last_index*2 != byte_size {
tot += unsafe{ u32::from(tmp.wrapping_add(last_index as usize).read()).to_le() &
0xff00};
if(tot & 0x10000) == 1 { tot = (tot & 0xffff) + 1;}
}
return 0xffff - (tot as u16);
}

static mut BUFFER: [u8;1500] = [0;1500];

fn main() {
let dstip: [u8;4] = [147,162,2,100];
let dstmac: [u8;6] = [0x00, 0x1A, 0x2B, 0x3C, 0x4D, 0x5E];
let len = size_of::<EthernetFrame<IpDatagram<IcmpPacket<[u8;32]>>>>();
for i in 0..1 {
let ptr = unsafe {
BUFFER[0..len].align_to_mut::<EthernetFrame<IpDatagram<IcmpPacket<[u8;32]>>>>()};
let ethernet_frame = &mut ptr.1[0];
forge_eth::<IpDatagram<IcmpPacket<[u8;32]>>>(ethernet_frame, &dstmac, 0x0800);
forge_ip::<IcmpPacket<[u8;32]>>(&mut ethernet_frame.payload, &dstip, 40, 1);
forge_icmp(&mut ethernet_frame.payload.payload);

let eth_packet_dst = ethernet_frame.dst;
let eth_packt_src = ethernet_frame.src;
let eth_packet_type = ethernet_frame.r#type;
let ip_packet_ver_ihl = ethernet_frame.payload.ver_ihl;
```

```
let ip_packet_tos = ethernet_frame.payload.tos;
let ip_packet_len = ethernet_frame.payload.len;
let ip_packet_id = ethernet_frame.payload.id;
let ip_packet_flags_off = ethernet_frame.payload.flags_offs;
let ip_packet_ttl = ethernet_frame.payload.ttl;
let ip_packet_proto = ethernet_frame.payload.proto;
let ip_packet_checksum = ethernet_frame.payload.checksum;
let ip_packet_src = ethernet_frame.payload.src;
let ip_packet_dst = ethernet_frame.payload.dst;
let icmp_packet_type = ethernet_frame.payload.payload.r#type;
let icmp_packet_code = ethernet_frame.payload.payload.code;
let icmp_packet_checksum = ethernet_frame.payload.payload.checksum;
let icmp_packet_id = ethernet_frame.payload.payload.id;
let icmp_packet_seq = ethernet_frame.payload.payload.seq;
let icmp_packet_data = ethernet_frame.payload.payload.data;
}
}
```

Ringraziamenti

Mi è doveroso dedicare questo spazio della mia tesi a tutte le persone che mi hanno supportato nel mio percorso di crescita universitaria e professionale.
Per prima cosa, vorrei ringraziare il mio relatore Nicola Zingirian, per i suoi preziosi consigli e per la sua disponibilità. Grazie per avermi fornito spunti fondamentali nella stesura di questo lavoro e per avermi indirizzato nei momenti di indecisione.
Ringrazio infinitamente i miei genitori, che mi hanno sempre motivato a dare il meglio, e i miei più cari amici, che hanno condiviso con me gioie e dolori di questo percorso universitario.