



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Fuzzing Linux IPsec implementation with Syzkaller

Progetto di Software Security

Anno Accademico 2023/2024

Giuseppe Capasso
matr. M63001498

Indice

1	Fuzzing	3
1.1	Syzkaller	3
1.1.1	Syzlang	5
1.2	Esecuzione	6
1.2.1	Preparazione del kernel	6
2	IPsec	10
2.1	Implementazione IPsec in Linux	10
3	Ambiente di test	13
3.1	Approccio naive: interfacce TUN/TAP	13
3.2	Approccio avanzato: network namespaces e virtual ethernet	14
3.2.1	Configurazione	14
3.3	Configurazione kernel: socket NETLINK	15
4	Implementazione in Syzkaller	18
4.1	Fuzzing SA e SAD	18
4.2	Fuzzing header ESP e AH	19
4.3	Esperimento	20
	References	22

Introduzione

Negli ultimi anni, il kernel Linux ha visto un aumento significativo nell'adozione del fuzzing come strumento di sicurezza. Progetti come Syzkaller, sviluppato da Google, hanno dimostrato l'efficacia di questa tecnica nel rilevare numerose vulnerabilità prima che possano essere sfruttate da attori malintenzionati. Syzkaller utilizza tecniche avanzate per generare input complessi e coprire ampie porzioni del codice del kernel, identificando bug che potrebbero rimanere nascosti ai metodi di testing tradizionali.

Lo scopo del progetto è quello di esplorare l'attività di fuzzing per la ricerca di bug e vulnerabilità nel kernel Linux. Il kernel Linux è un'applicazione monolitica, concorrente e asincrona che prende direttamente il controllo dell'hardware il cui fallimento (detto ***kernel panic***) porta il sistema in *crash* permanente.

Il kernel Linux è composto da diversi sottosistemi ciascuno dei quali espone un'interfaccia verso l'utente che può essere utilizzata per il fuzzing congiuntamente allo sviluppo degli strumenti per l'analisi per l'individuazione di bug in memoria a *runtime* quali ***KASAN*** (utilizzato come *address sanitizer*), ***KMSAN*** (utilizzato come *memory sanitizer*), il più recente ***KCMSAN*** (utilizzato per individuare *race condition*) e all'introduzione dell'opzione ***KCOV*** per l'strumentazione del codice kernel.

1 Fuzzing

Il fuzzing è una tecnica utilizzata per effettuare testing di robustezza e sicurezza delle applicazioni basata sulla generazione automatica di input. I fuzzer possono essere classificati in:

- puramente randomici: producono per la maggior parte input non validi;
- generativi: si basano su modelli (come FSM) dei protocolli e cercano di navigarli per generare input validi;
- basati su mutazione: partono da *seed* validi e mutano il seed per generare altri input;

Il kernel Linux è un'applicazione monolitica, concorrente ed asincrona che espone un'interfaccia verso gli utenti sottoforma di system-call. Il fuzzing di quest'applicazione richiede l'utilizzo di strumenti per individuare bug nell'utilizzo della memoria. In particolare, sono stati abilitati:

- KASAN: strumento di debug per il kernel Linux progettato per rilevare errori di memoria come buffer overflow, use-after-free e altri tipi di violazioni di accesso alla memoria. Utilizza un meccanismo basato su shadow memory per monitorare le allocazioni e le deallocazioni di memoria, segnalandosi quando si verificano accessi illegali (Figura 1 e Figura 2). Inoltre, KASAN introduce il concetto di *quarantine* che viene utilizzato per evitare gli errori *use-after-free* nello *heap* che altrimenti sarebbe gestito con una politica LIFO dal kernel (Figura 3 e Figura 4);
- KMSAN: strumento di debug utilizzato per individuare valori di memoria non inizializzati. Analogamente, al caso KASAN, utilizza gli shadow byte per tenere traccia della memoria allocata;

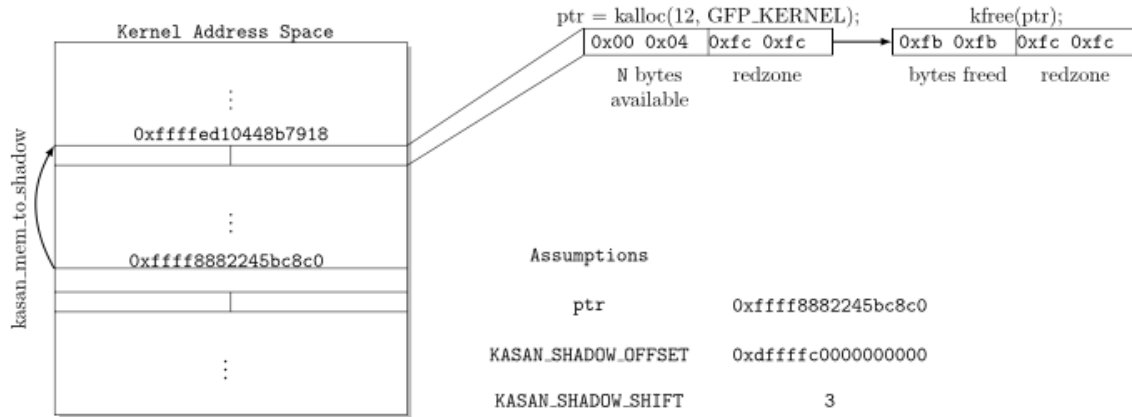


Figure 1: Allocazione di un range di indirizzi nel kernel

1.1 Syzkaller

Syzkaller(google, 2024) è un'applicazione che effettua *generative fuzzing* di applicazioni kernel in ambiente virtualizzato il cui "seed" è un programma che consiste in una sequenza di system-call. La tecnica utilizzata è detta *coverage-based*: decide che un programma ha un'altra probabilità di sopravvivere nella generazione di input in base alla copertura che offre.

Con riferimento al kernel Linux, l'input è costituito da system-call descritte staticamente attraverso *syzlang* (un linguaggio dichiarativo) in semplici file di testo. Una sequenza di system-call forma un programma che può essere eseguito dal kernel e viene utilizzato da syzkaller per mutare e generare altri programmi. Ad esempio, un semplice programma che apre un file e legge alcuni byte da esso può essere scritto nel seguente modo (google, 2024)[docs/syscall_descriptions.md]:

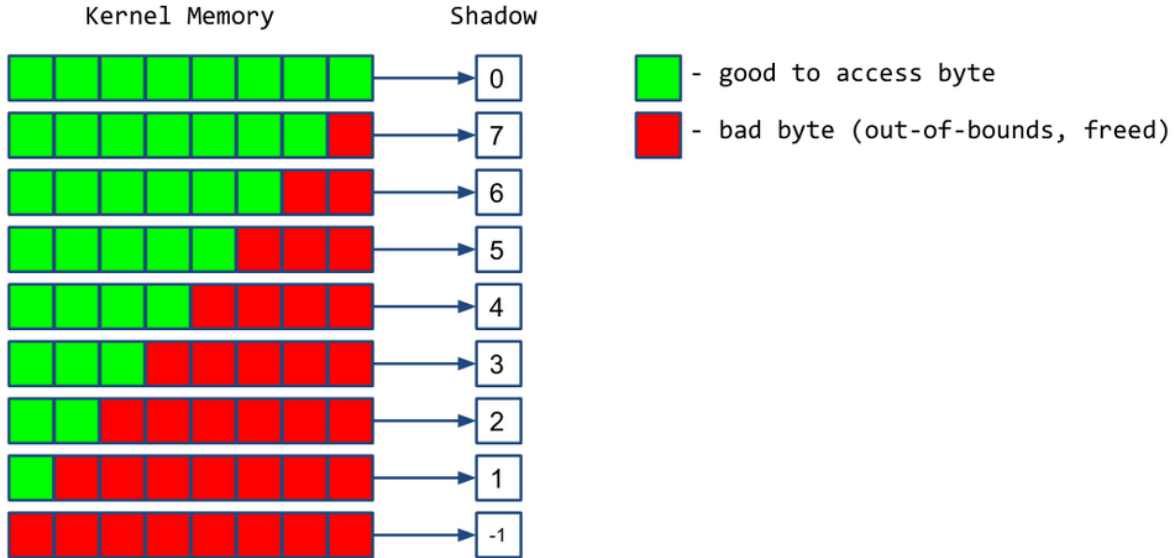


Figure 2: Shadow byte per controllare le allocazioni

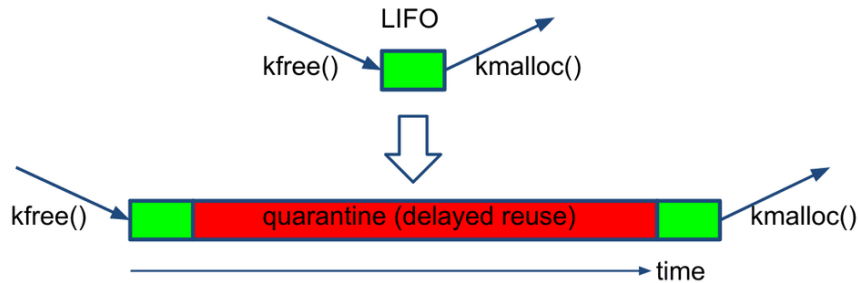


Figure 3: Kasan quarantena per gli oggetti dell'heap

```
open(file filename, flags flags[open_flags], mode flags[open_mode]) fd
read(fd fd, buf buffer[out], count len[buf])
close(fd fd)
open_mode = S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH
```

Syzkaller ha la struttura modulare presentata in Figura 5.

Syz-manager Syz-manager è il componente centrale di Syzkaller, responsabile della gestione e del coordinamento dell'intero processo di fuzzing. Gestisce le configurazioni dei test, avvia e controlla le istanze di Syz-fuzzer e monitora i risultati del fuzzing. Syz-manager raccoglie le statistiche, analizza i crash e mantiene un registro delle esecuzioni, facilitando il monitoraggio e l'analisi dei risultati. In particolare, si connette alla macchina target attraverso SSH, comunica con *syz-fuzzer*. Inoltre, syz-manager offre una dashboard di monitoraggio in cui si può verificare lo stato dell'esecuzione e analizzare eventuali crash riportati.

Syz-fuzzer Si occupa di generare input a partire dai programmi (collezionati in un corpus) proveniente dal manager. Inoltre, si occupa di monitorare lo stato del kernel cercando dei crash che si possono verificare per:

- nessun output dalla macchina target;

- connessione alla macchina virtuale persa;
- macchina virtuale target non esegue nessun programma per molto tempo;

Infine, il fuzzer raccoglie metriche di coverage (grazie alla funzionalità KCOV) e le riporta attraverso un'API RPC al manager che fornisce un nuovo input.

Syz-executor Si occupa di eseguire una sequenza di system-call che riceve in ingresso dal fuzzer e comunica con il fuzzer attraverso una memoria condivisa.

1.1.1 Syzlang

Il linguaggio permette di definire *risorse* che possono essere utilizzate come tipo di variabile e possono guidare il fuzzer durante la generazione di parametri di input. Ad esempio, la risorsa *filename* indica al fuzzer di generare solo percorsi e nome di file validi in Linux.

Alla descrizione formale delle system-call viene poi associato un programma che syzkaller può mostrare in:

- binaria: caricato ed eseguito direttamente dal kernel;
- testuale: utilizzato per analizzare visivamente l'output del fuzzer;
- in codice C: syzkaller è in grado di generare un programma C da una rappresentazione binaria per essere d'aiuto agli sviluppatori nella riproduzione di bug e vulnerabilità trovate;

Ad esempio, il programma descritto in precedenza può portare alla generazione del codice seguente con parametri reali:

```
r0 = open(&(0x7f0000000000)="/file0", 0x3, 0x9)
read(r0, &(0x7f0000000000), 42)
close(r0)
```

Inoltre, il linguaggio permette di definire costanti, campi di lunghezza variabile e *unions*. Ad esempio, il seguente codice definisce il tipo *port_sock* come un intero 16-bit che può andare da 20000 a 20004:

```
type sock_port int16be[20000:20004]
```

Mentre, la seguente *union* rappresenta alcuni indirizzi IPv4:

```
ipv4_addr [
# Random public addresses 100.1.1.[0-2]:
  rand_addr int32be[0x64010100:0x64010102]
# 0.0.0.0:
  empty const[0x0, int32be]
# 127.0.0.1:
  loopback const[0x7f000001, int32be]
] [size[4]]
```

Inoltre, in seguito viene dichiarato il *file descriptor* di una *socket* ereditando dalla risorsa *fd* (*Syzkaller networking*, n.d.):

```
resource sock[fd]
resource sock_in[sock]
resource sock_tcp[sock_in]
```

Infine, le system-call descritte possono avere diverse varianti in base al tipo di configurazione ricevuta. Ad esempio, una socket UDP è diversa da una socket TCP, ma vengono generate a partire dalla stessa system-call. In syzlang, è possibile utilizzare il simbolo "\$" per indicare una versione particolare di una funzione (*Syzkaller networking*, n.d.):

```
socket$inet_tcp(domain const[AF_INET], type const[SOCK_STREAM],
                proto const[0]) sock_tcp
socket$inet_udp(domain const[AF_INET], type const[SOCK_DGRAM],
                proto const[0]) sock_udp
```

Pseudo-system call Syzlang consente di descrivere funzioni che non hanno una diretta implementazione nel kernel, ma può essere fornita dall'utente. Queste pseudo-syscall servono a configurare o a racchiudere in una sola funzione più system call. Ad esempio, per arrivare a testare il codice di rete è necessario fornire al fuzzer la nozione di connessione TCP che viene implementata attraverso la pseudo-syscall *syz_emit_ethernet*.

```
syz_emit_ethernet(len len[packet],
                  packet ptr[in, eth_packet],
                  frags ptr[in, vnet_fragmentation, opt])
```

1.2 Esecuzione

Una volta compilato syzkaller, è necessario:

- compilare un kernel abilitando gli strumenti di coverage, debug e i sottosistemi che si vogliono testare;
- creare un'immagine "bootable" con programmi base della suite GNU;

1.2.1 Preparazione del kernel

I flag principali per instrumentare il codice del kernel sono i seguenti:

```
# Coverage collection.
CONFIG_KCOV=y

# Debug info for symbolization.
CONFIG_DEBUG_INFO=y

# Memory bug detector
CONFIG_KASAN=y
CONFIG_KASAN_INLINE=y

# Required for Debian Stretch
CONFIG_CONFIGFS_FS=y
CONFIG_SECURITYFS=y
```

Una volta compilato il kernel, bisogna configurare l'immagine. In particolare, syzkaller fornisce uno script di configurazione in cui crea un'immagine debian a partire da un *rootfs*. In questa immagine sarà configurato opportunamente un server SSH e generata una coppia di chiavi per fornire l'accesso a syzkaller-manager. Una volta preparata l'immagine può essere effettuato il test con QEMU eseguendo il seguente comando:

```
qemu-system-x86_64 \
-m 2G \
-smp 2 \
-kernel $KERNEL/arch/x86/boot/bzImage \
-append "console=ttyS0 root=/dev/sda earlyprintk=serial net.ifnames=0" \
-drive file=$IMAGE/buster.img,format=raw \
-net user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:10021-:22 \
-net nic,model=e1000 \
-enable-kvm \
```

```
-nographic \  
-pidfile vm.pid \  
2>&1 | tee vm.log
```

Infine, è possibile eseguire syzkaller con il seguente file di configurazione in cui si eseguono 8 kernel in parallelo ciascuno con una cpu e in 1GB di memoria e monitorare il processo nella dashboard web Figura 6.

```
{  
  "target": "linux/amd64",  
  "http": "127.0.0.1:56741",  
  "workdir": "syzkaller/workdir",  
  "kernel_obj": "$KERNEL",  
  "image": "$IMAGE/buster.img",  
  "sshkey": "$IMAGE/buster.id_rsa",  
  "syzkaller": "syzkaller/",  
  "procs": 8,  
  "type": "qemu",  
  "vm": {  
    "count": 8,  
    "kernel": "$KERNEL/arch/x86/boot/bzImage",  
    "cpu": 1,  
    "mem": 1024  
  }  
}
```

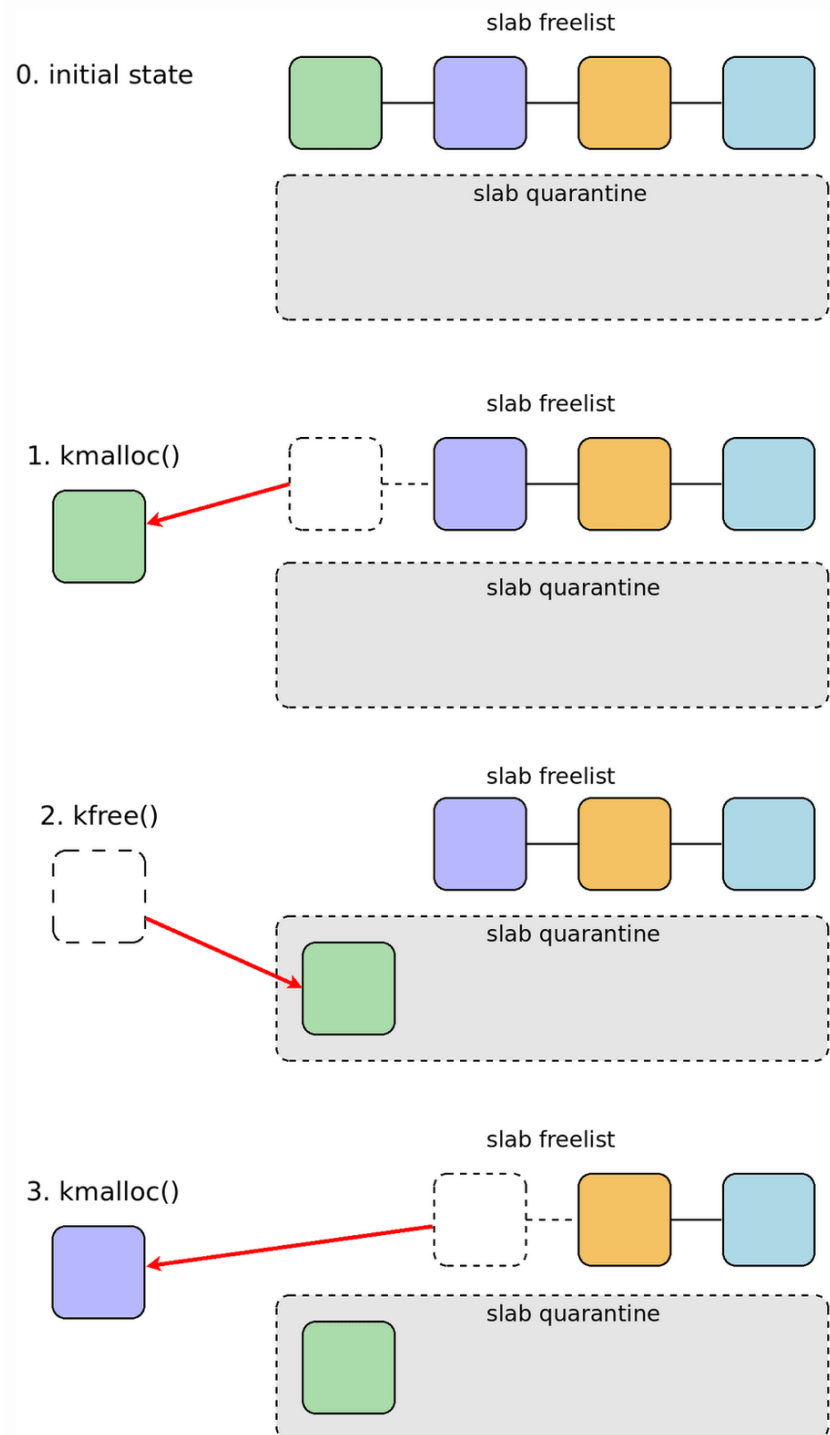



Figure 4: Use-after-free con KASAN

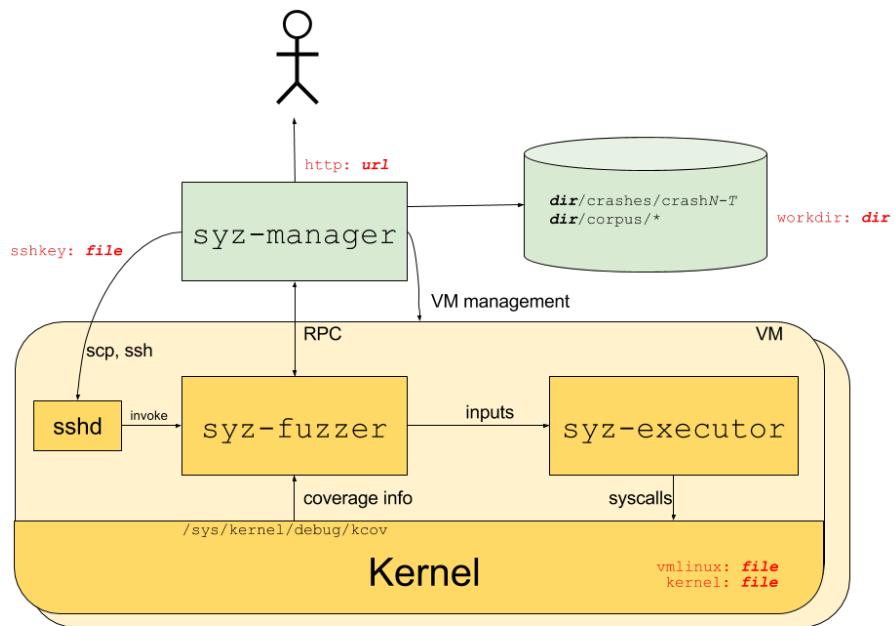


Figure 5: Architettura Syzkaller



Figure 6: Esecuzione syzkaller

2 IPsec

IPsec (Internet Protocol Security) è una suite di protocolli per la protezione delle comunicazioni IP attraverso l'autenticazione e la crittografia di ogni pacchetto IP in una sessione. La suite contiene due protocolli principali:

- Encapsulating Security Payload: è un protocollo che fa parte del framework IPsec che incapsula il pacchetto originale e lo cripta. Consente di implementare la riservatezza dei dati, l'autenticazione della sorgente che la verifica dell'integrità
- Authentication Header: header che fornisce l'integrità dei dati e autenticazione dell'origine, ma non offre funzioni di cifratura;

In particolare, IPsec ha due modalità di funzionamento:

- Transport mode: protegge solo il payload del pacchetto IP lasciando intatti gli header originali;
- Tunnel mode: protegge l'intero pacchetto IP, incapsulandolo in un nuovo pacchetto IP con nuovi header;

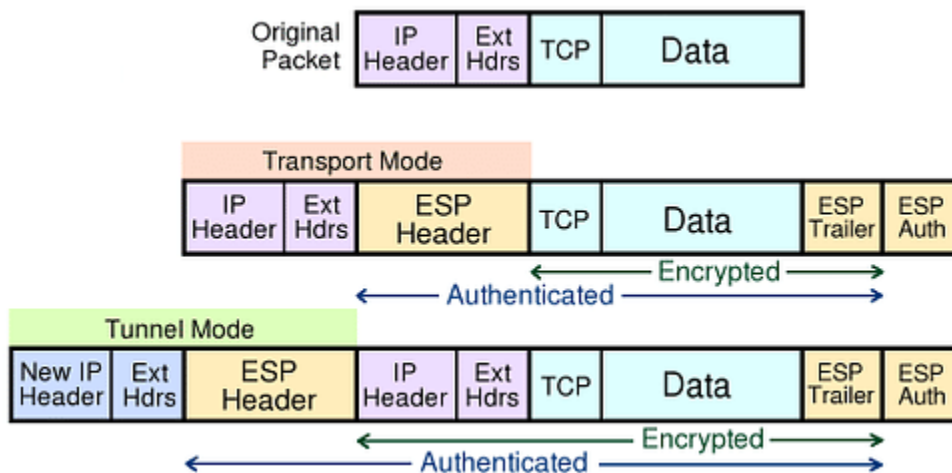


Figure 7: Struttura di un pacchetto ESP in tunnel e in transport mode

La componente principale di IPsec è la **Security Association (SA)**: una connessione logica unidirezionale tra due host caratterizzata da:

- SPI: stringa che identifica la security association locale;
- Indirizzo IP di destinazione: indirizzo della destinazione della SA;
- SPI: identifica se l'associazione SA è un Authentication Header o un Encapsulation

Le SA sono conservate all'interno del **Security Association Database (SAD)**.

Infine, un componente fondamentale per IPsec è la gestione e lo scambio delle chiavi che avviene attraverso il protocollo **Internet Key Exchange**.

2.1 Implementazione IPsec in Linux

Linux implementa in kernel space le strutture dati necessarie ad IPsec. In particolare, in questo progetto, ci si soffermerà sul protocollo ESP in transport mode. In questo modo, possono essere testati i componenti mostrati in Figura 8

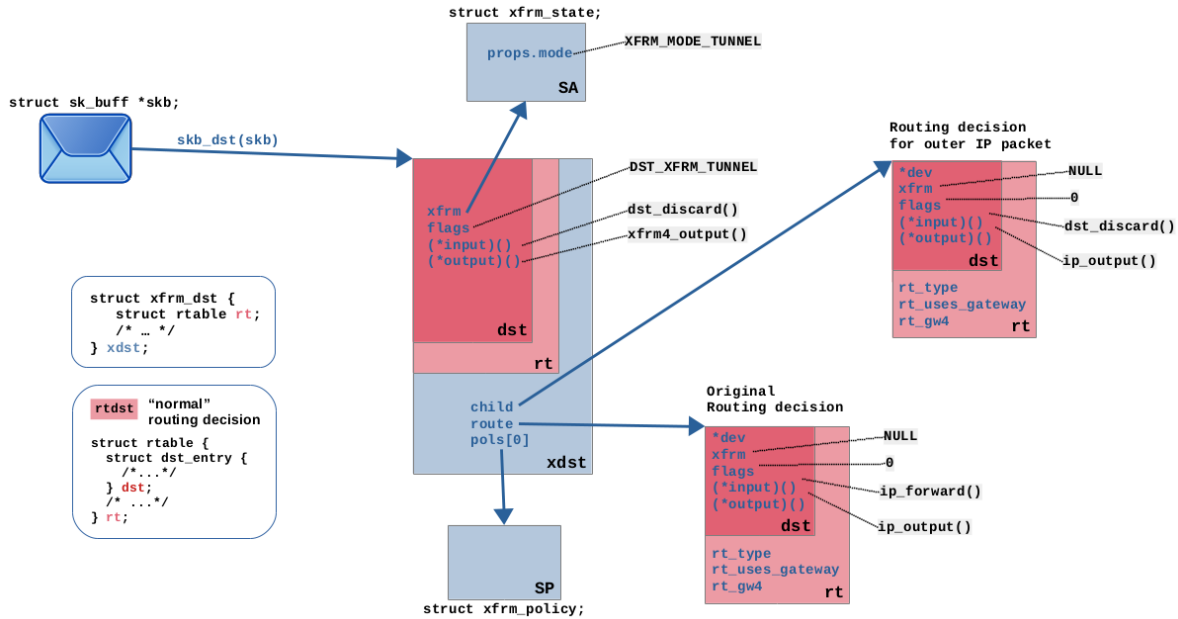


Figure 8: Architettura IPsec in Linux

Dallo user space è possibile creare security association e manipolare i pacchetti IP in un particolare indirizzo con l'utility *ip xfrm*. In particolare, dati due host tra i quali si vuole definire un canale sicuro, definite le seguenti variabili:

```

$ host1="IP_HOST_1"
$ host2="IP_HOST_2"
$ key12=0x$(xxd -c 32 -l 32 -ps /dev/random)
$ key21=0x$(xxd -c 32 -l 32 -ps /dev/random)
$ spi12=0x$(xxd -c 4 -l 4 -ps /dev/random)
$ spi21=0x$(xxd -c 4 -l 4 -ps /dev/random)

```

È possibile creare una SA su entrambi gli host specificando modalità transport con protocollo ESP utilizzando AES come algoritmo di crittografia con i comandi:

```

$ ip xfrm state add src $host1 dst $host2 \
  proto esp spi $spi12 enc 'cbc(aes)' \
  $key12 mode transport

$ ip xfrm state add src $host2 dst $host1 \
  proto esp spi $spi21 enc 'cbc(aes)' \
  $key21 mode transport

```

Essendo le SA unidirezionali, esse vanno create in tutti i versi in cui si vuole che la comunicazione avvenga. Successivamente, bisogna definire le security policy in cui si indica quali sono gli estremi della comunicazione modificando le strutture della tabella netfilter (Figura 9) Ad esempio, sull'host1 bisogna configurare come in uscita il traffico in direzione host2 e viceversa (bisogna fare l'operazione duale sull'host2).

```

$ ip xfrm policy add dir out \
  src $host1 dst $host2 \
  tmpl proto esp mode transport

```

```
$ ip xfrm policy add dir in \
src $host2 dst $host1 \
tmpl proto esp mode transport
```

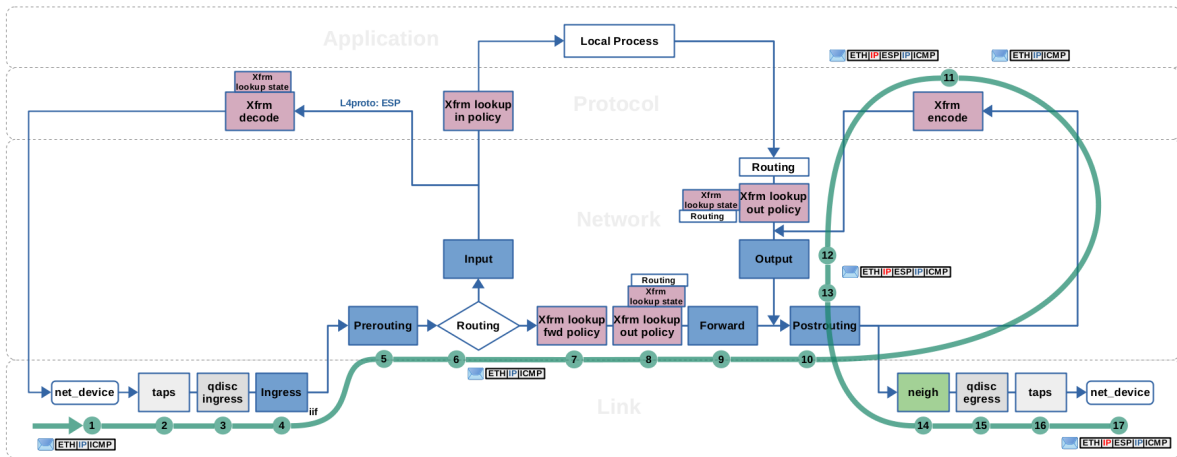


Figure 9: Flusso completo seguito da un pacchetto IPsec all'interno del kernel Linux

3 Ambiente di test

La difficoltà principale per il testing di IPsec è la creazione di un tunnel locale alla macchina virtuale. Il requisito principale è quello di generare traffico cifrato andando a coprire l'implementazione del protocollo IPsec all'interno del kernel. La figura in Figura 10 mostra la soluzione progettata.

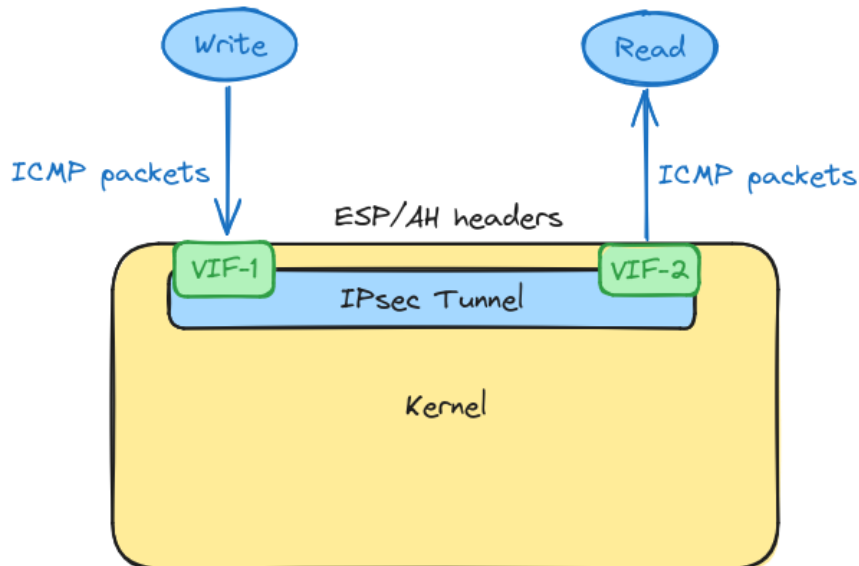


Figure 10: IPsec tunnel con interfacce virtuali

Infatti, a differenza di quanto mostrato in (*Syzkaller networking*, n.d.), in questo caso è necessario creare una comunicazione quanto più vicina alla realtà e utilizzando le interfacce TUN/TAP non è facile metterle in comunicazione.

3.1 Approccio naive: interfacce TUN/TAP

Il primo approccio tentato è quello della configurazione di una interfaccia TUN/TAP (?). Questo tipo di interfaccia è già supportato in Syzkaller ed è di facile implementazione poichè consente di riutilizzare le procedure già definite per inviare pacchetti sull'interfaccia.

La creazione della configurazione è mostrata in Figura 11 e ripercorre quella già descritta in (*Syzkaller networking*, n.d.) aggiungendo come passaggio intermedio quello della configurazione delle SA sull'interfaccia. In verde sono mostrate le funzionalità già implementate in (*Syzkaller networking*, n.d.)

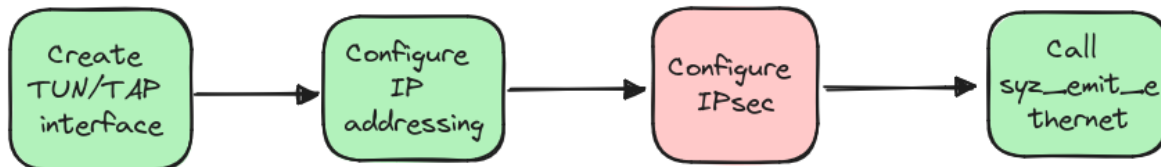


Figure 11: Flusso configurazione IPsec su interfaccia TAP

3.2 Approccio avanzato: network namespaces e virtual ethernet

Una soluzione più completa prevede l'utilizzo del concetto di namespace all'interno del kernel Linux. In questo caso, sono stati utilizzati i **network namespaces** per creare ambienti di rete completamente isolati a cui è possibile aggiungere delle interfacce. Per la configurazione di test, è stato utilizzato il driver **veth** del kernel linux che crea una coppia di interfacce virtuali ciascuna delle quali viene spostata nel suo namespace.

Come mostrato in Figura 12, la configurazione implementata utilizza i network namespace per duplicare lo stack di rete implementato nel kernel. Quest'approccio può essere vantaggioso per il fuzzing in quanto

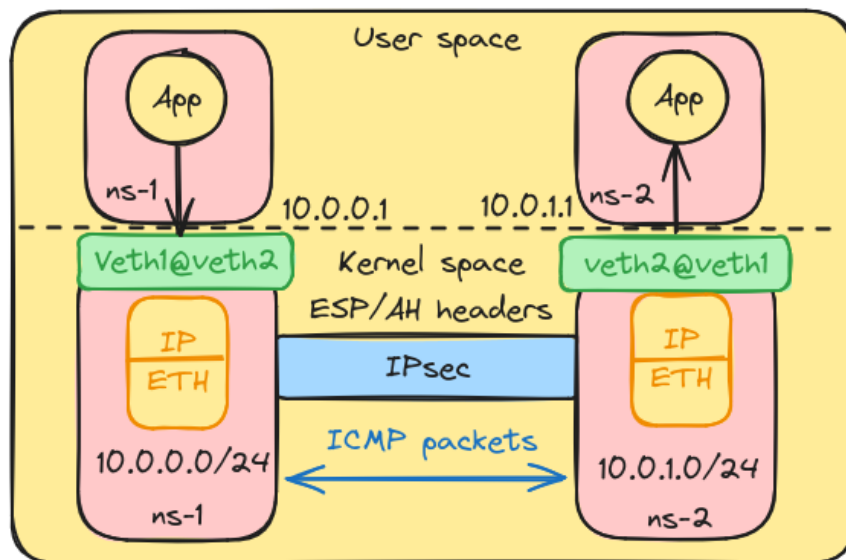


Figure 12: Configurazione con network namespaces

viene testata un'interfaccia più ampia che include:

- le system call per la creazione di container: la creazione di un namespace Linux include l'utilizzo delle system call di **unshare**, **setns** e **clone**;
- implementazione del driver virtual ethernet;
- implementazione del sistema **xfrm** per la manipolazione di pacchetti IP;

Inoltre, configurando le interfacce per utilizzare IPsec possono essere utilizzate tutte le altre funzionalità già implementate in Syzkaller per il fuzzing delle funzionalità di rete (creazione socket TCP e UDP) senza implementare manualmente gli header ESP e AH.

3.2.1 Configurazione

Utilizzando una macchina di test, la configurazione è stata prodotta utilizzando semplici comandi da terminale che poi saranno tradotti in messaggi NETLINK, come spiegato di seguito e saranno utilizzate come pseudo-systemcall. Per creare un **network namespace** si usa il comando:

```
$ ip netns add <ns-id>
```

Come qualsiasi namespace di Linux, supportano l'esecuzione di comandi al suo interno. Infatti, appena creato, il namespace contiene solo un'interfaccia di loopback (Figura 13).

Successivamente si crea la coppia di interfacce virtuali, configura con gli indirizzi IP mostrati in Figura 12, vengono aggiunte ai namespace con il comando:

```

debian@debian:~$ sudo ip netns exec ns1 ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
debian@debian:~$

```

Figure 13: Interfacce presenti in un netns nuovo

```

$ ip link add <veth-name-1> type veth peer name <veth-name-2>
$ ip link set <ifname> netns <netns-id>

```

Quindi applicando anche la configurazione con *xfrm*, si riesce ad avere un tunnel IPsec configurato come mostrato in Figura 14.

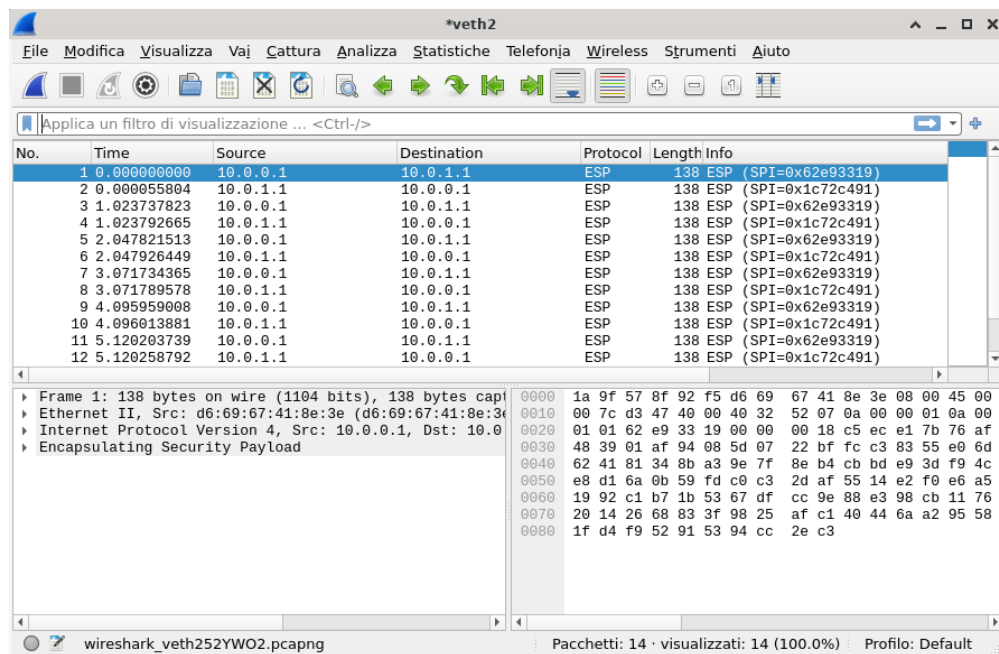


Figure 14: Realizzazione del tunnel IPsec tra network namespace separati

3.3 Configurazione kernel: socket NETLINK

Le socket NETLINK sono un meccanismo di comunicazione inter-processo (IPC) in Linux che permettono la comunicazione tra il kernel e gli utenti in spazio user. NETLINK viene utilizzato per vari scopi, inclusi la configurazione della rete e la gestione delle politiche di sicurezza. All'interno del codice aggiunto a Syzkaller, le socket netlink sono utilizzate per configurare le interfacce di rete. Ad esempio, per aggiungere un'interfaccia ad un network namespace si può usare la funzione:

```

1 // ip link set <ifname> netns <ns-id>
2 void move_if_to_pid_netns(char *ifname, int netns) {
3     // gestione dell'errore omessa per brevità
4     int sock_fd = socket(PF_NETLINK, SOCK_RAW | SOCK_CLOEXEC, NETLINK_ROUTE);
5
6     struct nl_req req = {

```



```

7         .n.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifinfomsg)),
8         .n.nlmsg_flags = NLMFREQUEST | NLMFACK,
9         .n.nlmsg_type = RTMNEWLINK,
10        .i.ifi_family = PF_NETLINK,
11    };
12
13    // funzioni di utility per aggiungere parametri alla richiesta
14    addattr_l(&req.n, sizeof(req), IFLA_NET_NS_FD, &netns, 4);
15    addattr_l(&req.n, sizeof(req), IFLA_IFNAME,
16             ifname, strlen(ifname) + 1);
17
18    // invio del messaggio
19    send_nlmsg(sock_fd, &req.n);
20
21    close(sock_fd);
22 }

```

Listing 1: Migrazione di un'interfaccia di rete in un network namespace con messaggi Netlink in C

Mentre invece per configurare una SA è stata usata una funzione simile alla seguente:

```

1 void create_sa(int sock) {
2     char buffer[NL_BUFSIZE];
3     struct nlmsghdr *nlmsg = (struct nlmsghdr *)buffer;
4     struct xfrm_usersa_info *sa_info;
5     struct rtattr *rta;
6     int key_len = 16;
7     unsigned char key[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
8                             0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};
9
10    memset(buffer, 0, NL_BUFSIZE);
11
12    // Setup NLMSG header
13    nlmsg->nlmsg_len = NLMSG_LENGTH(sizeof(struct xfrm_usersa_info));
14    nlmsg->nlmsg_type = XFRMLMSG_NEWSA;
15    nlmsg->nlmsg_flags = NLMFREQUEST | NLMF_CREATE;
16    nlmsg->nlmsg_seq = 1;
17
18    // Setup SA info
19    sa_info = NLMSG_DATA(nlmsg);
20    sa_info->sel.family = AF_INET;
21    inet_pton(AF_INET, "192.168.1.1", &sa_info->saddr);
22    inet_pton(AF_INET, "192.168.1.2", &sa_info->id.daddr);
23    sa_info->id.proto = IPPROTO_ESP;
24    sa_info->id.spi = htonl(0x100);
25    sa_info->lft.hard_byte_limit = XFRM_INF;
26    sa_info->lft.hard_packet_limit = XFRM_INF;
27    sa_info->lft.soft_byte_limit = XFRM_INF;
28    sa_info->lft.soft_packet_limit = XFRM_INF;
29    sa_info->mode = XFRM_MODE_TRANSPORT;
30    sa_info->reqid = 0;
31
32    // Add encryption key attribute
33    rta = (struct rtattr *)((char *)nlmsg + NLMSG_ALIGN(nlmsg->nlmsg_len));
34    rta->rta_type = XFRMA_ALG_CRYPT;
35    rta->rta_len = RTA_LENGTH(sizeof(struct xfrm_algo) + key_len);

```

```

36
37     struct xfrm_algo *alg = (struct xfrm_algo *)RTADATA(rta);
38     strcpy(alg->alg_name, "cbc(aes)");
39     alg->alg_key_len = key_len * 8;
40     memcpy(alg->alg_key, key, key_len);
41
42     nlmsg->nlmsg_len = NLMSG_ALIGN(nlmsg->nlmsg_len) +
43         RTALENGTH(sizeof(struct xfrm_algo) + key_len);
44
45     // Send Netlink message
46     if (send_netlink_msg(sock, nlmsg) < 0) {
47         perror("send_netlink_msg");
48     }

```

Listing 2: Creazione di una SA con socket Netlink in C

4 Implementazione in Syzkaller

L'implementazione del fuzzing per IPsec prevede due fasi distinte derivanti dal fatto che il protocollo non include lo scambio delle chiavi e quindi si compone di due fasi distinte. Ciò può essere sfruttato per testare da due punti di vista differenti l'implementazione:

- configurazione SA: utilizza lo strumento XFRM per creare delle entry all'interno del SAD. Nella realtà, lo scambio e la gestione delle chiavi avviene con IKE e Diffie-Helman;
- generazione del traffico: invio di pacchetti con header IPsec;

4.1 Fuzzing SA e SAD

Per il fuzzing del sistema di gestione delle chiavi sono state predisposte 2 pseudo systemcall:

- ***syz_emit_ipsec()***: aggiunge delle SA per la comunicazione tra due IP prima di inviare il pacchetto;
- ***syz_flush_ipsec()***: elimina lo stato delle SA;
- ***syz_send_ipsec()***: encapsula un pacchetto IP in un header ESP;

Per forzare la creazione di stati inconsistenti, la funzione flush è stata implementata come mostrato di seguito.

```
1 static long syz_flush_ipsec(volatile long a0) {
2     int n = (int)a0;
3     if (n < 0) {
4         flush_ipsec_policy();
5     } else if (n == 0) {
6         flush_ipsec_policy();
7         flush_ipsec_state();
8     } else {
9         flush_ipsec_state();
10    }
11    return 0;
12 }
```

Listing 3: Pseudo systemcall per la rimozione di SA dal SAD

Mentre la funzione ***syz_emit_ipsec()***, estrae gli indirizzi IP dalla frame e crea una SA per la coppia. In versione semplificata, la funzione è mostrata di seguito.

```
1 static long syz_emit_ipsec(volatile long a0, volatile long a1, volatile long
    a2) {
2     // controllo degli errori omessi
3     struct ethhdr* eth = (struct ethhdr*)data;
4     struct iphdr* ip = (struct iphdr*)(data + sizeof(struct ethhdr));
5
6     // estrazione IP
7     inet_ntop(AF_INET, &ip->saddr, src_ip, INET_ADDRSTRLEN);
8     inet_ntop(AF_INET, &ip->daddr, dst_ip, INET_ADDRSTRLEN);
9
10    add_ipsec_state(sock, src_ip, dst_ip, key, spi);
11    add_ipsec_policy_out(sock, src_ip, dst_ip);
12    add_ipsec_policy_in(sock, dst_ip, src_ip);
13
14    return write(ipsec_tun, data, length);
15 }
```

Listing 4: Creazione di una SA per una frame Ethernet

Infine, è stata aggiunta una pseudo-systemcall per generare header ESP e incapsulare pacchetti.

```

1 static long sys_send_ipsec(volatile long a0, volatile long a1, volatile long
    a2) {
2     int sock = (int)a0;
3     int length = (int)a2;
4     uint8_t* data = (uint8_t*)a0;
5     uint8_t* esp = 0;
6     size_t* len = 0;
7     int err;
8
9     err = encapsulate_ip_in_esp(data, length, esp, len);
10
11     if (err < 0)
12         return err;
13
14     return write(sock, esp, *len);
15 }

```

Listing 5: Incapsulamento di un pacchetto proveniente da una socket

4.2 Fuzzing header ESP e AH

Per testare l'handling di questi pacchetti è stata aggiunta la definizione di queste intestazioni per il protocollo IPsec nel file `vnet.txt`. Ad esempio, per esp:

```

esp_header {
    spi int32be[0x1234:0x12345]
    seq const[0x1, int32be]
    icv int32[0x0:0xffffffff]
} [packed]

esp_trailer {
    pad_length int8[0x0:0xff]
    next_header const[ipv4_types, int8]
} [packed]

esp_packet {
    esp_hdr esp_header
    data ipv4_packet_t[flags[ipv4_types, int8], array[int8]]
    esp_tr esp_trailer
} [packed]

```

In questa definizione si definisce l'incapsulamento di un pacchetto IP con quanto viene specificato nella specifica di ESP. Infatti, *esp_packet* viene poi riportato all'interno dei pacchetti *ipv4* definiti di seguito:

```

ipv4_packet [
    # ... altri tipi di pacchetti
    esp ipv4_packet_t[const[IPPROTO_ESP, int8], esp_packet]
    # ... altri tipi di pacchetti
] [varlen]

```

Allo stesso modo, è stato aggiunto il supporto per *Authentication Header* sfruttando la possibilità di *syzlang* di aggiungere campi con lunghezza calcolata a runtime. Infatti, il campo *payload_len* viene calcolato con la *macro* che si riferisce alla lunghezza del pacchetto.

```

ah_header {
  next_header const[ipv4_types, int8]
  payload_len len[ah_header, int8]
  reserved const[0x0, int16be]
  spi int32be[0x1234:0x12345]
  seq const[0x1, int32be]
  icv int32[0x0:0xffffffff]
} [packed]

ah_packet {
  header ah_header
  payload ipv4_packet_t[flags[ipv4_types, int8], array[int8]]
} [packed]

```

Infine, è stato aggiunto al tipologia di pacchetti supportati:

```

ipv4_packet [
  # altri pacchetti
  ah ipv4_packet_t[const[IPPROTO_AH, int8], ah_packet]
] [varlen]

```

4.3 Esperimento

Per cercare di focalizzare il testing sul sottosistema di rete, è stato avviato il fuzzing solo con le system call che possono impattare sulle strutture mostrate in Figura 8 e Figura 9 includendo la seguente lista nella configurazione:

```

// elenco abbreviato
"enable_syscalls": [
  "syz_emit_ethernet",
  "syz_emit_ipsec",
  "syz_send_ipsec",
  "syz_flush_ipsec",
  "syz_extract_tcp_res",
  "syz_extract_tcp_res$synack",
  "sendmsg$nl_xfrm",
  "socket$nl_xfrm",
  "socket$nl_netfilter",
  "socket$inet_icmp",
  "sendto",
  "recvfrom",
  "sendmsg",
  "recvmsg",
  "socket$igmp",
  "listen",
  "shutdown"
]


```

In Figura 15, è mostrato un insieme di programmi generato dal sistema ordinati per coverage.

In Figura 16, è riportata la dashboard di ricapitolazione del fuzzing.

Coverage	Program
2710	syz_emit_ipsec
2421	syz_emit_ipsec
2375	syz_emit_ipsec
2316	syz_emit_ipsec
2236	syz_emit_ipsec
2143	syz_emit_ipsec
2135	syz_extract_tcp_res-syz_emit_ipsec
2074	syz_init_net_socket\$nl_generic
2000	syz_emit_ipsec
1936	syz_emit_ipsec
1873	syz_emit_ipsec

Figure 15: Programmi generati durante l'esecuzione

[Stats](#) 

candidates	0
corpus	259
coverage	6776
exec total	17453 (50/sec)
fuzzing VMs	6
reproducing	0
crash types	0
crashes	0
suppressed	0
syscalls	85
uptime	267 sec
VM	235 MB
buffer too small	0
exec candidate	4 (41/hour)
exec collide	285 (49/min)
exec fuzz	664 (114/min)
exec gen	88 (15/min)
exec hints	207 (35/min)
exec minimize	12185 (35/sec)
exec retries	0 (0/hour)
exec seeds	493 (85/min)
exec smash	449 (77/min)
exec triage	3033 (522/min)
executor restarts	56 (579/hour)
filtered coverage	0
fuzzer jobs	653
fuzzing	1529 sec
heap	148 MB
hints jobs	296
instance restart	40 sec
max signal	9946
new inputs	1000
no exec duration	11937116240 (34302058/sec)
no exec requests	105 (18/min)
prog exec time	150
rpc recv	102 MB (299 kb/sec)
rpc sent	63 MB (185 kb/sec)
signal	7589
smash jobs	318
triage jobs	39
vm output	0 MB (0 kb/sec)
vm restarts	6 (62/hour)

Figure 16: Esecuzione del fuzzing per il sottosistema di rete

References

- google. (2024). *Syzkaller*. <https://github.com/google/syzkaller>. GitHub.
- Syzkaller networking*. (n.d.). <https://xairy.io/articles/syzkaller-external-network#-about-syzkaller>. (Accessed: 2024-06-10)