Implementation and Modification for CPE Routers: Filter Rule Optimization, IPsec Interface and Ethernet Switch

Masanobu SAITOH(msaitoh@netbsd.org)[*] Hiroki SUENAGA(hsuenaga@iij.ad.jp)[†]

March 2014

Abstract

Internet Initiative Japan Inc. (IIJ) has developed its own Customer Premises Equipment (CPE), called *SEIL* , for 15 years. The firmware of *SEIL* is based on NetBSD and IIJ has modified NetBSD to optimize for the use as a CPE.

A CPE is one of special use cases, so we don't say all of our modifications is worth to merge. Nevertheless, we think some of them are worth to merge and there are some considerable ideas. We mainly describes about three things: filter rule optimization, IPsec interface and Ethernet switch.

1 Implementation and modification for CPE

IIJ has modified the some parts of NetBSD to improve performance and functionalities of our CPE.

Several years ago, IIJ implemented own IP filter and NAT/NAPT functions named *iipf* and *iipfnat*. NetBSD had an IP filter implementation *ipf*, however it didn't satisfy our requirements. *pf* and *npf* wasn't born yet. *iipf* had implemented some ideas to improve throughput. It has a hash-based flow-caching layer. Even if cache-miss occurs, *iipf* keeps reasonable throughput thanks to flow rules that are stored in an optimized tree.

Our IPsec stack also has a caching layer on Security Policy Database (SPD) and Security Association Database (SAD). Because NetBSD's *PF_KEY* API uses list structures for SPD and SAD, throughput will drop if there are a number of SP or SA. A CPE is often used to create VPNs, so the number of SP and SA can be very large. IPsec tunneling is also important for VPN; many customers prefer Route-based VPN to Policy-based VPN. (This topic will be described in another article.)

For small office, Ethernet switch is required. Ethernet switch chip is not expensive and it's easy to integrate into CPE. Integrating Ethernet switch into CPE is better than nothing because both router function and Ethernet switch function can be managed comprehensively.

2 Filter Rule Scan Optimization

In this section, we describe the new optimization of our packet filer.

2.1 Filter rule, state and result cache

On the implementations of general packet filter and old filter implementation on SEIL, the processing speed is proportional to the number of filter rules. If the number of the rules is 100, in the worst case, all 100 rules are checked.

To avoid this problem, the state mechanism is used. When a packet was passed, the interface, source/destination address, source/destination port and so on were saved into an entry of a hash table. And then, a hash value is calculated in each packet and the value is looked up. If the entry was found,

[*]The NetBSD Foundation

[†]Internet Initiative Japan Inc.

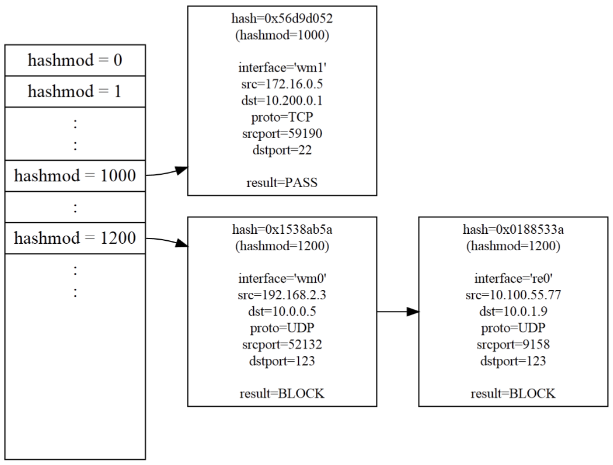


Figure 1: Filter Result Cache

the packet is passed. This mechanism is good because it doesn't scan rules if it is in a hash table. If a lot of data is processed with the same state, the hit ratio is very high. But, if a hash miss occurred a rule scan is done. It's heavy task.

In iipf, it caches a packet's information(interface, source, dest, protocol), the hash value and the result(block or pass) independently of the result. When rule is scanned, at first, a packet's hash value is calculated from the address, protocol and port number, and the value is used to check the result cache. If it exists, the result(block or pass) is returned immediately. If it doesn't exist, rule scan is done, and the packet information, hash value and the result are saved into the result cache for the next check (Figure 1). The entries are managed by LRU algorithm.

2.2 Optimizing rule scan itself

An fundamental way to speedup filtering is to improve the performance of rule scanning which is done when a cache miss happened.

Figure 2 is an example of iipf's filter rules. The top entry of the rules is evaluated first and the bottom entry is evaluated at last. The third column is an identifier string name to specify each rule. With the old implementation, the rule is evaluated like Figure 3. So, the more number of rules increased, the longer the processing speed becomes. One of optimization way is to change such evaluations like Figure 4. New implementation does such optimization when rules are set.

2.3 Implementation

One of the way to implement optimization described above is to use compiler technique. Rules are decomposed into small statements and then those statements are optimized with composer technique. It's possible but the implementation is not easy.

Our solution is splitting filter rule lists using with special condition statements to reduce the number of rules that are scanned. The merit of this way is that it's unnecessary to change the evaluation of each filter rule.

1. Make a list of conditions which are used for splitting rules into two groups. Interface, address, protocol an port are used for the conditions.

2. Select one of conditions.

3. Split filer rules into two groups by checking whether a rule matches or not. If a rule can't be identified whether it matches or not. The rule is put into both groups.

4. For each group, goto step 2 and retry.

5. Stop when the number of rules was decreased a specified limit.

Try this algorithm to Figure 2's rules. Figure 5 is the first try. pppoe0 is selected. An rule that the interface "any" matches both pppoe0 and others, so the rule is put into both groups. Figure 6 is the second try. The next condition is "protocol number is less than 17". The rule of "protocol any" belongs to both groups. Figure 7 is the third try. The next condition is "protocol number is less than 6". The rule of "protocol any" belongs to both groups. The result means that three rules are checked on the worst case. The number was decreased from 6 to 3.

```
filter add LAN        interface lan0   direction in/out                            action pass
filter add PING_PASS  interface pppoe0 direction in     protocol icmp icmp-type 8 action pass
filter add ICMP_BLOCK interface pppoe0 direction in     protocol icmp             action block
filter add DNS_PASS   interface pppoe0 direction in/out protocol udp  dstport 53  action pass
filter add TCP_PASS   interface pppoe0 direction in/out protocol tcp              action pass
filter add BLOCK_RULE interface any    direction in/out protocol any              action block
```

Figure 2: iipf's filter rules example 1

```
/* LAN */
 if (pkt.interface == "lan0")
   return PASS;

 /* PING_PASS */
 if ((pkt.interface == "pppoe0") &&
     (pkt.direction == in) &&
     (pkt.protocol == icmp) &&
     (pkt.icmp.type == 8))
        return PASS;

 /* ICMP_BLOCK */
 if ((pkt.interface == "pppoe0") &&
     (pkt.direction == in) &&
     (pkt.protocol == icmp))
        return BLOCK;

 /* DNS_PASS */
 if ((pkt.interface == "pppoe0") &&
     (pkt.protocol == udp) &&
     (pkt.udp.dstport == 53))
        return PASS;

 /* TCP_PASS */
 if ((pkt.interface == "pppoe0") &&
     (pkt.protocol == tcp) &&
        return PASS;

 /* BLOCK_RULE */
 return BLOCK;
```

Figure 3: Normal processing example of Figure 2 rules

```
if (pkt.interface == "pppoe0") {
  if (pkt.direction == in) {
    if (pkt.protocol == icmp) {
      if (pkt.icmp.type == 8) {
        return PASS;
      } else {
        return BLOCK;
      }
    } else if (pkt.protocol == udp) {
      if (pkt.udp.dstport == 53) {
        return PASS;
      }
    } else if (pkt.protocol == tcp) {
      return PASS;
    } else {
      return BLOCK;
    }
  } else {
    if (pkt.protocol == udp) {
      if (pkt.udp.dstport == 53) {
        return PASS;
      }
    } else if (pkt.protocol == tcp) {
      return PASS;
    } else {
      return BLOCK;
    }
  }
} else {
  if (pkt.interface == "lan0") {
    return PASS;
  }
  return BLOCK;
}
```

Figure 4: Optimized processing example of Figure 2 rules

```
COND_INTERFACE("pppoe0")
  filter add PING_PASS    interface pppoe0 direction in     protocol icmp icmp-type 8 action pass
  filter add ICMP_BLOCK   interface pppoe0 direction in     protocol icmp             action block
  filter add DNS_PASS     interface pppoe0 direction in/out protocol udp  dstport 53  action pass
  filter add TCP_PASS     interface pppoe0 direction in/out protocol tcp              action pass
  filter add BLOCK_RULE   interface any    direction in/out protocol any              action block

!COND_INTERFACE("pppoe0")
  filter add LAN          interface lan0   direction in/out                           action pass
  filter add BLOCK_RULE   interface any    direction in/out protocol any              action block
```

Figure 5: Try 1. Split with pppoe0.

```
COND_INTERFACE("pppoe0")
  COND_PROTOCOL(<17)
    filter add PING_PASS  interface pppoe0 direction in     protocol icmp icmp-type 8 action pass
    filter add ICMP_BLOCK interface pppoe0 direction in     protocol icmp             action block
    filter add TCP_PASS   interface pppoe0 direction in/out protocol tcp              action pass
    filter add BLOCK_RULE interface any    direction in/out protocol any              action block
  !COND_PROTOCOL(<17)
    filter add DNS_PASS   interface pppoe0 direction in/out protocol udp  dstport 53  action pass
    filter add BLOCK      interface any    direction in/out protocol any              action block
!COND_INTERFACE("pppoe0")
  filter add LAN          interface lan0   direction in/out                           action pass
  filter add BLOCK_RULE   interface any    direction in/out protocol any              action block
```

Figure 6: Try 2. Split with protocol number is less than 17.

```
COND_INTERFACE(‘‘pppoe0’’)
  COND_PROTOCOL(<17)
    COND_PROTOCOL(<6)
      filter add PING_PASS  interface pppoe0 direction in     protocol icmp icmp-type 8 action pass
      filter add ICMP_BLOCK interface pppoe0 direction in     protocol icmp             action block
      filter add BLOCK      interface any    direction in/out protocol any              action block
    !COND_PROTOCOL(<6)
      filter add TCP_PASS   interface pppoe0 direction in/out protocol tcp              action pass
      filter add BLOCK      interface any    direction in/out protocol any              action block
  !COND_PROTOCOL(<17)
    filter add DNS_PASS     interface pppoe0 direction in/out protocol udp  dstport 53  action pass
    filter add BLOCK_RULE   interface any    direction in/out protocol any              action block
!COND_INTERFACE(‘‘pppoe0’’)
  filter add LAN            interface lan0   direction in/out                           action pass
  filter add BLOCK_RULE     interface any    direction in/out protocol any              action block
```

Figure 7: Try 3. Split with protocol number is less than 6.

2.4 Selection of condition value

In above example, condition values were selected a little intentionally. In real program, all values which appeared in the rules are tried and the best balanced condition is used.

Next filter rule example is Figure 8. At first step, make a list of conditions which are used for splitting rules into two groups. The list is as follows:

```
INTERFACE = "pppoe0"
SRC < 10.0.0.0
SRC < 11.0.0.0
SRC < 172.16.0.0
SRC < 172.32.0.0
SRC < 192.168.0.0
SRC < 192.169.0.0
PROTOCOL < TCP
DSTPORT < 22
DSTPORT < 24
DSTPORT < 80
DSTPORT < 443
DSTPORT < 512
DSTPORT < 514
```

And then, split rules using with the all conditions. The result is Table 1. The best balanced condition is COND_DSTPORT(<24), so the condition is selected. Then, apply the algorithm with top half of Figure 9. The next candidates are:

```
INTERFACE = pppoe0
SRC < 10.0.0.0
SRC < 11.0.0.0
SRC < 172.16.0.0
SRC < 172.32.0.0
SRC < 192.168.0.0
SRC < 192.169.0.0
PROTOCOL < TCP
DSTPORT < 22
DSTPORT < 24
```

And, the next split candidates and the results are in Figure 2 The best balanced condition is COND_SRC(<11.0.0.0), so the condition is selected. By repeating this, the final result is in Figure 10. This rules can't be split anymore. The result means that three rules are checked on the worst case. The number was decreased from 7 to 2.

2.5 Performance result

The performance result is shown in Figure 11. Test environment is:

- SEIL/B1 (Intel IXP432 400MHz, RAM 128MB)
- Packet length 512bytes
- One direction.
- Incrementing source address in the range of 10.0.0.0/8.

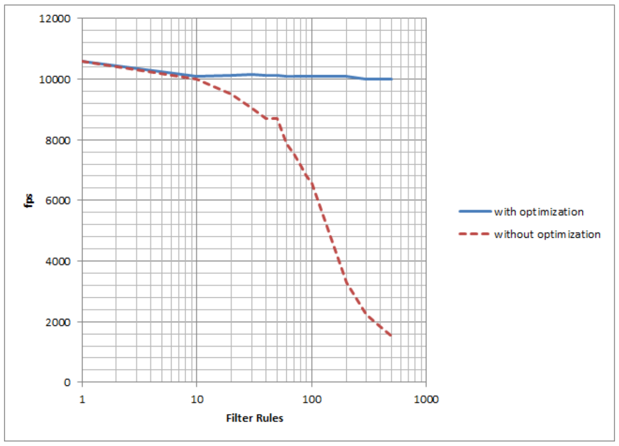


Figure 11: Performance comparison of filter rule optimization

The result showed that the new algorithm works fine and the effect of the number of filter rule is very small.

3 Route packets to IPsec tunnel using routing table

IPsec based VPN is one of important functionalites of IIJ's CPE. Many corporations use IPsec VPN for internal communications. Some corporations have a large number of satellite offices, and redundant data center networks. Each of satellite office has redundant VPN connections to each data center network. So the CPE on a satellite network needs to select one from the redundant connections somehow.

A typical IPsec implementation uses 'Security Association Database(SAD)' to create VPN connections, and uses 'Security Policy Database(SPD)' to select one from the VPN connections. On NetBSD, the SPD is implemented as strictly ordered lists like

```
filter add RULE1 interface pppoe0 src 10.0.0.0/24    protocol any                action pass
filter add RULE2 interface pppoe0 src 172.16.0.0/12  protocol any                action pass
filter add RULE3 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 22-23   action pass
filter add RULE4 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 80      action pass
filter add RULE5 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 443     action pass
filter add RULE6 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 512-513 action pass
filter add RULEB interface any                       protocol any                action block
```

Figure 8: iipf's filter rules example 2

condition hogehoge	rules
COND_INTERFACE("pppoe0")	RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
!COND_INTERFACE("pppoe0")	RULEB
COND_SRC(<10.0.0.0)	RULEB
!COND_SRC(<10.0.0.0)	RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<11.0.0.0)	RULE1,RULEB
!COND_SRC(<11.0.0.0)	RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<172.16.0.0)	RULE1,RULEB
!COND_SRC(<172.16.0.0)	RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<172.32.0.0)	RULE1,RULE2,RULEB
!COND_SRC(<172.32.0.0)	RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<192.168.0.0)	RULE1,RULE2,RULEB
!COND_SRC(<192.168.0.0)	RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<192.169.0.0)	RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
!COND_SRC(<192.169.0.0)	RULEB
COND_PROTOCOL(<tcp)	RULE1,RULE2,RULEB
!COND_PROTOCOL(<tcp)	RULE3,RULE4,RULE5,RULE6,RULEB
COND_DSTPORT(<22)	RULE1,RULE2,RULEB
!COND_DSTPORT(<22)	RULE3,RULE4,RULE5,RULE6,RULEB
COND_DSTPORT(<24)	RULE1,RULE2,RULE3,RULEB
!COND_DSTPORT(<24)	RULE4,RULE5,RULE6,RULEB
COND_DSTPORT(<80)	RULE1,RULE2,RULE3,RULEB
!COND_DSTPORT(<80)	RULE4,RULE5,RULE6,RULEB
COND_DSTPORT(<443)	RULE1,RULE2,RULE3,RULE4,RULEB
!COND_DSTPORT(<443)	RULE5,RULE6,RULEB
COND_DSTPORT(<512)	RULE1,RULE2,RULE3,RULE4,RULE5,RULEB
!COND_DSTPORT(<512)	RULE6,RULEB
COND_DSTPORT(<514)	RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
!COND_DSTPORT(<514)	RULEB

Table 1: First Split candidates and the results

```
COND_PROTOCOL(<24)
        filter add RULE1 interface pppoe0 src 10.0.0.0/24    protocol any                 action pass
        filter add RULE2 interface pppoe0 src 172.16.0.0/12  protocol any                 action pass
        filter add RULE3 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 22-23   action pass
        filter add BLOCK interface any                       protocol any action block
!COND_PROTOCOL(<24)
        filter add RULE4 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 80      action pass
        filter add RULE5 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 443     action pass
        filter add RULE6 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 512-513 action pass
        filter add RULEB interface any                       protocol any                 action block
```

Figure 9: 1st split result

condition	rules
COND_INTERFACE("pppoe0")	RULE1,RULE2,RULE3,RULEB
!COND_INTERFACE("pppoe0")	RULEB
COND_SRC(<10.0.0.0)	RULEB
!COND_SRC(<10.0.0.0)	RULE1,RULE2,RULE3,RULEB
COND_SRC(<11.0.0.0)	RULE1,RULEB
!COND_SRC(<11.0.0.0)	RULE2,RULE3,RULEB
COND_SRC(<172.16.0.0)	RULE1,RULEB
!COND_SRC(<172.16.0.0)	RULE2,RULE3,RULEB
COND_SRC(<172.32.0.0)	RULE1,RULE2,RULEB
!COND_SRC(<172.32.0.0)	RULE3,RULEB
COND_SRC(<192.168.0.0)	RULE1,RULE2,RULEB
!COND_SRC(<192.168.0.0)	RULE3,RULEB
COND_SRC(<192.169.0.0)	RULE1,RULE2,RULE3,RULEB
!COND_SRC(<192.169.0.0)	RULEB
COND_PROTOCOL(<tcp)	RULE1,RULE2,RULEB
!COND_PROTOCOL(<tcp)	RULE3,RULEB
COND_DSTPORT(<22)	RULE1,RULE2,RULEB
!COND_DSTPORT(<22)	RULE3,RULEB
COND_DSTPORT(<24)	RULE1,RULE2,RULE3,RULEB
!COND_DSTPORT(<24)	RULEB

Table 2: Second Split candidates and the results

```
COND_PROTOCOL(<24)
   COND_SRC(<11.0.0.0)
      filter add RULE1       interface pppoe0 src 10.0.0.0/24    protocol any                 action pass
      filter add BLOCK       interface any                       protocol any action block
   !COND_SRC(<11.0.0.0)
      COND_SRC(<172.32.0.0)
         filter add RULE2    interface pppoe0 src 172.16.0.0/12  protocol any                 action pass
         filter add BLOCK    interface any                       protocol any action block
      !COND_SRC(<172.32.0.0)
         filter add RULE3    interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 22-23   action pass
         filter add BLOCK    interface any                       protocol any action block
!COND_PROTOCOL(<24)
   COND_DSTPORT(<512)
      COND_DSTPORT(<443)
         filter add RULE4    interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 80      action pass
         filter add BLOCK    interface any                       protocol any action block
      !COND_DSTPORT(<443)
         filter add RULE5    interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 443     action pass
         filter add BLOCK    interface any                       protocol any action block
   !COND_ DSTPORT(<512)
      filter add RULE6       interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 512-513 action pass
      filter add BLOCK       interface any                       protocol any action block
```

Figure 10: final split

filter rules. Each entry of SPD describes a packet to be secured.

IIJ added some modifications to the NetBSD's implementation. We discuss about the modifications in this section. Figure 12 shows the outline of IIJ's modifications. Gray colored components in the figure are extended by IIJ.

3.1 Problem of typical IPsec implementation

We have 2 problem to use NetBSD as a CPE.

One problem is performance. The implementation of SPD is simple and secure, but we must execute $LIST_FOREACH()$ to each packet. Encryption throughput of our CPE is about 100 - 200 Mbps. If average packet length is 1000 bytes, the packet arriving rate is about 12 kpps to 25 kpps. This means $LIST\_FOREACH()$ will be executed 25,000 times a seconds. And if the SPD has 100 entries, $memcmp()$ will be executed 2,500,000 times a seconds(2.5 MHz!). Of course, the SPD is much smaller on many workstations, but the SPD of VPN devices often have hundreds of entries. SPD grows lager due to number of offices and data centers, and due to number of net-

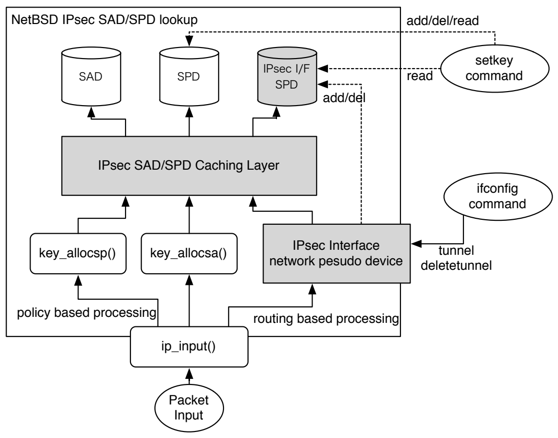


Figure 12: IPsec modifications

work segments in each of networks. It is very easy to grow the SPD.

The other problem is redundancy. SPD is a strictly ordered list, and there is no same order(priority). Each of entry just has a single actions, and there is no way to select multiple connection. It is hard to have a benefit of redundant connections. Some VPN devices can use a routing table instead of SPD. Because there are many existing redundant routing techniques, it easy to have a benefit of the redundant VPN connections. IIJ's CPE supports such routing based IPsec VPN. Here is a example loop in *netipsec/key.c*.

```
struct secpolicy *
key_allocsp(const struct secpolicyindex *spidx,
u_int dir, const char* where, int tag)
{
        struct secpolicy *sp;
        int s;
...
        LIST_FOREACH(sp, &sptree[dir], chain) {
...
                if (sp->state == IPSEC_SPSTATE_DEAD)
                        continue;
                if (key_cmpspidx_withmask(&sp->spidx,
                    spidx))
                        goto found;
        }
...
```

3.2 Improve SAD/SPD lookup performance

IIJ implements software based caching layer to SPD and SAD. The caching code takes a packet header, hash it, then lookup the cache table. The table has a pointer to a SAD/SPD entry. If there is no entry for the packet, scan the SAD/SPD and write it to cache table. This strategy works fine for the CPE. Because the number of node in the corporation's network is much smaller than the real Internet, the flow table doesn't become so large. Our implementation uses 2048 entries for the cache table and it works fine to connect to 100 - 200 satellite networks. Of course, there are some exceptions. For example, random traffics generated by malwares are pollutes the cache table.

There are 13 API functions for SPD/SAD caching layer management, 1 initialization, 4 lookups for each of structure, and 8 invalidates.

```
void key_cache_init();

struct secpolicy *sp_cache_lookup();
struct secashead *sah_cache_lookup();
struct secasvar *sav_cache_lookup();
struct secacq *acq_cache_lookup();

void sp_cache_inval(void);
void sp_cache_inval1(struct secpolicy *);
void sah_cache_inval(void);
void sah_cache_inval1(struct secashead *)
void sav_cache_inval(void);
void sav_cache_inval1(struct secasvar *);
void acq_cache_inval(void);
void acq_cache_inval1(struct secacq *);
```

Cache lookup code is simply inserted before the *LIST_FOREACH*().

```
struct secpolicy *
key_allocsp(const struct secpolicyindex *spidx,
u_int dir, const char* where, int tag)
{
        struct secpolicy *sp;
        int s;
...
        if (key_cache_enable > 0) {
                /* IIJ Extension: lookup cache */
                sp = sp_cache_lookup(spidx, dir);
                goto skip;
        }

        LIST_FOREACH(sp, &sptree[dir], chain) {
```

The hashing algorithm is very important component. The algorithm must be fast enough and must have enough distribution. Unfortunately, there is no specialist of mathematics in IIJ's CPE team, the algorithm should not be a best. Here is our hashing code for your interest.

```
if (src->ss_family == AF_INET) {
        u_int32_t *saddr, *daddr;
        u_int32_t sport, dport;

        saddr = (u_int32_t *)&satosin(src)->sin_addr;
        daddr = (u_int32_t *)&satosin(dst)->sin_addr;
        sport = (u_int32_t)satosin(src)->sin_port;
        dport = (u_int32_t)satosin(dst)->sin_port;

        hash = *saddr ^ bswap32(*daddr) ^
               (sport << 16) ^ dport;
        hash = (hash >> 16) ^ hash;
        hash = (hash >> 4) ^ hash;
}
else if (src->ss_family == AF_INET6) {
        struct in6_addr *saddr, *daddr;
        u_int32_t sport, dport;
        u_int32_t hash128[4];

        saddr = &satosin6(src)->sin6_addr;
        daddr = &satosin6(dst)->sin6_addr;
        sport = (u_int32_t)satosin6(src)->sin6_port;
        dport = (u_int32_t)satosin6(dst)->sin6_port;

        /* stage 1 */
        hash128[0] =
            saddr->s6_addr32[0] ^ daddr->s6_addr32[3];
        hash128[1] =
            saddr->s6_addr32[1] ^ daddr->s6_addr32[2];
        hash128[2] =
            saddr->s6_addr32[2] ^ daddr->s6_addr32[1];
        hash128[3] =
            saddr->s6_addr32[3] ^ daddr->s6_addr32[0];

        /* stage 2 */
        hash128[0] = hash128[0] ^ hash128[3];
        hash128[1] = hash128[1] ^ hash128[2];

        /* stage 3 */
        hash = hash128[0] ^ hash128[1] ^
               (sport << 16) ^ dport;
}
```

3.3 VPN tunnel network device

IIJ implements a VPN tunnel network device named IPsec Interface. The device has BSD name *ipsec*0, *ipsec*1, ..., *ipsecN*. It is a kind of pseudo network device like a IP-IP tunneling device like gif, gre. If a packet is routed into the IPsec interface, the kernel apply IPsec tunnel encryption. There is no need to write a SPD.

The device is controlled by *ifconfig* command as same as *gif* device. When tunnel address is configured, the device create Security Policies automatically. The Security Policies are registered to a SPD other than NetBSD's genuine SPD. i.e. IIJ's kernel has 2 separated SPDs. SP lookup code always looks for genuine SPD 1st, then the IPsec Interface's SPD 2nd. The generated entry is fully compatible with Security Policies of transport mode IPsec to secure tunnel end-point address. Thus, there is no modification for crypto subsystem. And IPsec Interfaces can share the NetBSD's genuine SAD. The code snippet is here. A *LIST_FOREACH* is just added.

```
struct secpolicy *
key_allocsp(const struct secpolicyindex *spidx,
   u_int dir, const char* where, int tag)
{
        struct secpolicy *sp;
        int s;
...
        LIST_FOREACH(sp, &sptree[dir], chain) {
...
        }
#if NIPSECIF > 0
        LIST_FOREACH(sp, &ipsecif_sptree[dir], chain) {
...
        }
#endif
```

setkey command can add or delete entries in genuine SPD but it cannot add or delete entries in IPsec Interface's SPD. But the *setkey* command can read the entries in IPsec Interface's SPD. An IKE server can also read the entries in IPsec Interface's SPD, and create a SA for a entry in IPsec Interface's SPD. We don't need to modify IKE server and most of management services. In kernel IPsec stack also read a entry in IPsec Interface's SPD via APIs in key.c, so we don't need to modify existing IPsec stack. We just modified DB lookup code in key.c. Here is simple example of SPD behavior.

Example 1, configure the interface. IPv6 traffic is dropped by default. Lack of awareness of IPv6 is security risk.

```
# setkey -DP
No SPD entries.
# ifconfig ipsec0
ipsec0: flags=8010<POINTOPOINT,MULTICAST>
        inet6 fe80::2e0:4dff:fe30:28%ipsec0
          ->  prefixlen 64 scopeid 0xf
# ifconfig ipsec0 tunnel 203.0.113.1 203.0.113.2
# ifconfig ipsec0 inet 192.0.2.1
# ifconfig ipsec0
ipsec0: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST>
        tunnel inet 203.0.113.1 --> 203.0.113.2
        inet 192.0.2.1 ->  netmask 0xffffff00
        inet6 fe80::2e0:4dff:fe30:28%ipsec0
          ->  prefixlen 64 scopeid 0xf
# setkey -DP
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        in discard
        spid=36 seq=3 pid=1807
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 4(ipv4)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16402
        spid=34 seq=2 pid=1807
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        out discard
        spid=35 seq=1 pid=1807
        refcnt=1
203.0.113.1[any] 203.0.113.2[any] 4(ipv4)
        out ipsec
        esp/transport/203.0.113.1-203.0.113.2/unique#16401
        spid=33 seq=0 pid=1807
        refcnt=1
#
```

Example 2, setkey cannot delete SP entries for IPsec Interfaces.

```
# setkey -FP
# setkey -DP
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16410
        spid=44 seq=3 pid=2229
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 4(ipv4)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16408
        spid=42 seq=2 pid=2229
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        out ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16409
        spid=43 seq=1 pid=2229
        refcnt=1
203.0.113.1[any] 203.0.113.2[any] 4(ipv4)
        out ipsec
        esp/transport/203.0.113.1-203.0.113.2/unique#16407
        spid=41 seq=0 pid=2229
        refcnt=1
```

Example 3, accept IPv6 traffic. It is controlled by link2 option.

```
# ifconfig ipsec0 link2
# setkey -DP
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16406
        spid=40 seq=3 pid=13654
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 4(ipv4)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16404
        spid=38 seq=2 pid=13654
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        out ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16405
        spid=39 seq=1 pid=13654
        refcnt=1
203.0.113.1[any] 203.0.113.2[any] 4(ipv4)
        out ipsec
        esp/transport/203.0.113.1-203.0.113.2/unique#16403
        spid=37 seq=0 pid=13654
        refcnt=1
#
```

Example 4, unconfigure tunnel.

```
# ifconfig ipsec0 deletetunnel
# setkey -DP
No SPD entries.
#
```

Once IPsec Interface is configured, and IKE server creates SAs for it, we can use the interface as common P2P network interface like gif, ppp, pppoe, and so on. We can manage VPN traffic by RIP, OSPF, floating stack routes, other common routing techniques. We can also use IP Filter on the IPsec Interface. It very easy to have a benefit of redundant VPN connections.

4 Ethernet Switch Framework

One of previous product named SEIL/X2 has an Ethernet switch. The function is almost the same as SA-W1's Ethernet switch chip, but the old code had not enough functions and it's difficult to reuse. At that time, FreeBSD has Ethernet switch function but it's little hardware dependent, so we designed new Ethernet switch framework from scratch.

4.1 Design

The main concept on design is separating code into Ethernet switch common function part and hardware

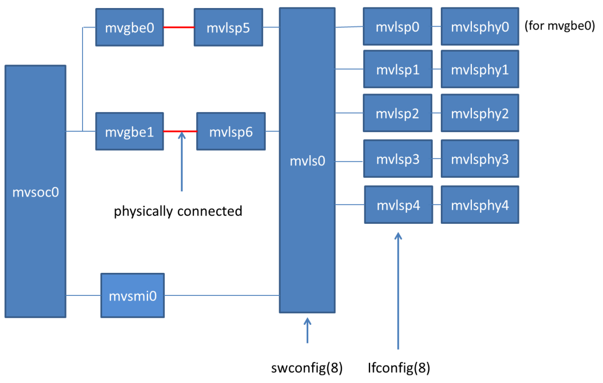
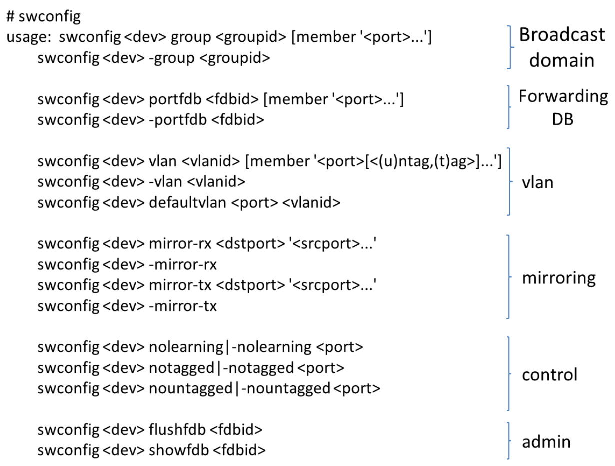


Figure 13: block diagram of SA-W1


```
# swconfig
usage:  swconfig <dev> group <groupid> [member '<port>...']                          Broadcast
        swconfig <dev> -group <groupid>                                                domain

        swconfig <dev> portfdb <fdbid> [member '<port>...']                            Forwarding
        swconfig <dev> -portfdb <fdbid>                                                   DB

        swconfig <dev> vlan <vlanid> [member '<port>[<(u)ntag,(t)ag>]...']
        swconfig <dev> -vlan <vlanid>                                                  vlan
        swconfig <dev> defaultvlan <port> <vlanid>

        swconfig <dev> mirror-rx <dstport> '<srcport>...'
        swconfig <dev> -mirror-rx                                                      mirroring
        swconfig <dev> mirror-tx <dstport> '<srcport>...'
        swconfig <dev> -mirror-tx

        swconfig <dev> nolearning|-nolearning <port>
        swconfig <dev> notagged|-notagged <port>                                       control
        swconfig <dev> nountagged|-nountagged <port>

        swconfig <dev> flushfdb <fdbid>                                                admin
        swconfig <dev> showfdb <fdbid>
```

Figure 14: swconfig(4)

specific part. For example, Ethernet function is separated into if_ethersubr.c and if_xxx.c (e.g. if_bge.c). Like that, Ethernet Switch framework is separated into Ethernet switch common part and hardware specific part. Former is if_etherswsubr.c and latter is mvls.c for SA-W1.

To control Ethernet switch function, we made new command swconfig(8). The main purpose of this command was to hide hardware dependent part. The current function of swconfig is similar to brconfig(4). We think swconfig(8) and brconfig(8) can be integrated into one command.

The driver is separated into two parts. One is driver for controlling switch function(mvls(4)) and another is driver for each port (mvlsp(4)). The ifnet structure is used for those drivers. To control each PHY with mii(4) layer, mvlsply(4) was made and is attached from mvlsp(4) via mii_attach(). Figure 13 is the block diagram of SA-W1's Ethernet switch part.

With this design, ifconfig, netstat, snmp can be used without any modification. the media status and each port's counter can be checked with those programs.

See Figure 14 for the detail of the function of swconfig(4). swconfig(4) calls MI ioctls to control switch functions.

4.2 Current problem

Currently we have some problem. First, though this is not specific to Ethernet switch, there is no best way to know what mechanism is used between Ethernet MAC and switch (or MII PHY), e.g. GMII, RGMII, I2C or something else. So we sometimes have to write it by hard-coding.

Another problem is the relation between the framework and vlan(4). It's little difficult to cooperate with each other.

4.3 Future work

We implemented this framework only for Marvell 88E6171R. We are planning to port this framework to other chips to check whether our design is appropriate or not.

5 Conclusion

Some of implementation can be merged to NetBSD and other *BSD's. For filter rule optimization, the idea can be useful for some other filter implementations.