Passive Covert Channels Implementation in Linux Kernel

Joanna Rutkowska

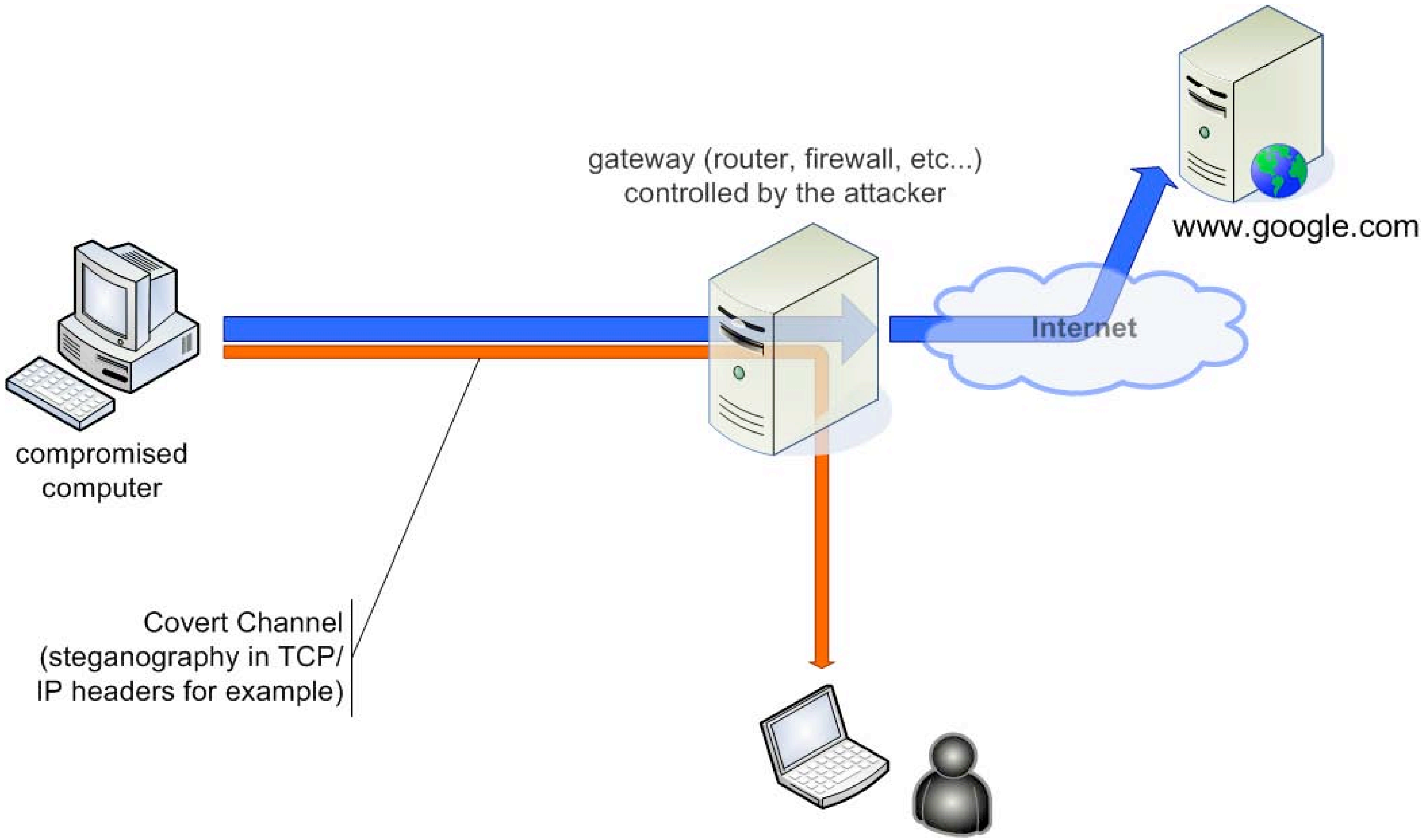
joanna (at) invisiblethings (dot) org

Chaos Communication Congress,
December 27th -29th 2004, Berlin

Passive Covert Channels

- Do not generate their own traffic
- Only change some fields in the packets generated by user (like HTTP requests)
- Best used for stealing data from desktop computers
- Usually requires that the attacker control the company's gateway (for example works in ISP)
- Typical usage: information stealing from corporate Workstations and servers (in "mirror mode", see later)

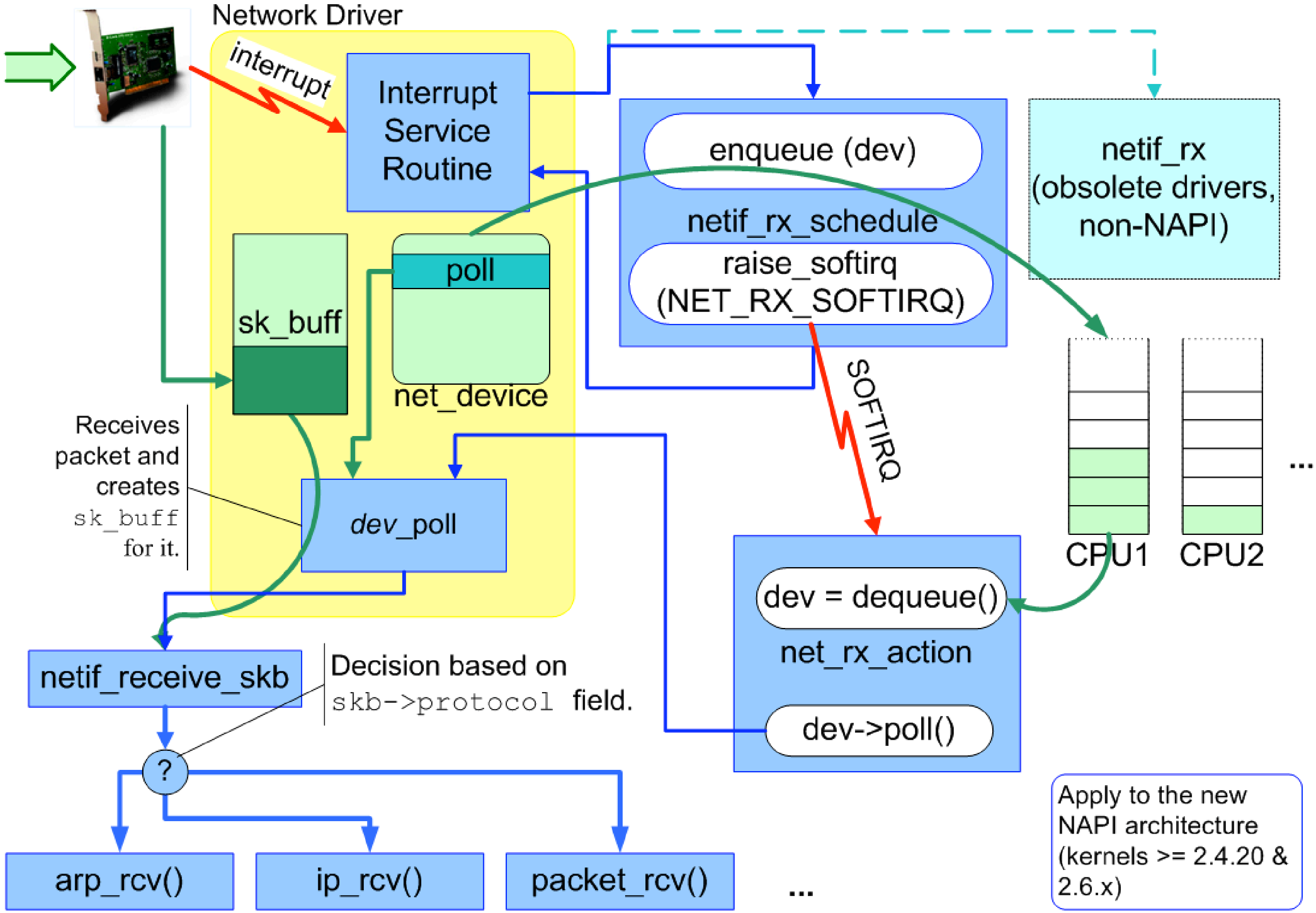
Passive Covert Channels



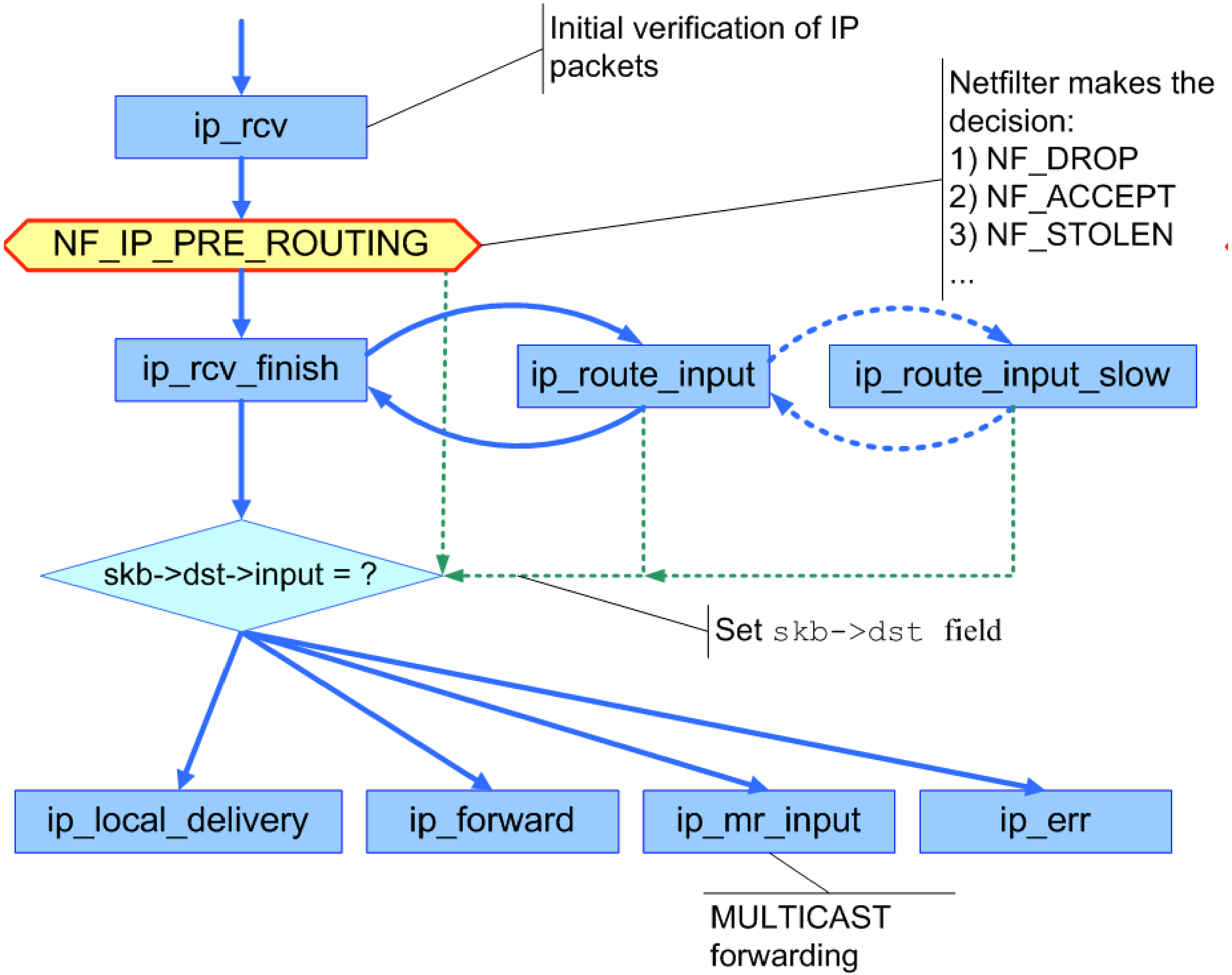
How to implement?

Let's first have a look at how packets are handled inside the Linux kernel…

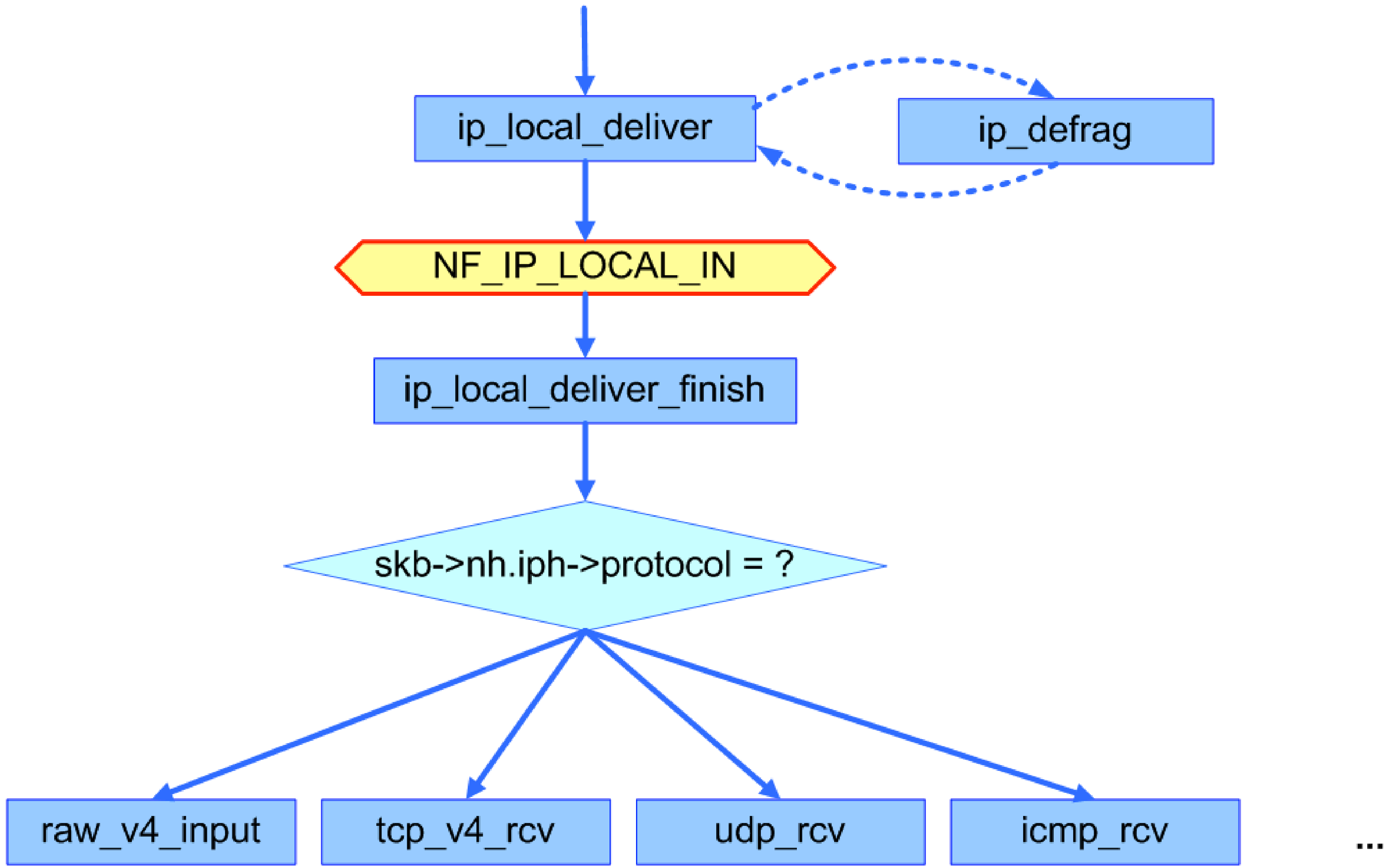
Handling Incoming Packets



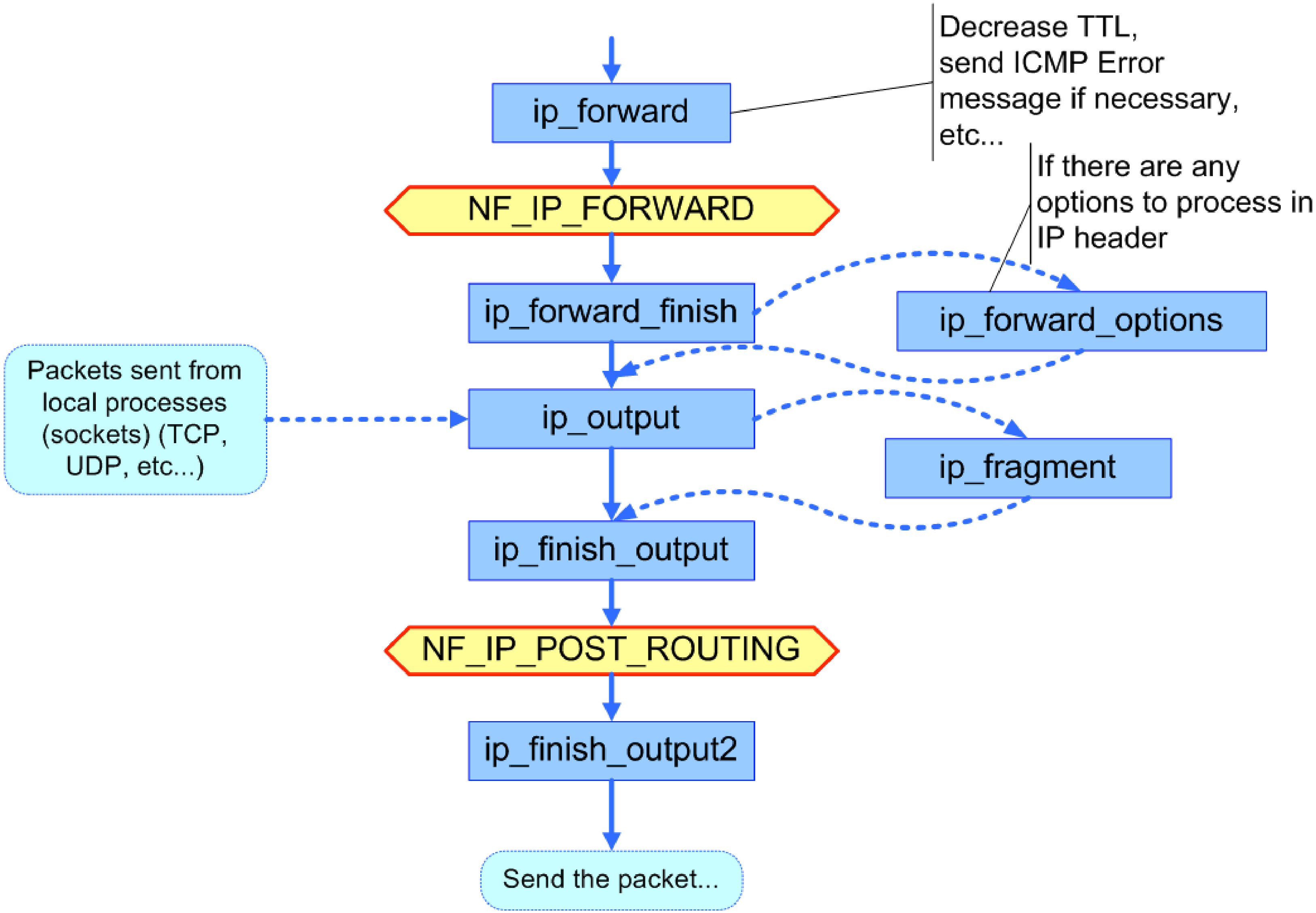
Incoming IP packets



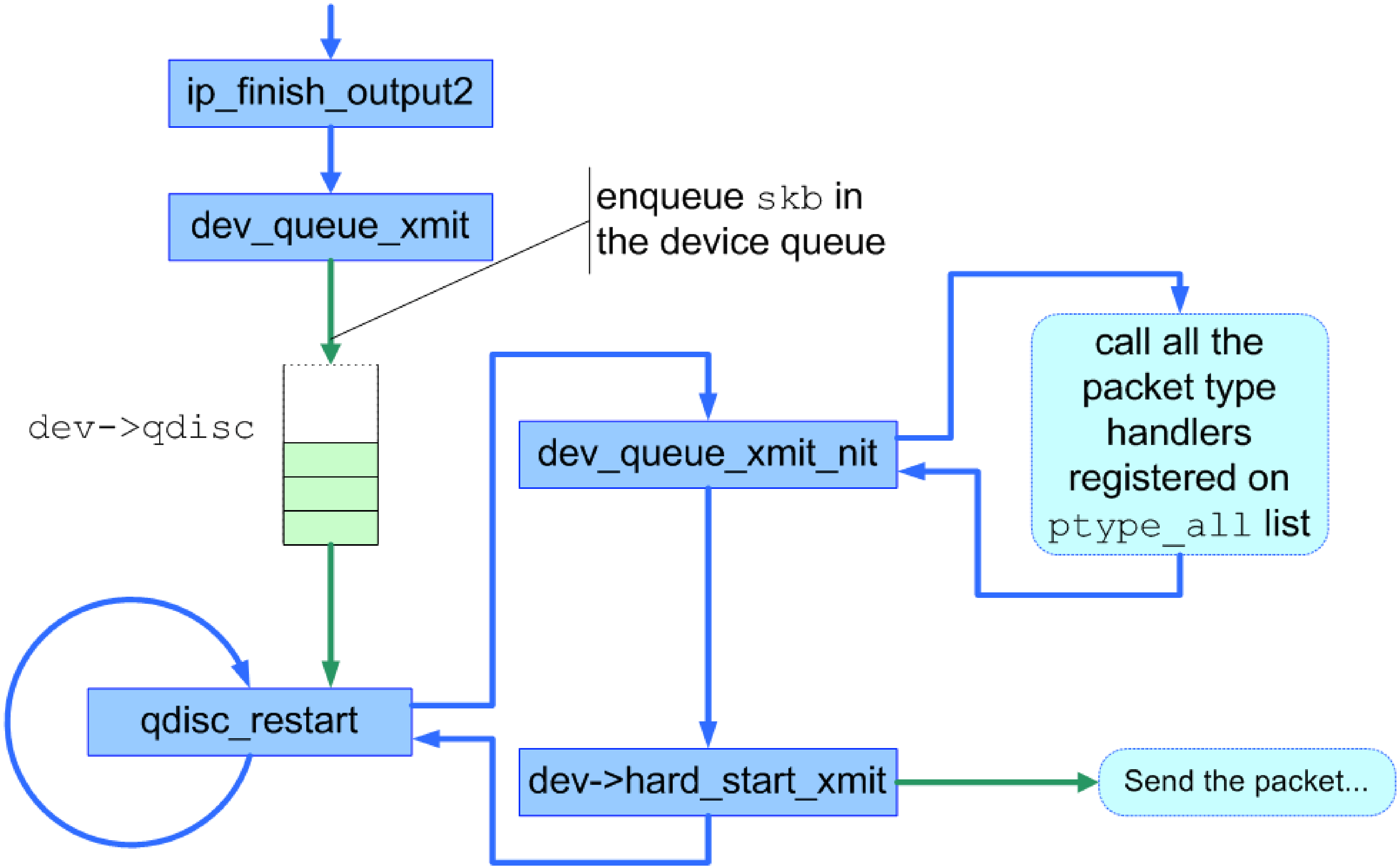
Local delivery



Forwarding packets



Outgoing packets

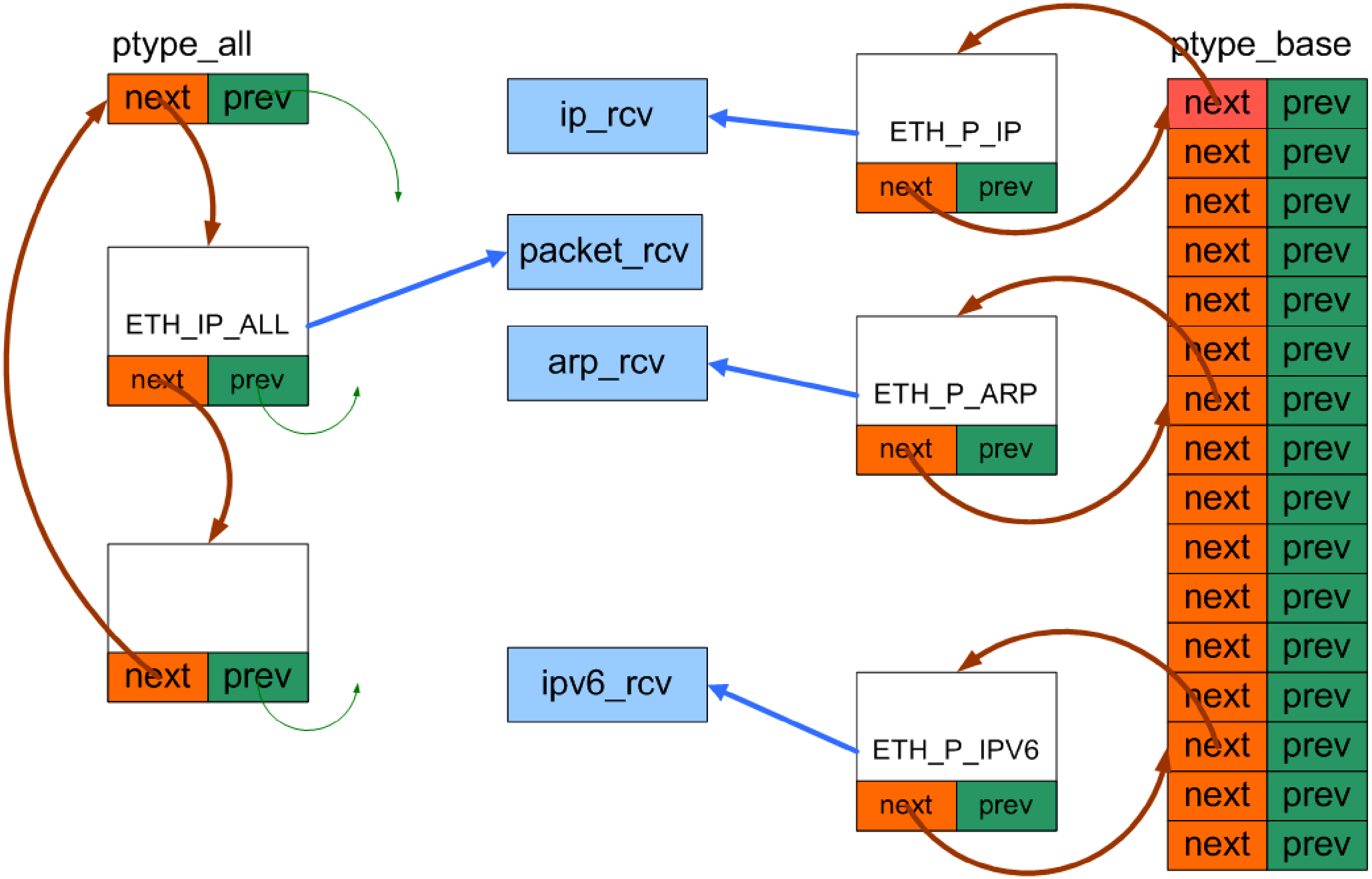


Two important techniques

`ptype_*` handlers

Netfilter hooks

Protocol handlers



Key structure: packet_type

```
struct packet_type
{
 unsigned short    type;  ← htons(ether_type)
 struct net_device *dev;   ← NULL means all dev
 int (*func) (...); ← handler address
 void *data; ← private data
 struct list_head list;
};
```

There are two exported kernel functions for adding and removing handlers:

- `void dev_add_pack(struct packet_type *pt)`
- `void dev_remove_pack(struct packet_type *pt)`

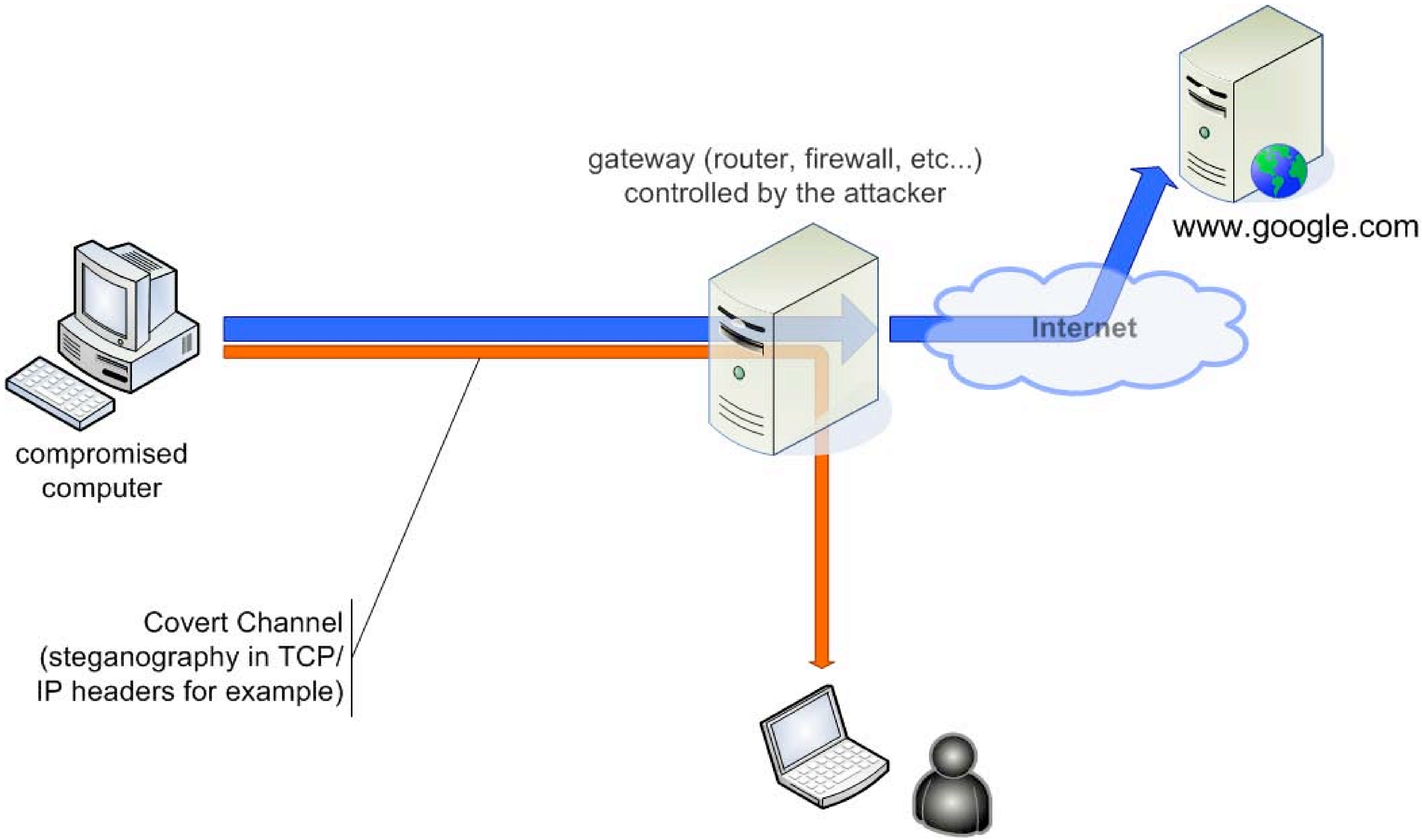
Addition of own handler

```
struct packet_type myproto;

myproto.type = htons(ETH_P_ALL);
myproto.func = myfunc;
myproto.dev  = NULL;
myproto.data = NULL;

dev_add_pack (&myproto)
```

Passive Covert Channels



TCP Header

<table>
<tr><td colspan="10">source port</td><td>destination port</td></tr>
<tr><td colspan="11">sequence number (SEQ#)</td></tr>
<tr><td colspan="11">ack number (ACK#)</td></tr>
<tr><td>Data offset</td><td></td><td>CWR</td><td>ECN</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>window size</td></tr>
<tr><td colspan="10">checksum</td><td>urgent pointer</td></tr>
</table>

The SEQ#, which is transited first is called Initial Sequence Number (ISN)

TCP handshake

SYN: SEQ = ISN_1, ACK = 0

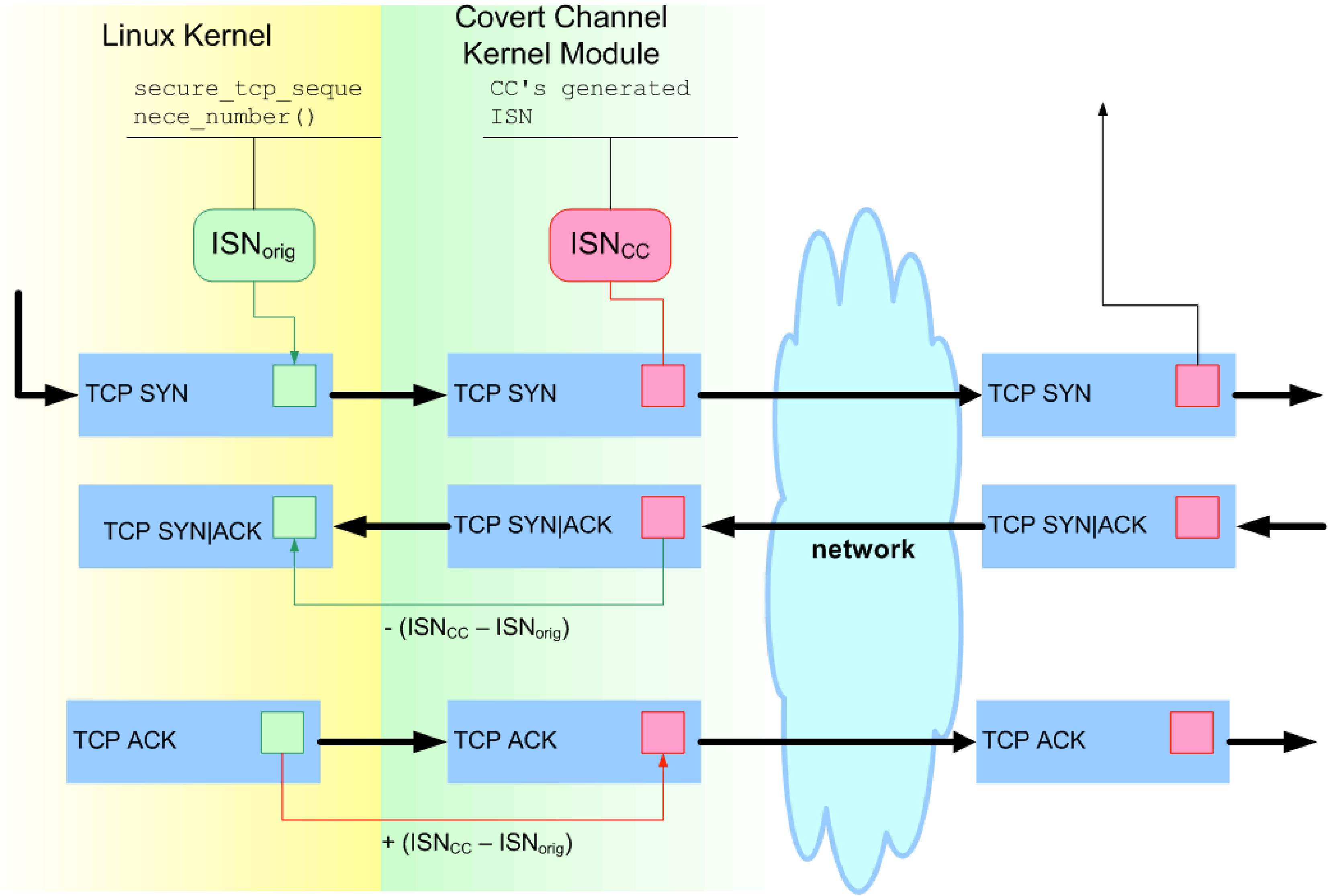
SYN|ACK: SEQ = ISN_2, ACK = ISN_1+1

ACK: SEQ = ISN_1+1, ACK = ISN_2+1

Idea of ISN based passive CC

- Change ISN numbers in all (or only some) outgoing TCP connections (on compromised host)
- Make sure to change back the ACK numbers in incoming connections, co kernel will not discard the packets
- Also, change SEQ# in all consecutive packets belonging to the same TCP connection
- We can send 4 bytes per TCP connection this way.
- Not much, but when considering lots of HTTP connections made by ordinary users it should be ok for sending for e.g. sniffed passwords, etc...

Passive TCP ISN covert channel idea



Tracing TCP connections

- For each TCP connection a block of data is allocated (by a CC kernel module):

```
struct conn_info {
   __u32 laddr, faddr;
   __u16 lport, fport;
   __u32 offset;   // new_isn -
  orig_isn
   struct list_head list;
};
```

- It allows you to correctly change the SEQ numbers of all incoming and outgoing TCP packets

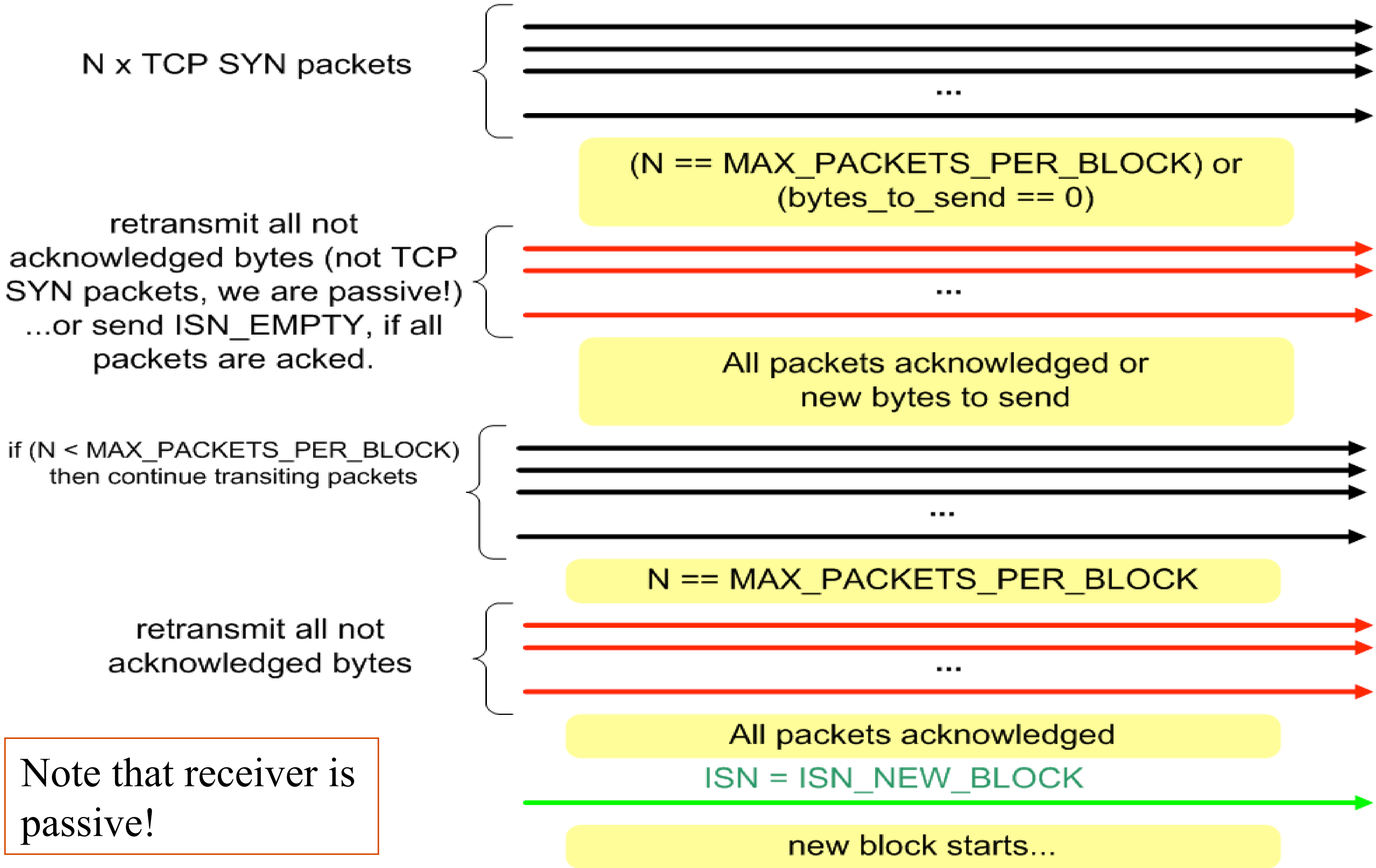
Detecting end of connection

- After the user close the connection it would be nice that the CC module free the `conn_info` structure for that connection (memory in kernel is a an important resource)
- We can implement TCP state machine in CC module to detect when the connection is actually closed (and we don't need to worry about changing its SEQ/ACK numbers anymore)
  - but this is too much work;)
- Another solution: look at the kernel `tcphash_info`, which holds all information about live TCP connections
- From time to time remove dead TCP connection info (`struct conn_info`).

Adding Reliability Layer

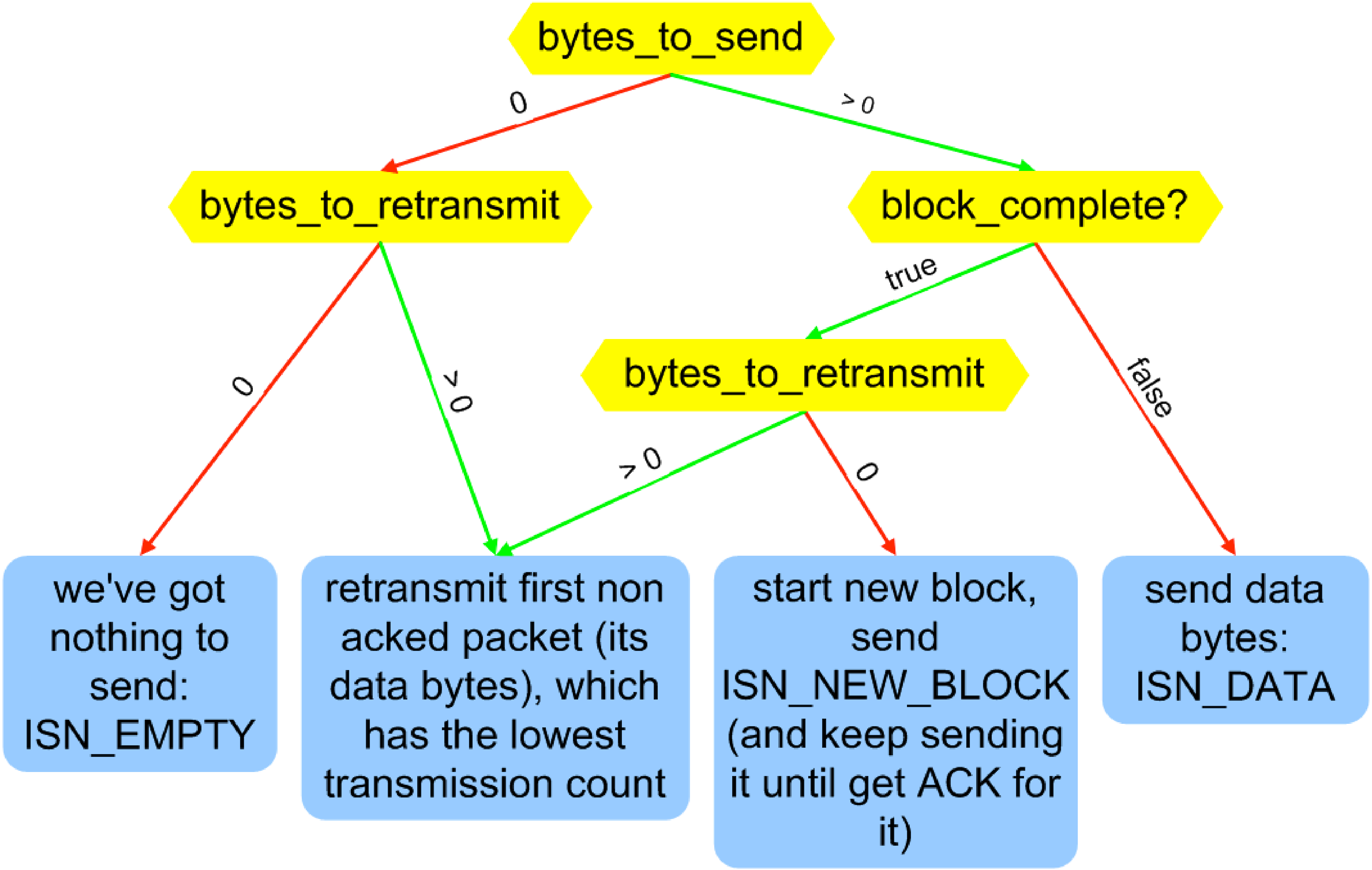
- Any communication channel without reliability mechanism is not really useful outside lab
- In ISN based CC we can exploit the nature of TCP protocol: every SYN packet is acknowledged either by SYN|ACK or by RST packet
- All we need to do is to trace which packets were actually acknowledged
- We need to add packet ordering (our own sequence numbers)

Protocol

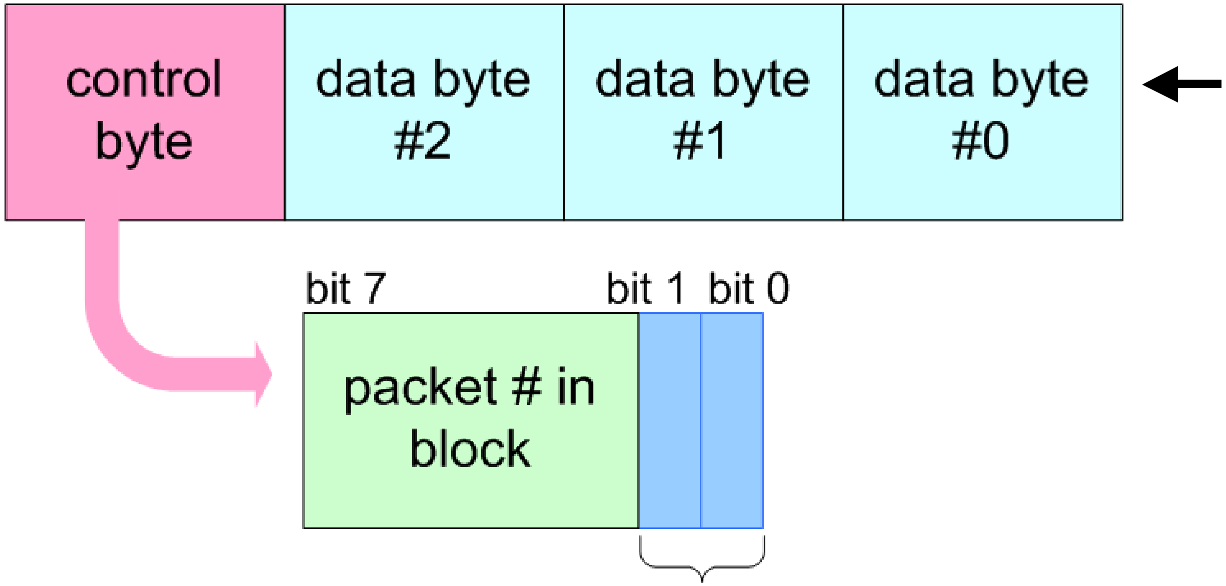


Note that receiver is passive!

Protocol Diagram



Protocol implementation: TCP ISN field



This is the 32bit SEQ (or ACK) field from TCP packet

# of actual data bytes sent in this packet:

`00`: no data (control packet)

`01`: b0 is valid

`10`: b0 & b1 are valid

`11`: b0, b1 & b2 are valid

Special packets

0x00	tcp.sport	0xAA

←ISN_NEW_BLOCK
(starts new transmission black)

0x00	tcp.sport	0xFF

←ISN_EMPTY
(when there is no data to send)

- Special packets contain “random” bytes, to avoid duplicated ISN numbers (which could easy betray the covert channel). Remember that all ISN’s are encrypted with a block cipher before sending to the wire.

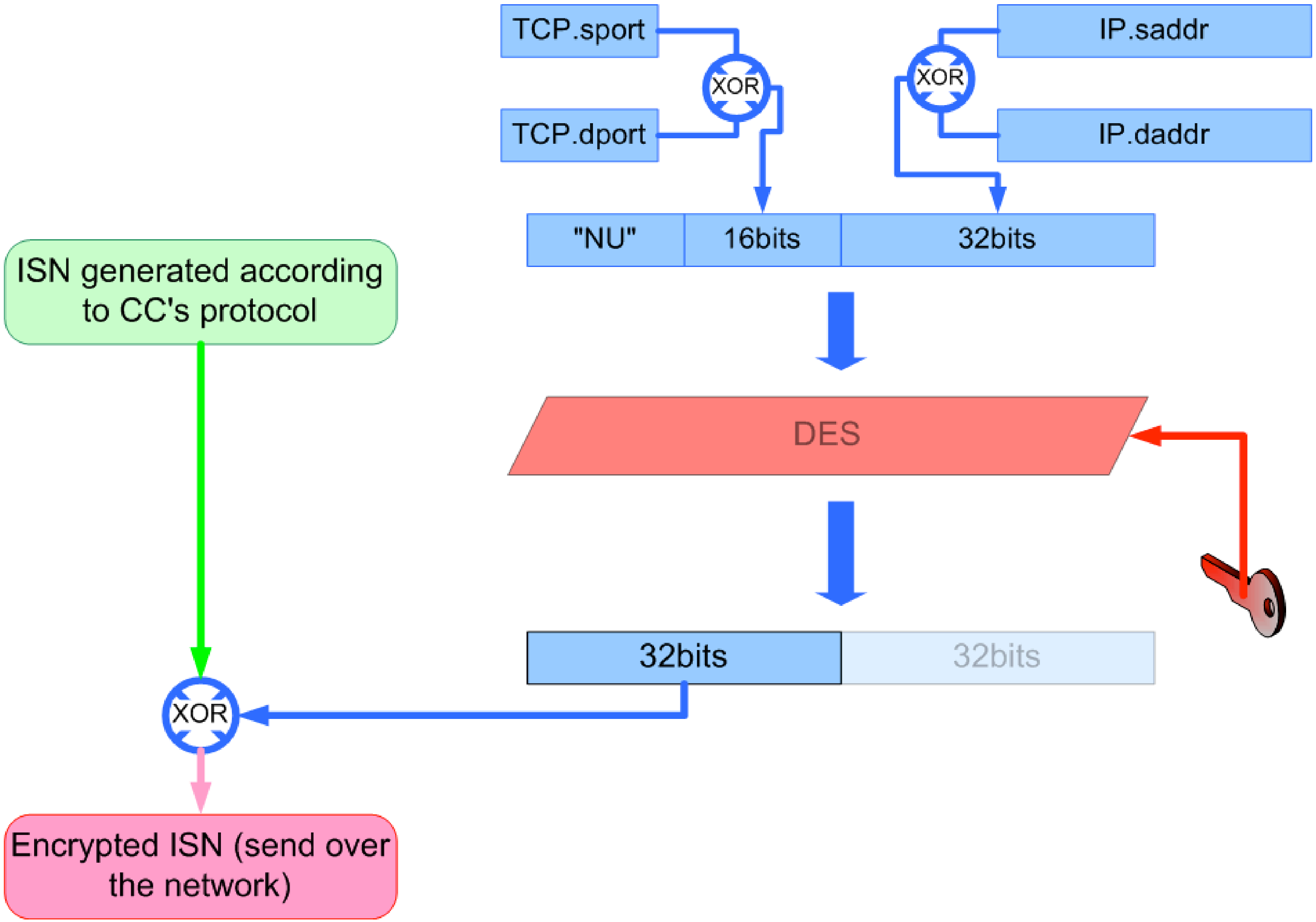
ISN Encryption

- Every ISN, generated by CC protocol engine, is encrypted with a block cipher (see later)
- Both sides share the common key
- Probably the most important thing about the algorithm used is how similar the characteristics of the "random“ numbers it generates are to the ISN numbers generated by the Linux kernel.
- The security of the cipher algorithm plays rather second role here, since it seems unlikely that anybody will try to break it;)

ISN Encryption

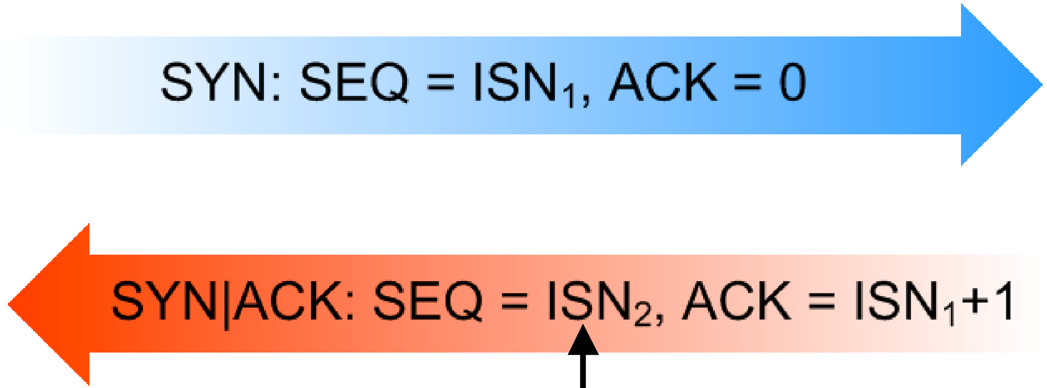
- ISN (SEQ) field is only 32 bit wide
- Most good block cipher operates on blocks greater or equal to 64 bits
- Solution: Use DES to generate a "one-time-pad" key and xor ISN with the lowest 32bits of the generated key.
- We use TCP source and destination port and IP source and destination address as a "seed" to generate key.

ISN Encryption



Encryption

- NOTE: we can only use these elements to generate key, since we need to assure that not only the receiver will be able to decrypt it but also the sender, when decrypting the ACK packet's ACK#!
- This is also the reason for XORing destination and source, so we don't need to worry about reversing them when considering the ACK packet.



We need to be able to decrypt this number too!

Nü Shu __

- Secret language of Chinese women
- Characters were often disguised as decorative marks or as part of artwork
- Existed for centuries, but was not known to most of the world until 1983!

NUSHU – TCP ISN based passive Covert Channel

Features:

- on-the-fly SEQ# changing
- Reliability layer
- PF_PACKET cheating
- For Linux 2.4 kernels (see later discussion on 2.6 kernels)

NUSHU Live Demo

- Time to show some working code :)

PART II

- Inquisitive PF_PACKETS
- Cheating local PF_PACKETs sniffers + DEMO
- "Reverse mode" & bidirectional channels
- Host based detection + DEMO
- Discussion of network based detection
- Some notes about hiding LKMS and LKMs in 2.6 kernels

Inquisitive PF_PACKET sockets

- Q: If you try running `tcpdump` on a host compromised with NUSHU, what will happen?
- A: The outgoing packets will have the ISN displayed correctly (i.e. the ISN inserted by CC). However, the incoming TCP packets will have the ISN displayed incorrectly (i.e. the ISN after the CC changed it)
- Surprisingly, this behavior doesn't depend on whether the PF_PACKET socket (the `tcpdump`'s one) was loaded before or after the CC module registered its handler!

local tcpdump problem

```
[SYN packet as seen on compromised host (172.16.100.2)]:
172.16.100.2.1092 > 172.16.100.1.888: SYN
      4500 003c 03ac 4000 4006 16ec ac10 6402
      ac10 6401 0444 0378 4242 4242 0000 0000
      a002 16d0 7b99 0000 0204 05b4 0402 080a
      0018 0921 0000 0000 0103 0300
```

ISN (SEQ#)

```
[SYN|ACK packet, again, as seen on compromised host]:
172.16.100.1.888 > 172.16.100.2.1092: SYN|ACK
      4500 003c 0000 4000 4006 1a98 ac10 6401
      ac10 6402 0378 0444 1636 5a84 37bf 0a8e
      a012 16a0 1e82 0000 0204 05b4 0402 080a
      0017 2e9d 0018 0921 0103 0300
```

ACK# (should be: 0x43424242)

`skb_clone()` vs `skb_copy()`

```
dev_queue_xmit_nit (skb, ...) {
        skb2 = skb_clone(skb);
        (...)
        ptype->func (skb2);
}
```

- Every ptype handler operates de facto on the same data (`skb->data` is not copied during `skb_clone()`).
- If the CC's ptype handler is called before `PF_PACKETS`'s `packet_rcv()`, then `tcpdump` displays the changed SEQ#.
- When the `packet_rcv()` is called first, the userland process' socket still gets the wrong data, since it effectively reads the data (`skb->data`) after all the kernel stuff is executed on this packet

PF_PACKET Cheating idea

- Redirect all ptype handlers calls, except CC's one, through additional function (`cc_packet_rcv`), which will copy (not clone!) the `skb` buffer and call original handler.
- To do this:
  - Traverse `ptype_all` list and replace all `pt->func` to point to `cc_packet_rcv()`
  - hook `dev_add_pack()` to catch all future ptype registrations

PF_PACKET cheating

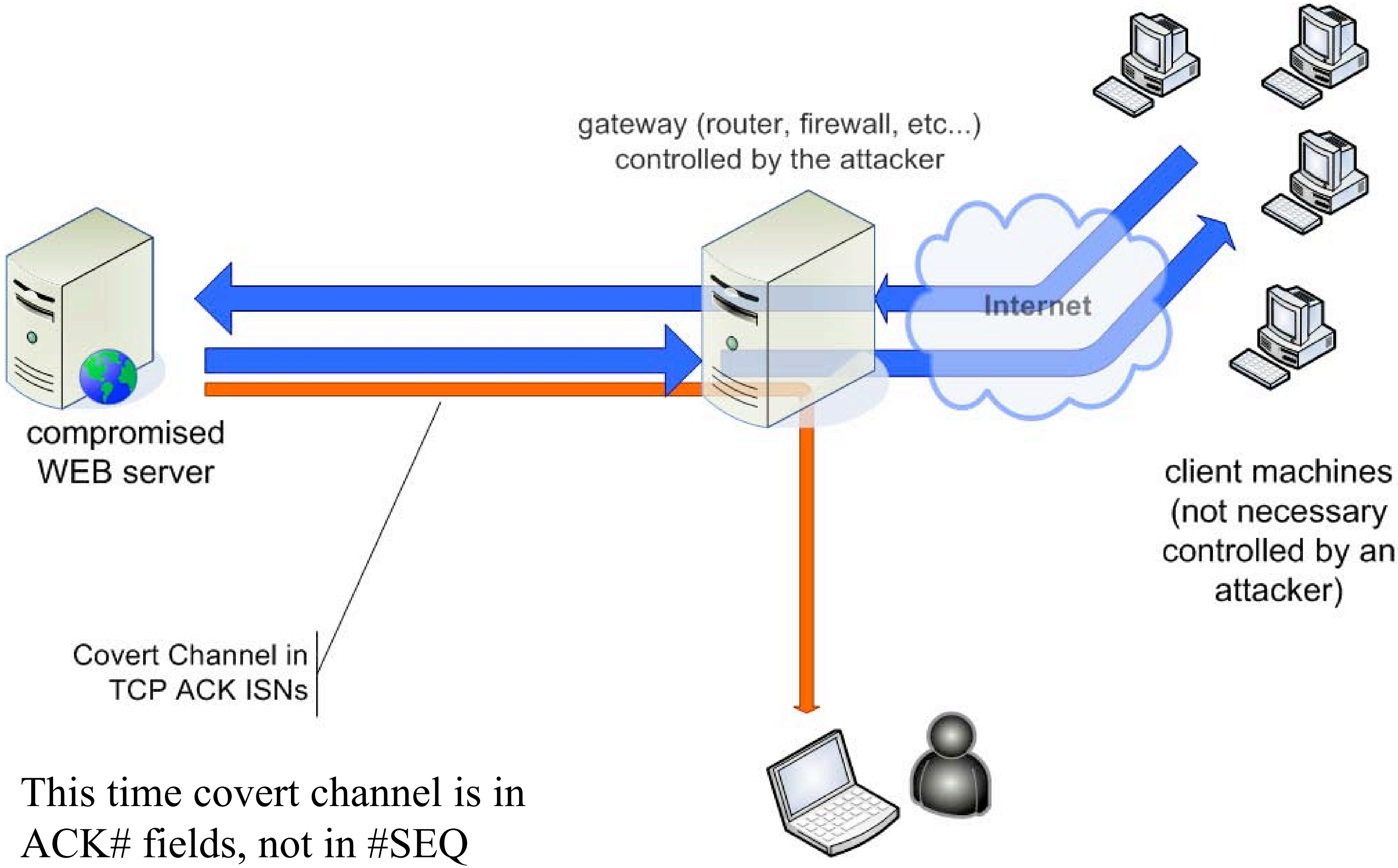
```
int cc_packet_rcv (struct sk_buff skb, …) {
  skb2 = skb_copy (skb);
  if (incoming_packet and
      orig_func != cc_func)
      return orig_func (skb2, …);
  else return orig_func (skb, …);
}
```

```
void cc_dev_add_pack (pt) {
  pt->func = cc_packet_rcv;
  pt->data → {orig_func, orig_data};
}
```

Live DEMO

- NUSHU + PF_PACKET cheating

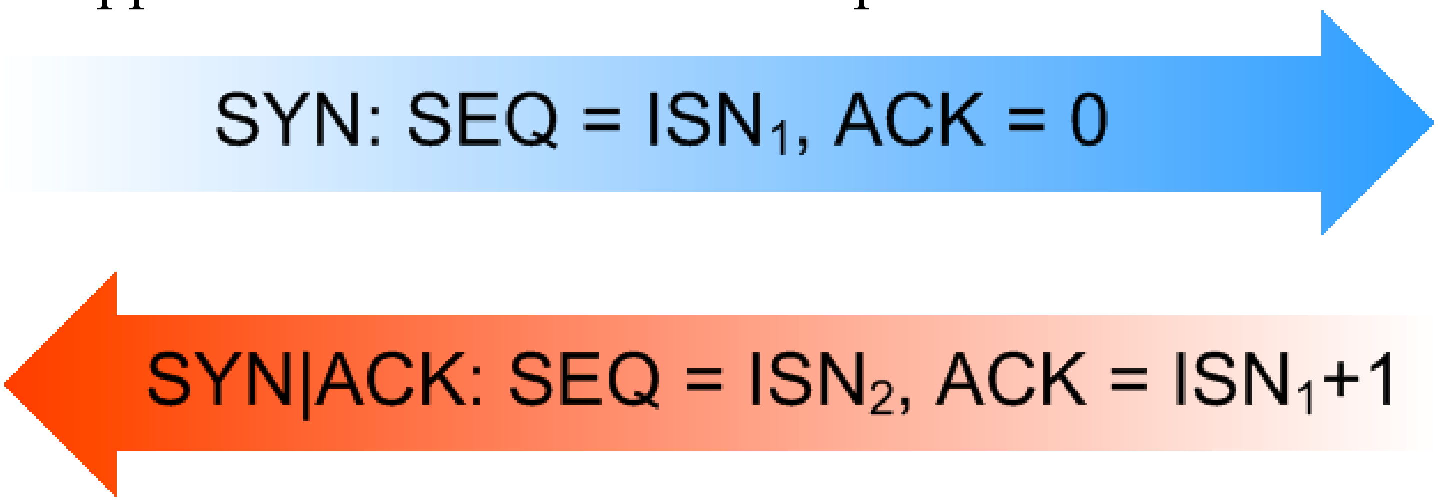
Server mode ("reverse mode")



This time covert channel is in ACK# fields, not in #SEQ

Bidirectional channels

- In this presentation we focused on information stealing, rather then backdoor technology (thus unidirectional channels)
- NUSHU could pretty easily be extended to support bidirectional transmission:
  - one direction: SYN packet's ISN
  - opposite direction: SYN ACK packet's ISN



Covert Channels Detection

Host Based

Network Based

Detecting extra ptype handler (host based detection)

- invasive (requires a special module, which registers a dummy ptype handler for a while)
- noninvasive (does not require any kernel changes, can be implemented through `/dev/kmem`)

How to detect?

- How to get a list of registered protocol handlers?
- Author does not know any tool (or even kernel API) for doing that!
- We need to "manually" check the following lists:
  - `ptype_all`
  - `ptype_base`
- But their addresses are not exported!

Where are the protocol lists?

- Two kernel global variables (`net/core/dev.c`):
  - `static struct packet_type *`**`ptype_base`**`[16];`
  - `static struct packet_type *`**`ptype_all`**` = NULL;`
- Only the following functions are referencing those variables (i.e. "know" their addresses):

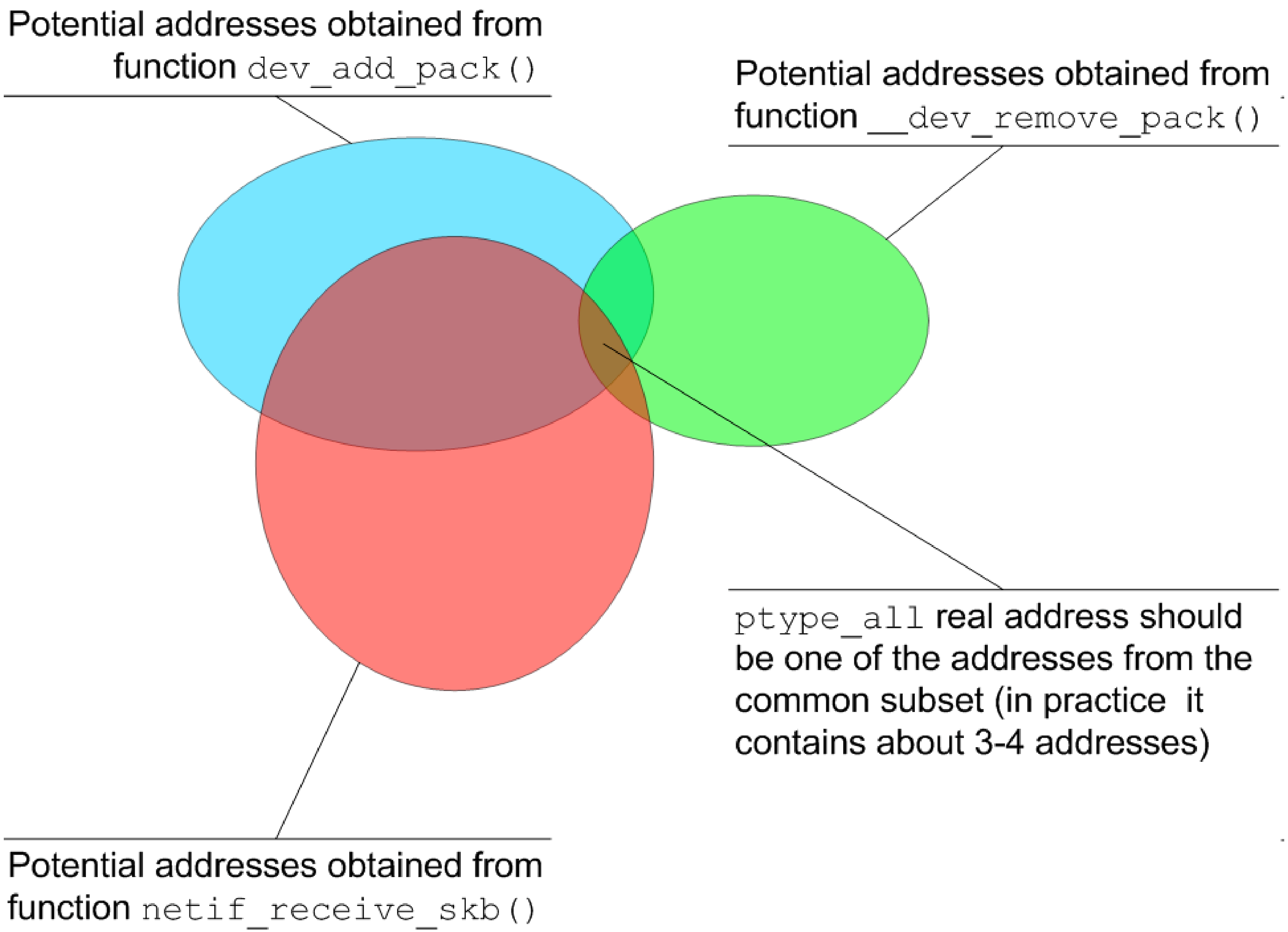
Kernel 2.4.20	*Kernel 2.6.7*
1. dev_add_pack() **2. dev_remove_pack()** 3. dev_queue_xmit_nit() **4. netif_receive_skb()**	**1. dev_add_pack()** **2. __dev_remove_pack()** **3. dev_queue_xmit_nit()** **4. netif_receive_skb()** 5. net_dev_init()

The functions in **green** are exported.

Approaches for finding the lists

- `System.map` file
  - problem: the file is not always up to date or sometimes it does not even exist (for security reasons;))
- "heuristic" method
  - We know the addresses of several functions which are using the addresses we are looking for.
  - We can look at their body to find all the 32 bit words which *look* like kernel pointers.
  - We then need to find the common set of those pointer-like words from all functions we considered.
  - Finally we need to check every potential value from the common subset to see if it *looks* like (or *could be*) the `ptype_all` or `ptype_base` list head.

Illustration for the heuristic method



Live DEMO: detecting additional protocol handlers

PTYPE_ALL:
hook type ETH_P_ALL (0x3)
hook at: 0xc487e060 [module: **unknown module**]

PTYPE_BASE[]:
hook type ETH_P_IP (0x800)
hook at: 0xc0203434 -> ip_rcv() [k_core]

hook type ETH_P_802_2 (0x4)
hook at: 0xc01f8050 [k_core]

hook type ETH_P_ARP (0x806)
hook at: 0xc0223778 -> arp_rcv() [k_core]

"Invasive" method

- Write a little module, which adds its own (dummy) packet type handlers:

```
int dummy_handler (...) { return 0; }
myproto.type = ETH_P_ALL;
myproto.func = dummy_handler;
dev_add_pack (&myproto);
```

- So, you can now traverse the interesting list, starting from: `myproto.next`
- After reading all the handler addresses, you can simple deregister the dummy protocol handler.

Notes about host-based detection

- We mentioned only two possible ways of implementing passive covert channels
  - ptype handlers
  - Netfilter hooks
- These are the easiest (and probably most elegant)
- But there are many other possible ways to create covert channels in the Linux kernel, for example:
  - internal kernel function hooking (biggest problem: most of them are not exported). Quite easy to detect.
  - function pointer hooking, like:
    - `arp_*_ops.hh_output`
    - `net_device.poll`
    - etc...

  ...hard to detect!

host-based backdoor and covert channel detector

Properly implemented host-based compromise detector, should:

- Checks for hidden processes
- Checks for hidden sockets
- Checks ptype handlers (noninvasive method)
- Checks Netfilter hooks
- Checks integrity of kernel code (ala Tripwire)
- Checks important network code pointers

Network Based Detection

- The characteristics of ISN numbers generated by NUSHU will be different from the ISN generated by Linux Kernel.
- We need a reliable method for fingerprinting PRNG
- We have to save the correct PRNG (Linux kernel's) characteristics in a detector database
- The detector measures the characteristics of the suspected TCP flows and compares them against the stored fingerprints (note: detector must be told which exact OSs are running in the network)
- Writing a network based covert channel detector is on my TODO list ;)

Notes about stealth modules

- load module as usual (`insmod`)
- in `init_module()`:
  - allocate some memory by `kalloc()`
  - do not use `vmalloc()`, since such memory goes beyond (`phys_mem + VMALLOC_START`), which makes it easy to detect
  - copy all code and global data to allocated buffer
  - relocate code
- remove module (`rmmod`)
- NOTE: `/dev/kmem` cannot be used on for example Fedora Core 2&3 systems.

Linux 2.6 Considerations

- Changed module loading scheme:
  `http://lwn.net/Articles/driver-porting/`
- There is no compatibility at binary level for modules anymore (no MODVERSIONS)! :-o
- Each module needs to be recompiled for the exact kernel version
  - You can expect some strange incompatibility issues, like different structure layouts between one minor kernel version to another (for example `struct module` has been changed in 2.6.6, breaking all binary compatibility)
- Besides that, seems to be no important differences which would make the implementation difficult

Linux 2.6 LKM hell

- Special macro, VERMAGIC_STRING, has been added to allow checking if the module matches the kernel
- When trying to load `test.ko` module built for Fedora Core 2 on a Slackware 10 system we get the following error (vermagic mismatch):

```
test: version magic '2.6.5-1.358 686
REGPARM 4KSTACKS gcc-3.3' should be
'2.6.7 486 gcc-3.3'
```

- We see a calling convention mismatch and different stack sizes. Loading such module will probably crash the system

VERMAGIC_STRING

include/linux/vermagic.h:

```
#define VERMAGIC_STRING   \
     UTS_RELEASE " "             \  // e.g: "2.6.5-1.358"
     MODULE_VERMAGIC_SMP         \  // "SMP" or ""
     MODULE_VERMAGIC_PREEMPT \  // "preempt" or ""
     MODULE_ARCH_VERMAGIC  \  // see below
     "gcc-" __stringify(__GNUC__) "."    \
     __stringify(__GNUC_MINOR__)  // "gcc-3.3"
```

include/asm-i386/module.h:

```
#define MODULE_ARCH_VERMAGIC \
     MODULE_PROC_FAMILY \  // e.g."PENTIUM4"
     MODULE_REGPARM \      // "REGPARM" or ""
     MODULE_STACKSIZE      // "4KSTACKS" or ""
```

Future work

- Windows port
- Bidirectional channel
- Network based detector (statistical analysis, PRNG fingerprinting)
- Different courier then TCP ISN (HTTP Cookie?)

Credits & Greets

- All members of the #convers channel
- Ian Melven
- Paul Wouters

- JG