ERSPAN Support for Linux

William Tu
VMware Inc.
u9012063@gmail.com

Greg Rose
VMware Inc.
gvrose8192@gmail.com

Abstract

Port mirroring is one of the most common network troubleshooting techiques. Switch Port Analyzer, SPAN, allows a user to send a copy of the monitored traffic to a local or remote device using a sniffer or packet analyzer. Encapsulated Remote SPAN, ERSPAN, extends the basic port mirroring capability from Layer 2 to Layer 3, allowing the mirrored traffic to be sent through an IP network.

ERSPAN was added to Linux kernel in 4.14 for IPv4 and 4.16 for IPv6. In this paper, we demonstrate three ways to use the ERSPAN protocol. First, using iproute2 to create native tunnel net device. Traffic sent to the net device will be encapsulated with the protocol header accordingly and traffic matching the protocol configuration will be received from the net device. Second, for eBPF users, using iproute2 to create metadata-mode ERSPAN tunnel and attach the tunnel metadata implementation in eBPF code. Finally, Open vSwitch users can use netlink interface to create a switch and programmatically parse, lookup, and forward the ERSPAN packets based on flows installed from the userspace.

1. Introduction

Port mirroring is one of the most common network troubleshooting techniques. SPAN (Switch Port Analyzer) allows a user to send a copy of the monitored traffic to a local or remote device using a sniffer or packet analyzer. RSPAN is similar, but sends and received traffic on a VLAN. ERSPAN extends the port mirroring capability from Layer 2 to Layer 3, allowing the mirrored traffic to be encapsulated in an extension of the GRE (Generic Routing Encapsulation) protocol and sent through an IP network. In addition, ERSPAN carries configurable metadatas (e.g., session ID, timestamps), so that the packet analyzer has better understanding of the packets.

ERSPAN for IPv4 was added into Linux kernel in 4.14, and for IPv6 in 4.16. The implementation includes both transmission and reception and is based on the existing ip_gre and ip6_gre kernel modules. As a result, Linux today can act as an ERSPAN traffic source sending the ERSPAN mirrored traffic to the remote host, or an ERSPAN destination which receives and parses the ERSPAN packets generated from Cisco or other ERSPAN-capable switches.

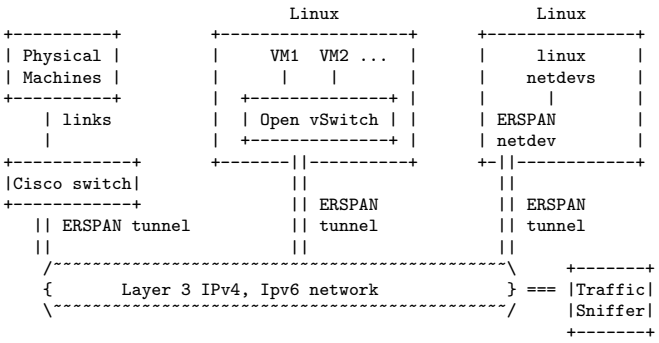


Figure 1: Overview of the ERSPAN tunnel use cases. ERSPAN tunnels can be created from a sniffer machine to Cisco switches, a Linux machine with multiple VMs, or simply a Linux machine.

We have added both the native tunnel support and metadata-mode tunnel support. we demonstrate three ways to use the ERSPAN protocol. First, for Linux users, using iproute2 to create native tunnel net device. Traffic sent to the net device will be encapsulated with the protocol header accordingly and traffic matching the protocol configuration will be received from the net device. Second, for eBPF users, using iproute2 to create metadata-mode ERSPAN tunnel. With eBPF TC hook and eBPF tunnel helper functions, users can read/write ERSPAN protocols fields in finer granularity. Finally, for Open vSwitch users, using the netlink interface to create a switch and programmatically parse, lookup, and forward the ERSPAN packets based on flows installed from the userspace.

ERSPAN is popular in the followin use cases [3]:

- Debugging network issues by tracking the control and data frames.
- Monitoring Voice-over-IP, VoIP, packets for delay and jitter analysis
- Monitoring network transactions for latency analysis
- Monitoring network traffic for anomaly detection

Figure 1 shows an example setup of ERSPAN tunnels. A network administrator first sets up multiple source network devices and filters the interested portion of the traffic he/she wants to inspect. One case on the left-most is to create the ERSPAN tunnel between the Cisco switch and a traffic snif-

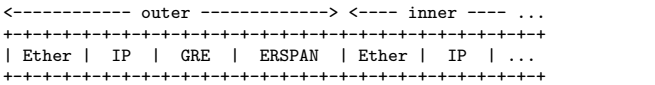


Figure 2: An example of mirrored packet with outer header containing the GRE and ERSPAN header, followed by the inner Ethernet frame.

fer. Depending on the features in the Cisco switch, different filters can be applied to the traffic. In the middle of the figure, for multiple virtual machines running inside a Linux box, the virtual switch forwarding the packet between virtual and physical networks can also create ERSPAN tunnels between the software switch and remote traffic sniffer. Here, Open vSwitch [7] is an example capable of creating filters and forwarding packets to ERSPAN tunnels. More detailed configurations of Open vSwitch are described in later section.

The ERSPAN tunnel is represented in Linux as a netdev and configured through iproute2 [4]. Any packet that is placed into its send queue will be encapsulated based on the netdev's ERSPAN configuration. As a result on the right-most, any other linux netdev which wants to create a ERSPAN mirrored packet simply makes a copy and forwards to the ERSPAN netdev. For example, a physical netdev can use linux TC [8] with mirror action to copy a packet to the erspan tunnel.

Mirrored traffic arriving at the sniffer machine needs to be able to extract and restore the original monitored frame. To differentiate the three use cases, the administrator can create three ERSPAN session IDs, a configuration parameter for grouping the mirrored traffic. For Linux users, an ERSPAN tunnel can also be used at the sniffer side. Any packet arriving at the ERSPAN tunnel netdev's receive queue will be decapsulated. Tools such as Wireshark [10, 11] can be used to inspect the mirrored packet.

2. ERSPAN Protocol Implementation

The ERSPAN protocol was developed by Cisco and its specification is published at IETF draft [3]. Figure 2 shows an example of ERSPAN encapsulated packet, with outer header consisting of Ethernet header, following by IPv4/IPv6 header, following by a fixed 8-byte GRE header, and following by ERSPAN header. After the ERSPAN header, the inner frame is followed so that the ERSPAN receiver or packet sniffer can extract the original frame. The use of the IP protocol as part of the outer header is important because it makes the mirrored traffic routable across any IP network.

ERSPAN protocol has two versions; version 1 (type II) and version 2 (type III). ERSPAN protocol is layered on top of the GRE (Generic Routing Encapsulation) protocol, with GRE's sequence number enabled. For ERSPAN type II, the GRE's next protocol type is 0x88BE with 8-byte ERSPAN header size, and for ERSPAN type III, the GRE's next protocol type is 0x22EB with 12-byte ERSPAN header size, if no optional subheader enabled.

In this section we describe the basic ERSPAN protocol header format along with its implementation in the Linux kernel. For IPv4/IPv6, the implementation is under net/ipv4/ip_gre.c and net/ipv6/ip6_gre.c. Also a userspace API header, include/uapi/linux/erspan.h is added for metadata-mode tunnel users.

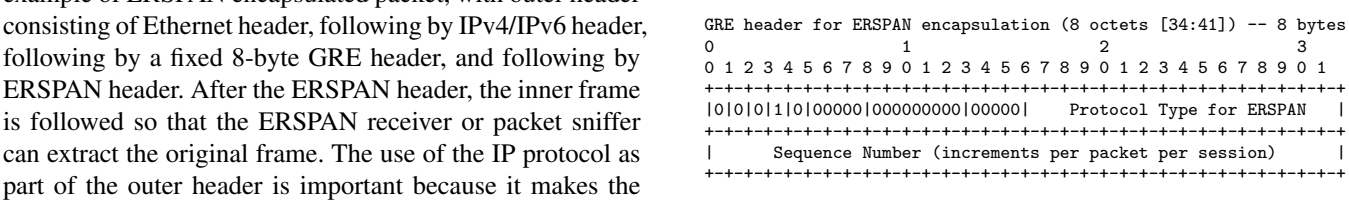
2.1 Native vs Metadata-Mode Tunnel

There are two tunnel type implementations in Linux kernel: native tunnel and metadata-mode tunnel [5]. Native tunnel is the basic way of creating tunnels in Linux. A tunnel netdev is created with per tunnel-specific configuration, tied together with the netdev. For example, creating a GRE tunnel with key and sequence number can be done by: `ip link add dev gre123 type gretap local 1.1.1.1 remote 2.2.2.2 seq key 0xfb`. As a result, N different tunnel configurations require creating N number of netdevs. In certain cases such as network virtualization, this is not scalable because every host in the network creates mutiple tunnels with different configurations to every other hosts [6].

Metadata-mode tunnel, or called light-weight tunnel, is designed for solving the limitation. The fundamental idea is that only one netdev per tunnel type is required to represent multiple tunnels. This means that the tunnel configuration of a particular type of the tunnel must be passed to the tunnel netdev in order to encapsulate the packet. For example, creating a metadata-mode tunnel can be done by: `ip link add dev type gretap external`. Note that there is no configuration parameters assigned at device creation time. The tunnel configuration is set-up per-packet at run-time. Currently there are two ways of using metadata-mode tunnel, one through OVS and the other through eBPF [1]. We implement both the native mode and metadata-mode [13] for ERSPAN type II and type III. More examples of using native and metadata-mode tunnel are upstreamed under tools/testing/selftest/bpf/{test_tunnel.sh, test_tunnel_kern.c}.

2.2 GRE

ERSPAN follows a fixed 8-byte GRE header with the below value.



Note that only the sequence number bit in the FLAGS fields is set. Sequence number is useful at the snifffer site where the mirrored traffic arrives out-of-the-order. Depending on the protocol type, ERSPAN type II or type III is followed next.

Implementation: Before introducing ERSPAN, Linux kernel already supports IPv4 GRE native and metadata mode. So our effort is to purely add ERSPAN implementation on top of existing GRE code base [12]. One minor

limitation is that existing metadata-mode does not support GRE sequence number, we've upstreamed the implementation [14].

For IPv6, there is no metadata mode feature before 4.16. We first implemented the metadata-mode support for IPv6 GRE [17], then upstreamed the ERSPAN feature [15, 16].

2.3 ERSPAN Type II

ERSPAN type II has 8-byte feature header with the following format.

```
ERSPAN Version 1 (Type II) header (8 octets [42:49])
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Ver  |          VLAN         | COS | En|T|    Session ID     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Reserved         |                    Index                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The ERSPAN Type II encapsulation adds to the original frame a composite header comprising: 14-byte (802.3) + 20-byte (IP) + 8-byte (GRE) + 8-byte (ERSPAN), in addition to a trailing 4-byte Ethernet CRC. The VLAN field shows the original VLAN of the frame, the COS means Class of Service of the monitored frame. En field shows the trunk encapsulation type associated with the ERSPAN source port. When the mirrored frame is truncated, T bit is set to indicate the frame has been truncated. Session ID is a 10-bit field as an identification of each ERSPAN mirroring session. Index is a platform-depedent field for specifying port number and direction.

Implementation: Type II introduces two new configurable fields to netlink API; the Session ID and Index. Session ID is configured by users through iproute2 tool with netlink API. Since ERSPAN does not use the GRE Key field, we re-use the `IFLA_GRE_IKEY, IFLA_GRE_OKEY` as the session ID field. Index is also configurable by users through iproute2. The COS field and VLAN field are extracted from the original frame and set properly. The truncate bit is detected by comparing the the mirrored frame's `skb->len` and the length its IP header reports.

2.4 ERSPAN Type III

ERSPAN type III has 12-byte feature header with the following format.

```
ERSPAN Version 2 (Type III) header (12 octets [42:49])
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Ver  |          VLAN         | COS |BSO|T|     Session ID    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Timestamp                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             SGT               |P|    FT   |   Hw ID   |D|Gra|O|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type III introduces more flexible composite header to support additional fields. BSO, Bad/Short/Oversized, allows the sniffer to identify whether the frame payload has CRC error, too short, or too large [18]. Timestamp is a 4-byte field and can be configured with different granularities (100 microseconds, 100 nanosecond, or IEEE1588) at the Gra field.

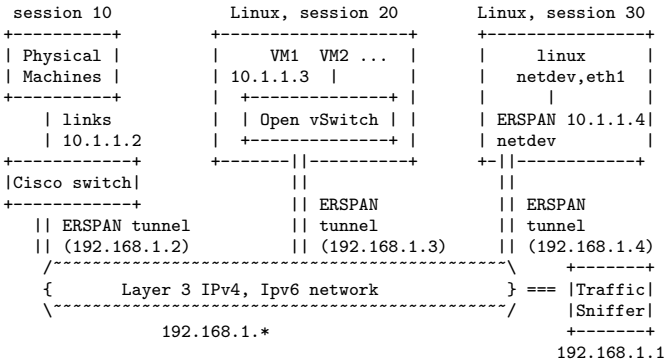


Figure 3: Example of creating three ERSPAN monitoring sessions, $10, 20, 30$, in a $10.1.1.*$ internal network, mirroring traffic over a $192.168.1.*$ IP network.

SGT stands for security group tag of the monitored frame, P field indicates that the ERSPAN payload is an Ethernet protocol frame. FT,Frame Type, indicates whether the mirrored frame is a Ethernet 802.3 frame, or a IP packet. Hardware ID, Hw ID, is an unique identifier of an ERSPAN engine, Direction bit, D, indicates whether the original frame was SPAN'ed in ingress(0) or in egress(1). Finally, O indicates whether or not the optional platform-specific subheader is presented.

Implementation: For type III, we introduced another two fields to kernel through netlink API; hardware ID and direction. The COS, BSO, and T fields can be extracted or inferred from the mirrored frame. Timestamp value is calculated by calling the kernel `ktime_get_real()` with 100 microseconds granularity. Currently we do not support other timer granularties. In addition, the SGT is hard-coded to 0, non-ethernet mirrored packet is not supported, so FT is always 0 and P is set to 1. There is no implementation of sub-headers, so O bit is 0.

3. Example Use Case

We use Figure 3 as an example topology to demostrate the three configuration ways of ERSPAN. Assuming a network administrator wants to monitor a network consisting of 1) physical machines connected to Cisco switches, 2) virtualized Linux machine with multiple VMs deployed and virtual switch (openvswitch.ko) enforcing the fowarding policies, and 3) non-virtualized Linux physical machines, e.g., service nodes in data center such as gateways. Assuming all the monitored servers are under IP network of $10.1.1.*$, and We place a traffic sniifer over another IP network of $192.168.1.*$. The following subsection describes the configuration for each case.

3.1 Cisco Switch

We use Nexus 5000 switch and configure its ERSPAN tunnel with session ID 10, remote IP pointing to the traffic sniffer, 192.168.1.1, and local IP address as 192.168.1.2. As a result, both the ingress and egress traffic on ports 11 and 12 will be msdfirrored to the remote sniffer.

```
monitor session 10 type erspan-source
  erspan-id 10
  vrf default
  destination ip 192.168.1.1
  source interface Ethernet1/11 both
  source interface Ethernet1/12 both
  no shut
monitor erspan origin ip-address 192.168.1.2 global
```

3.2 Open vSwitch Kernel Module

Open vSwitch consists of two components: a userspace daemon, called ovs-vswitchd, and a flow cache as a kernel module, called openvswitch.ko. While ovs-vswitchd talks to the OpenFlow controller and programs its OpenFlow flow tables, the openvswitch.ko keeps a cache where the subsequent flows are handled inside the kernel space.

The openvswitch.ko provides a user-facing netlink API that models a network bridge that connects multiple ports through a single table [9]. This example shows how to use the netlink API provided by openvswitch.ko module, with the utility, ovs-dpctl, to create a ERSPAN tunnel.

```
# creating datapath named "mydp", attach veth1(port 2)
ovs-dpctl add-dp mydp
ovs-dpctl add-if mydp veth1 // connected to VM1

# creating erspan dev named "myerspan" and attach
# lightweight tunnel is used with "external" keyword
ip link add dev myerspan type erspan external
ovs-dpctl add-if mydp myerspan

# flow entry for port 1 to erspan tunnel port 3
ovs-dpctl add-flow \
  "in_port(1),eth(src=00:01:02:03:04:05,dst=10:11:12:13:14:15),\
  eth_type(0x0800),ipv4(src=35.8.2.41,dst=172.16.0.20,proto=5,\
  tos=0x80,ttl=128,frag=no)" \
  "set(tunnel(tun_id=20,dst=192.168.1.1,ttl=64,\
  erspan(ver=2,dir=1,hwid=0x4),flags(df|key))),3"

ovs-dpctl dump-flows
```

3.3 iproute2 with/without eBPF

Assuming we want to mirror all traffic from the physical device eth1 to an ERSPAN tunnel with session ID 30, as shown in the right Figure 3, we first create a native-mode ERSPAN tunnel using ip-link command, and mirror traffic from eth1 to the ERSPAN tunnel netdev. For eBPF use case, instead of creating native-mode tunnel, we create a metadata-mode tunnel using the key word ”external”. Then, tc qdisc and filter rules are created for a eBPF program [1, 2], ”test_tunnel_kern.o” with section name ”set_erspan” to be executed, when receiving a packet from eth1 and sending through the ”myerspan” tunnel device.

```
# Native-mode without using eBPF
ip link add dev myerspan type erspan seq key 30 \
    local 192.168.1.4 remote 192.168.1.1 \
    erspan_ver 1 erspan 123

# eth1 is the mirrored device
tc qdisc add dev eth1 handle ffff: ingress
tc filter add dev eth1 handle ffff: ingress matchall \
    skip_hw action mirred egress mirror dev myerspan
```

```
# Metadata-mode with eBPF
ip link add dev myerspan type erspan external
tc qdisc add dev myerspan clsact
tc filter add dev myerspan egress bpf direct-action \
    obj test_tunnel_kern.o section erspan_set_tunnel

tc qdisc add dev eth1 handle ffff: ingress
tc filter add dev eth1 parent ffff: matchall \
    skip_hw action mirred egress mirror dev myerspan
```

Note that for metadata-mode tunnel, the tunnel configuration is not provided from the ip route command line, but is passed in to the tunnel by the eBPF program, test_tunnel_kern.o. A code snippet creating this object from tools/testing/selftests/bpf/test_tunnel_kern.c is shown below.

```
SEC("erspan_set_tunnel")
int _erspan_set_tunnel(struct __sk_buff *skb)
{
    struct bpf_tunnel_key key;
    struct erspan_metadata md;
    int ret;

    __builtin_memset(&key, 0x0, sizeof(key));
    key.remote_ipv4 = 0xc0a80101; /* 192.168.1.100 */
    key.tunnel_id = 30; // session ID
    key.tunnel_tos = 0;
    key.tunnel_ttl = 64;

    ret = bpf_skb_set_tunnel_key(skb, &key, sizeof(key),
                    BPF_F_ZERO_CSUM_TX);
    if (ret < 0) {
        ERROR(ret);
        return TC_ACT_SHOT;
    }

    __builtin_memset(&md, 0, sizeof(md));
    md.version = 1;
    md.u.index = bpf_htonl(123);

    ret = bpf_skb_set_tunnel_opt(skb, &md, sizeof(md));
    if (ret < 0) {
        ERROR(ret);
        return TC_ACT_SHOT;
    }

    return TC_ACT_OK;
}
```

4. Conclusion

Port mirroring is the most common troubleshooting technique allowing a user to send a copy of the monitored traffic to a packet analyzer. ERSPAN extends its precedences, SPAN and RSPAN, by allowing the monitored traffic to route across IP networks. In this paper, we describe the implementation of ERSPAN and demonstrate three ways to use the ERSPAN protocol in Linux, by creating a native-mode ERSPAN tunnel , or by creating eBPF byte-code with metadata-mode tunnel, or by using Open vSwitch kernel module. We’d like to thank many people for giving review comments feedbacks for our ERSPAN patches.

References

[1] Daniel Borkmann. Advanced programmability and recent updates with tc's cls_bpf. *NetDev 1.2*, 2016.

[2] Daniel Borkmann. On getting tc classifier fully programmable with cls bpf. *NetDev 1.1*, 2016.

[3] M. Foschiano. Cisco Systems' Encapsulated Remote Switch Port Analyzer (ERSPAN). `https://tools.ietf.org/html/draft-foschiano-erspan-00`, October 2014.

[4] T Graf. Lightweight flow based encapsulation. Linux kernel. `https://lwn.net/Articles/651497/`, August 2016.

[5] T Graf. Lightweight flow based encapsulation. Linux kernel. `https://lwn.net/Articles/651497/`, August 2016.

[6] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan J Jackson, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, volume 14, pages 203–216, 2014.

[7] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[8] Jamal Hadi Salim. Linux traffic control classifier-action subsystem architecture.

[9] Joe Stringer. Openvswitch without Open vSwitch: The API and its users. *NetDev 2.1*, 2017.

[10] W Tu. Erspan: add link to protocol spec and refactoring. Wireshark, commit 03bc58d07276, July 2016.

[11] W Tu. Erspan: support platform specific sub-header. Wireshark, commit 147cac3af73d, July 2016.

[12] W Tu. ERSPAN version 2 (type III) support. `https://lwn.net/Articles/741771/`, August 2016.

[13] W Tu. gre: add collect_md mode for ERSPAN tunnel. Linux kernel, commit, August 2016.

[14] W Tu. gre: add sequence number for collect md mode. Linux kernel, commit 77a5196a804e, December 2017.

[15] W Tu. ip6_gre: add erspan v2 support. Linux kernel, commit 94d7d8f29287, December 2017.

[16] W Tu. ip6_gre: add ip6 erspan collect_md mode. Linux kernel, commit ef7baf5e083c, December 2017.

[17] W Tu. ip6_gre: add ip6 gre and gretap collect_md mode. Linux kernel, commit 6712abc168eba, December 2017.

[18] W Tu. erspan: set bso bit based on mirrored packet's len. Linux kernel, commit d48f1958ab7d, May 2018.