Parse Program Design

Overview

This document gives an introduction on how the program has been adjusted to the IPv6 packets. To make the parse program work for IPv6 data, a large portion of the changes we made is the definition of a new `Set` data structure for IPv6 data. We also added one more struct called `EventSource` to solve the new problem caused by the new design. For other existing data structures, we modified the `Decoder`, `EventSignature`, and `EvenPackets` struct. Then, we adjusted corresponding methods of those structs. Finally, we defined a new struct for caching subnet data before flushing it out to some output file.

A New Set for IPv6

As IPv6 exceeds the limit of the integer type, we used a byte array with a size of 16 to store the IPv6 data

```
type IPSet struct {
   M map[[16]byte]struct{}
}
```

The corresponding methods, such as `NewIPSet`, `Add`, and `ByteSize`, were adjusted from the `Uint32Set` struct that stores IPv4 data. Note that the `ByteSize` method

```
func (s *IPSet) ByteSize() int {
    return len(s.M) * 16
}
```

returns 16 times the length size of the IPv6 set, as IPv6 addresses are 16 bytes. The new set for IPv6 is in the newly created file named `set_netip.go` under the set folder.

Adding and Modifying Data Structures

We also need to modify other data structures to consider operations related to IPv6 addresses.

1. In `decode.go`, we added these fields in the `Decoder` struct

   ```
   type Decoder struct {
       // ...
       ip6       layers.IPv6
       icmp6     layers.ICMPv6
       // ...
   }
   ```

   to receive and decode the IP layer's data. As we added these two new fields, we also added these tokens when we initialize a layer parser in `NewDecoder`

   ```
   d.parser = gopacket.NewDecodingLayerParser(layers.LayerTypeEthernet,
      &d.eth, &d.ip4, &d.ip6, &d.icmp4, &d.icmp6, &d.tcp, &d.udp, &d.pay)
   ```

2. In `event.go`, we changed the `EventSignature` and `EventPackets` struct to two interfaces

```
type EventSignature interface {
   GetPort() uint16
   GetTraffic() TrafficType
}
```

```
type EventPackets interface {
    Add(destIP net.IP, b uint64, t time.Time) int
    AddSample(i int, raw []byte)
    Size() uintptr
    GetFirst() time.Time
    GetLatest() time.Time
    GetPackets() uint64
    GetBytes()   uint64
    GetSamples() [][]byte

    // Generated by msgp
    DecodeMsg(dc *msgp.Reader) error
    EncodeMsg(en *msgp.Writer) error
    MarshalMsg(o []byte) ([]byte, error)
    UnmarshalMsg(bts []byte) ([]byte, error)
    Msgsize() int
}
```

Here it is because we want to distinguish between IPv4 and the IPv6 packets. Then, structs related to IPv4 and IPv6 data will implement these interfaces.

```
// EventSignature is the data structure used to associate different packets
to
// a single source and/or being of the same event.
type EventSignatureIPv4 struct {
    SourceIPv4 uint32
    Port      uint16
    Traffic   TrafficType
}

type EventSignatureIPv6 struct {
    SourceIPv6 [16]byte
    Port      uint16
    Traffic   TrafficType
}
```

Also, we created a new interface called `EventSource` to take care of all the `msgp` calls

```
// EventSource handles all msgp-related functions
type EventSource interface {
    // Generated by msgp
    DecodeMsg(dc *msgp.Reader) error
    EncodeMsg(en *msgp.Writer) error
    MarshalMsg(o []byte) ([]byte, error)
    UnmarshalMsg(bts []byte) ([]byte, error)
    Msgsize() int
}
```

The reason is that we use the `EventSignature` interface as a map key in the `Cache` struct. For interfaces, we can either assign a *concrete* type to an interface type or a pointer to the interface type. If we used a pointer to the interface type, then it would have caused an even larger overhead, as each pointer was unique. We could also use a concrete type (i.e. `EventSignatureIPv4` and `EventSignatureIPv6`), but this requires that the function receiver type must be changed to *non-pointer*, so all their method sets have a function receiver type of non-pointer. This requires us to manually modify the auto-generated code in `event_gen.go`. However, by introducing the `EventSource` interface, we avoid including `msgp`-related functions in the `EventSignature` interface, which makes the interface act as a proper key of the map. At the same time, we can still keep all the necessary information we want for annotating darknet events.

3. In `annotate.go`, we made a new struct to hold the subnet that we were interested

```
type Unique24s struct {
    Unique24sIPv4 set.Uint32Set
    Unique24sIPv6 set.IPSet
}
```

The purpose was to create an indirection to fit conditions on IPv4/v6 data in runtime and to reduce the new lines of code that were necessary for the output schema.

For the output schema, two new fields were introduced

```
Output {
    // ...
    UniqueDests:    uniqueDestsSize,
    UniqueDest24s: unique24sSize,
    // ...
}
```

`UniqueDests` is for the number of unique IP addresses, either in IPv4 or IPv6; `UniqueDest24s` is for the subnet `/24`.

Ways to distinguish between IPv4 and IPv6

By `net.IP.To4()`

As we have two different kinds of struct handling different IPs, the program also introduces ways to distinguish between these two types of IP and their corresponding data structures. One way is by the property of `net.IP`. If an instance of `net.IP` calls the `To4()` function but gets a `nil`, that means the actual IP address behind this `net.IP` instance is IPv6. Thus, we have

```
if d.ip4.DstIP.To4() != nil {
    es = analysis.NewEventSignatureIPv4(d.ip4.SrcIP, ...)
    ip = d.ip4.DstIP.To4()
} else {
    es = analysis.NewEventSignatureIPv6(d.ip6.SrcIP, ...)
    ip = d.ip4.DstIP.To16()
}
```

when seeing IPs at the first time.

By switch statements on the interfaces that are implemented by different structs

After creating those IPv4-specific or IPv6-specific structs, we can apply switch statements on interfaces.

```
// 1. switching on the EventSignature interface
// `es is an interface type of EventSignature
switch es.(type) {
    case analysis.EventSignatureIPv4:
        // do things for IPv4 here
    case analysis.EventSignatureIPv6:
        // do things for IPv6 here
}

// 2. switching on the EventSignature interface
// `ep` is an interface type of EventPackets
switch ep.(type) {
    case *analysis.EventPacketsIPv4:
        // do things for IPv4 here
    case *analysis.EventPacketsIPv6:
        // do things for IPv6 here
}
```