Implementation of the Internet Checksum

TSVWG, IETF113

Tom Herbert <tom@sipanda.io>
Internet checksums

- Checksums are 16-bit values
- Protocols define a two byte checksum field
- Computation: Ones’ complement two byte sum from a start offset (e.g. first byte of TCP header) to end offset (e.g. end of packet)
- Sender and receiver perform same algorithm and get same answer if data not corrupted
- Examples: TCP and UDP checksum, IPv4 header checksum, GRE checksum
Checksum computation

The Internet checksum employs ones’ complement sum to validate correct receipt of data (indicate by $\oplus$).

1. Add two binary numbers of some word size
2. Add generated carry (1 or 0) to the sum to get result

Example:

$0xD2 \oplus 0x6A = 0x3D$
Arithmetic properties

- 0xFFFF mathematically equivalent to 0
  \[ A \oplus 0xFFFF = A \oplus 0 = A \]

- Commutative and associative
  \[ A \oplus B = B \oplus A, \ (A \oplus B) \oplus C = A \oplus (B \oplus C) \]

- Checksum subtraction via “not” operation
  \[ A \oplus \sim A = 0xFFFF \ // \ i.e \ checksum \ 0 \]

- Checksums of a larger word size can be folded to a smaller word size with equivalency
Setting an Internet checksum

1) Perform one’s complement sum from start offset (e.g. first byte of TCP header) to end offset (e.g. end of packet)
2) Include pseudo header in sum if required by protocol
3) Not the result (i.e., $Res=\neg Sum$)
4) Set the result in the checksum field
Validating a protocol checksum

1) Perform one’s complement sum from start offset (e.g. first byte of TCP header) to end offset (e.g. end of packet)
2) Include pseudo header in sum if required by protocol
3) If result == 0xFFFF then checksum is valid

TCPCsum ⊕ PseudoHdrCsum = 0xFFFF?
Optimizing checksum calculation

- Computation of the Internet checksum is pretty expensive (add ins. & cache misses)
- Checksum over small data, like IP header checksum, warrants specialized instructions
- Checksum over large payload, like TCP checksum, warrants checksum offload
Pseudo code (naive method)

sum16 = 0;
if (len % 2 == 1) { data[len] = 0; len++ }
for (i = 0; i < len / 2; i++) {
    sum16 = sum16 + *(__u16 *)&data[i*2];
    if (sum16 > 0xffff) {
        sum16 = sum & 0xffff;
        sum++;
    }
}
Optimization: add larger words

- Consider computing checksum over four 16-bit words: $A_0 A_1 A_2 A_3$
  \[ S = A_0 \oplus A_1 \oplus A_2 \oplus A_3 \]
- Perform ones’ complement 32-bit addition
  \[ T_0 T_1 = A_0 A_1 \oplus A_2 A_3 \]
- “Fold checksum” to 16 bits
  \[ S = T_0 \oplus T_1 \]
Pseudo code for 32-bit words

sum32 = 0;
for (i = 0; i < len / 4; i++) {
    sum32 = sum32 + *__u32__ & data[i*4];
    if (sum32 > 0xFFFFFFFF) {
        sum32 = sum32 & 0xFFFFFFFF;
        sum32++;
    }
}

/* "Fold checksum to 16 bits */
Folding a checksum to 16 bits

- Convert 32-bit 1s’ complement value to 16 bits
- Ones’ complement add the low word to the high word

```c
sum16 = sum32 & 0xFFFF;
sum16 = sum16 + (sum32 >> 16)
if (sum16 > 0xFFFF) {
    sum16 = sum16 & 0xFFFF;
    sum16++;
}
```
Add with carry instruction

- Many CPU architectures have specialized instruction to perform one’s complement add
- In x86:
  - Carry bit is in a processor control register
  - Plain `add` instructions set carry bit
  - `adc` performs an addition and includes the value of the carry bit
Example: IPv4 header (8 byte words)

// Computing checksum (64-bit checksum)
omvq (%1), %0    // Load first eight bytes
adcq 8(%1), %0   // Add second word to first, set carry
adcl 16(%1), %0  // “Add with carry” third word
adcq $0, %0      // Add in final carry

// Folding 64-bit checksum to 16 bits
movl %0, %2      // Fold to thirty-two bits
shrq $32, %0
adcl %2, %0     // Fold to sixteen bits
movw %0, %2
shrl $16, %0
adcw %w2, %w0
adcw $0, %w0
Checksum delta

- A change in one field of packet can be offset by another to maintain correct checksum
  - e.g. updating TCP checksum for NAT addr changea
- Example: consider: $A_0A_1A_2A_3\ldots A_N$
  - where $A_3$ is checksum field. Support contents of $A_1$ are modified to make $A'_1$ then
  - $\Delta = \neg(A'_1 \oplus \neg A_1) = A_1 \oplus \neg A'_1$
  - $A'_3 = A_3 \oplus \Delta = A_3 \oplus (A_1 \oplus \neg A'_1)$
  - Which gives: $A_0A'_1 A_2A'_3\ldots A_N$
Checksum offload

● Preferred method: protocol agnostic offload
  ○ Any Internet checksum for arbitrary protocols
  ○ Work for encapsulated checksum
  ○ Validate multiple checksums in one packet

● Legacy method: protocol specific offload
  ○ Device performs checksum validation of specific protocol combinations
  ○ E.g. may handle simple TCP/IP but not TCP/IP/GRE/IP
Transmit checksum offload

1. Host OS specifies offset of checksum field and starting offset of checksum coverage in a transmit descriptor
2. Host initializes checksum field to “not” of 1s’ complement sum of pseudo header (or 0xFFFF if no pseudo header)
3. Device performs sum from start offset to end of packet and writes result in checksum field
Multi-TX csum (local csum offload)

1. Offload inner most checksum (i.e. TCP in this ex.)
2. TCPCsum = 0xFFFF ⊕ ~PHsum = ~PHsum
3. GRECsum = A2 ⊕ A3 ⊕ TCPCsum = A2 ⊕ A3 ⊕ PHsum
4. UDPCsum = A1 ⊕ GRECsum = A1 ⊕ A2 ⊕ A3 ⊕ TCPCsum
1. Device provides checksum of packet in RX descriptor (DevCsum)
   \[ Csum = DevCsum \]

2. Process IPv6 header and pull up checksum
   Compute ICsum in CPU, \[ Csum = Csum \oplus \sim ICsum \]

3. Process UDP header (pullup and validate)
   Compute Ssum in CPU, \[ Csum = Csum \oplus \sim SCsum \]
   \[ Csum = Csum \oplus PHsum, \text{ if } Csum == 0xFFFF \text{ then sum valid} \]
Recv offload three checksums

DevCsum

UDPCsum

GRECsum

TCPCsum

IPv4  UDP  GRE  IPv6  TCP  TCP payload

A₀  A₁  A₂  A₃

UDPCsum=DevCsum ⊕ ~A₀ (note A₀ == CS0)
GRECsum=UDPCsum ⊕ ~A₁
TCPCsum=GRECsum ⊕ ~A₂ ⊕ ~A₃
Checksum and GSO/TSO

Number of fields need update per packet?

1. Host gives big packet to stack (GSO) or device (TSO)
2. Big packet segmented into equal sized packets (i.e. MTU size)
3. Headers are replicated; csum, payload length, and IPID precomputed
4. Csum field and start offset validate
5. Seq number, IPID increment per pkt
6. IPv4 delta csum cancel IPID
7. TCP csum cancel payload len
Checksum and GRO

Avoid loss of information
1. Host stack receive packets
2. Validate checksum for each packet
3. Assemble large packets for a flow
4. Avoid loss of information, headers need to match precisely
5. Mark big packet metadata as checksum verified
**GSO/TSO/GRO w/encapsulation**

<table>
<thead>
<tr>
<th>IPv4</th>
<th>UDP</th>
<th>GRE</th>
<th>IPv6</th>
<th>TCP</th>
<th>TCP payload</th>
<th>TCP payload</th>
<th>TCP payload</th>
<th>TCP payload</th>
</tr>
</thead>
</table>

GSO/TSO: LCO to set up outer checksums. Rest of logic same as non-encap.

GRO: Validate each checksum using checksum complete and pullup. Rest of logic same as non-encap.
References

● Linux documentation: checksum offloads

● LCO,GSO_PARTIAL,TSO_MANGLEID, and Why Less is More
  ○ https://legacy.netdevconf.info/1.2/papers/LCO-GSO-Partial-TSO-MangleID.pdf

● UDP encapsulation in Linux
Thank you!