Efficient Layer 7 Search of IP Address Space in LuaJIT/OpenResty

Problem Domain

- Search for IP (client IP) in large set of CIDR blocks
- 10^5 CIDRs (IPv4 only)
- Hot path search (>250k req/s)
- Assume worst case execution (no match)

Problem Domain

- Large CIDR lists
  - Bogons (~4k)
  - Spamhaus, OpenBL (~40k +)

- Top-of-stack implementation
  - Low hanging fruit
  - Get in and get out

- Regular refresh
  - ~ 300s
  - No restarting Nginx workers
  - OpenResty to the rescue!

Problem Domain

- Search space
  - Linear growth prohibitively expensive

- Plenty of memory
  - CPU cycles are the bottleneck

- Goal: proper algorithm/implementation

Existing Implementations

- lua-resty-iputils
  - Battle tested
  - Simple interface
  - Transparent caching of parsed CIDRs/IPs
  - IPv4 only

- libcidr-ffi
  - Minimal binding to libcidr
  - IPv6 support
  - Only compare 2 CIDRs at a time

Existing Implementations

- mediador
  - Semi-transparent reverse proxy/IP library
  - Undocumented (404)
  - No in-line caching

- Nginx access module
  - Static config definition
  - Native C implementation == performant?

lua-resty-iputils

```
bool, err = iputils.binip_in_cidrs(bin_ip, cidrs)
```

```
location /iputils {
  content_by_lua_block {
    local iputils = require "resty.iputils"
    ngx.say(iputils.binip_in_cidrs(ngx.var.binary_remote_addr,
iputils_cidrs))
  }
}
```

lua-resty-iputils

```
# wrk -c 50 -d 10s -t5 http://localhost/iputils
Running 10s test @ http://localhost/iputils
  5 threads and 50 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    53.36ms   72.50ms   1.04s    96.25%
    Req/Sec   235.77     52.17   656.00     87.14%
  11521 requests in 10.01s, 2.11MB read
Requests/sec:   1150.56
Transfer/sec:    215.68KB
```

lua-resty-iputils

```
local bin_ip = 0
for i=1,4 do
  bin_ip = bor(lshift(bin_ip, 8), byte(bin_ip_ngx, i))
end
bin_ip = unsign(bin_ip)

for _,cidr in ipairs(cidrs) do
  if bin_ip >= cidr[1] and bin_ip <= cidr[2] then
    return true
  end
end
return false
```

lua-resty-iputils

```
Distribution of Lua code pure execution time (accumulated in each request, in
microseconds) for 13926 samples:
(min/avg/max: 20/837/2322)
value |-------------------------------------------------- count
    4 |                                                       0
    8 |                                                       0
   16 |                                                       1
   32 |                                                       0
   64 |                                                       0
  128 |                                                       7
  256 |                                                     254
  512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  12830
 1024 |@@@                                                  832
 2048 |                                                       2
 4096 |                                                       0
 8192 |                                                       0
```

lua-resty-iputils

```
Observed 6916 Lua-running samples and ignored 0
unrelated samples.

Compiled: 81% (5640 samples)

C Code (by interpreted Lua): 17% (1239 samples)

Interpreted: 0% (34 samples)

Garbage Collector (not compiled): 0% (3 samples)
```

libcidr-ffi

```
location /libcidr {
 content_by_lua_block {
  local libcidr = require "libcidr-ffi"
  local bin_ip = libcidr_cache:get(ngx.var.remote_addr)
  if not bin_ip then
   bin_ip = libcidr.from_str(ngx.var.remote_addr)
   libcidr_cache:set(ngx.var.remote_addr, bin_ip)
  end
  local ok = false
  for i = 1, len do
   ok = libcidr.contains(libcidr_cache:get(ips_tab[i]), bin_ip)
   if ok then break end
  end
  ngx.say(ok)
 }
}
```

libcidr-ffi

```
# wrk -c 50 -d 10s -t5 http://localhost/libcidr
Running 10s test @ http://localhost/libcidr
  5 threads and 50 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   258.53ms  145.49ms   1.98s    90.93%
    Req/Sec    37.40     19.79   140.00     72.51%
  1615 requests in 10.02s, 302.81KB read
  Socket errors: connect 0, read 0, write 0, timeout 22
Requests/sec:    161.25
Transfer/sec:     30.23KB
```

Nginx Access Module

- Static config unsuitable

- Can we learn from their design?
  - Linear loop! :(

Nginx Access Module

```
for (i = 0; i < alcf->rules->nelts; i++) {
     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                    "access: %08XD %08XD %08XD",
                    addr, rule[i].mask, rule[i].addr);


     if ((addr & rule[i].mask) == rule[i].addr) {
         return ngx_http_access_found(r, rule[i].deny);
     }
 }
```

Nginx Access Module

```
typedef struct {
   in_addr_t    addr;
   in_addr_t    mask;
} ngx_in_cidr_t;
```

Review Takeaways

- Individual search functions are not expensive
- Maintain a tiny call footprint
- “JIT-ability” is not a sole determining factor in usefulness

A New Approach

- Divide and conquer (binary search)

- Reduce the amount of time it takes to find the appropriate CIDR to search
  - Linear searches perform the comparison every time

- Logarithmic time is our friend
  - $\log_2(40000) = 16$

IPv4 Address

- Two portions
  - Network bits
  - Host bits

- CIDR notation indicates the size of the network
  - 192.168.0.0/24
  - 10.0.0.0/8
  - 1.2.3.4/32

IPv4 Address

```
192.168.1.43/24

11111111 11111111 11111111 00000000
11000000 10101000 00000001 00101011
```

- Network/broadcast (bottom/top)

```
11000000 10101000 00000001 00000000
11000000 10101000 00000001 11111111
```

IPv4 CIDR Compare

- 192.168.1.43
- 192.168.2.0/24

11111111 11111111 11111111 00000000

11000000 10101000 00000010 00000000

11000000 10101000 00000001 00101011

11000000 10101000 00000010

11000000 10101000 00000001

IPv4 CIDR Compare

- 192.168.1.43
- 192.168.0.0/23

11111111 11111111 11111110 00000000
11000000 10101000 00000000 00000000
11000000 10101000 00000001 00101011

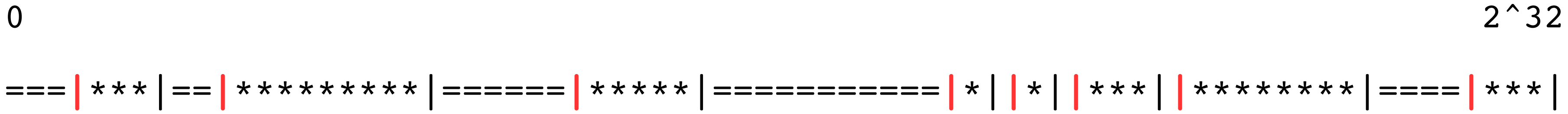
11000000 10101000 0000000
11000000 10101000 0000000

Binary Search

- We want to know if we’re “in” a CIDR range
  - Our integer is higher than the CIDR’s network
  - Our integer is lower than the CIDR’s broadcast
  - Challenge is efficiently finding the “right” CIDR in which to compare

Binary Search

```
0                                                                      2^32
===========================================================================
```



```
0                                                                      2^32
===|***|==|*********|======|*****|===========|*||*||***||********|====|***|
```

Binary Search

- Search for the highest possible CIDR network address (that is lower than our search value)
- Execute a single comparison once we’ve found a candidate CIDR
- Compare the CIDR network value (lowest value in CIDR range) with IP ^ CIDR mask value

Binary Search

```
{
 { “10.0.0.0”, “8” },
 { “172.12.0.0”, “12” },
 { “192.168.0.0”, “24” },
}


{
 { 167772160, 4278190080 },
 { 2886467584, 4293918720 },
 { 3232235520, 4294967040 },
}
```

Binary Search

```
ffi.cdef[[
  typedef uint32_t in_addr_t;
  typedef struct {
    in_addr_t   mask;
    in_addr_t   addr;
  } cidr_t;
]]
```

```
local function bin_search_cidr(ip, cidrs, len)
  local l, r = 0, len - 1

  while l <= r do
    local m = floor((l + r) / 2)

    -- if we're less, bisect to the left
    if ip < cidrs[m].addr then
      r = m - 1

    else -- ge

      -- we're higher than both m and the next cidr, bisect right
      -- note that we may also equal the next cidr, in which case
      -- just bisect again and we'll land in the check below
      if m + 1 <= len - 1 and ip >= cidrs[m + 1].addr then
        l = m + 1

      else -- we're in between, check!
        return usign(band(ip, cidrs[m].mask)) == cidrs[m].addr
      end
    end
  end
  return false
end
```

Limitations and Optimizations

- Binary search requires the array is sorted
  - table.sort?
  - Not with FFI objects!
  - qsort based on CIDR (network) address, then netmask

- Merge adjacent blocks into supersets

- Remove duplicate/subset CIDR blocks

Optimizations – Adjacency Merge

- Form a single CIDR definition from multiple adjacent CIDRs
    - The ranges border each other
    - The sizes of both CIDR masks are equal
    - The resultant CIDR would contain no meaningless bits

Optimizations – Adjacency Merge

`1.2.3.4, 1.2.3.5 → 1.2.3.4/31`

`192.168.0.0/24, 192.168.1.0/24 →`
`192.168.0.0/23`

Optimizations – Adjacency Merge

```
1.2.3.5, 1.2.3.6 → ?

00000001 00000010 00000011 00000101
00000001 00000010 00000011 00000110

11111111 11111111 11111111 11111110
00000001 00000010 00000011 00000101
```

Optimizations – Adjacency Merge

`1.2.3.5, 1.2.3.6, 1.2.3.7, 1.2.3.8 →`

`1.2.3.5, 1.2.3.6/31, 1.2.3.8`

Optimizations – Subset Prune

- Remove duplicate / subset CIDR definitions

- If a CIDR fits entirely into an adjacent CIDR, it is a meaningless definition

Optimizations – Subset Prune

`192.168.0.0/24, 192.168.0.0/25 → 192.168.0.0.24`

`192.168.0.0/24, 192.168.0.0/25, 192.168.0.128/26 → 192.168.0.0.24`

```
location /cidr-bin {
  content_by_lua_block {
    local cidr = require "cidr"
    local bin_ip = cidr_cache:get(ngx.var.remote_addr)
    if not bin_ip then
      bin_ip = cidr.bin_ip(ngx.var.remote_addr)
      cidr_cache:set(ngx.var.remote_addr, bin_ip)
    end
    ngx.say(cidr.bin_search_cidr(bin_ip, cidrs, len))
  }
}
```

```
location /cidr-bin {
  content_by_lua_block {
    local cidr = require "cidr"
    local bin_ip = cidr_cache:get(ngx.var.remote_addr)
    if not bin_ip then
      bin_ip = cidr.bin_ip(ngx.var.remote_addr)
      cidr_cache:set(ngx.var.remote_addr, bin_ip)
    end
    ngx.say(cidr.lin_search_cidr(bin_ip, cidrs, len))
  }
}
```

Performance

- lua-resty-iputils: ~1200 req/s
- libcidr-ffi: ~160 req/s
- CIDR bin search: ...

Performance

```
# wrk -c 50 -d 10s -t5 http://localhost/cidr-bin
Running 10s test @ http://localhost/cidr-bin
  5 threads and 50 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   811.73us    1.03ms  48.56ms   97.45%
    Req/Sec    13.67k     2.17k   30.03k    78.09%
  682942 requests in 10.10s, 125.02MB read
Requests/sec:  67620.94
Transfer/sec:     12.38MB
```

Performance

```
Distribution of Lua code pure execution time (accumulated in each request, in microseconds) for
163519 samples:
(min/avg/max: 11/22/4419)
value |-------------------------------------------------- count
    2 |                                                        0
    4 |                                                        0
    8 |                                                      182
   16 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  150981
   32 |@@@                                                 11271
   64 |                                                      950
  128 |                                                       52
  256 |                                                       22
  512 |                                                       25
 1024 |                                                       22
 2048 |                                                        6
 4096 |                                                        8
 8192 |                                                        0
16384 |                                                        0
```

Execution Time Comparison

# IPs	Binary (us)	Linear (us)
10	4792	2284
50	5239	7996
100	6926	13819
250	5793	30593
500	6113	58013
1000	6859	113552
10000	8008	1139527
40000	8920	3921091

Review / Notes

- Library assumes the IP is a Lua integer
  - Cache integer representation elsewhere

- Gains are exponential
  - “warm-up” scale
  - Less efficient than linear search at ~100 CIDRs
  - Complexity/overhead means small CIDR searches should continue to use other implementations

- No IPv6 support

Questions?