Over at $DAYJOB’s blog I’ve queued up a post that shows how to use our new [opendata[](https://cran.r-project.org/web/packages/ropendata/)](https://cran.r-project.org/web/packages/ropendata/) package to work with our [Open Data](https://opendata.rapid7.com/) portal’s API. I’m not super-sure when it’s going to be posted so keep an RSS reader fixed on <https://blog.rapid7.com/> if you’re interested in seeing it (I may make a small note of it here so it can wind its way into R Weekly & R-bloggers).

The example data used in the post is the public version of what I talked about in [a recent post here](https://rud.is/b/2019/02/03/send-udp-probes-with-payloads-and-receive-process-responses-in-r/), namely the devices discovered exposing the Ubiquity Discovery Protocol.

I’m quite blessed at work since we have virtually all of our icky payload data pre-processed and in parquet map columns in Athena so I don’t really have to do much data wrangling once we’ve fully baked a new study.

The format of the *public data* for the Ubiquiti discovery protocol scan results is a bit different than the base64 encoded data in the previous post in that the payload response is a hex-encoded character string; e.g.

0100009302000a002722bccf9db126fa9a02000a002722bdcf9dc0a80101010006002722bccf9d0a000400006ae40b000c626a732e6572656e696c646f0c00064147352d48500d00104d6f72726f5f446f757261646f5f30330e000102030022584d2e6172373234302e76352e362e332e32383539312e3135313133302e31373439100002e24514000d41697247726964204d35204850

So, every two characters is a byte (e.g. "01" is 0x01).

R has a nice strtoi() function for converting a hex-encoded byte into a raw value but it only works for one byte. We can split a string (like the one above) into a character vector of length 2 hex strings in many ways, one of which is using helper functions from the stringi package:

library(stringi)

library(magrittr) # for %>%

x <- "0100009302000a002722bccf9db126fa9a02000a002722bdcf9dc0a80101010006002722bccf9d0a000400006ae40b000c626a732e6572656e696c646f0c00064147352d48500d00104d6f72726f5f446f757261646f5f30330e000102030022584d2e6172373234302e76352e362e332e32383539312e3135313133302e31373439100002e24514000d41697247726964204d35204850"

stri\_sub(x, seq(1, stri\_length(x), by = 2), length = 2)

## [1] "01" "00" "00" "93" "02" "00" "0a" "00" "27" "22" "bc" "cf" "9d" "b1" "26" "fa" "9a"

## [18] "02" "00" "0a" "00" "27" "22" "bd" "cf" "9d" "c0" "a8" "01" "01" "01" "00" "06" "00"

## [35] "27" "22" "bc" "cf" "9d" "0a" "00" "04" "00" "00" "6a" "e4" "0b" "00" "0c" "62" "6a"

## [52] "73" "2e" "65" "72" "65" "6e" "69" "6c" "64" "6f" "0c" "00" "06" "41" "47" "35" "2d"

## [69] "48" "50" "0d" "00" "10" "4d" "6f" "72" "72" "6f" "5f" "44" "6f" "75" "72" "61" "64"

## [86] "6f" "5f" "30" "33" "0e" "00" "01" "02" "03" "00" "22" "58" "4d" "2e" "61" "72" "37"

## [103] "32" "34" "30" "2e" "76" "35" "2e" "36" "2e" "33" "2e" "32" "38" "35" "39" "31" "2e"

## [120] "31" "35" "31" "31" "33" "30" "2e" "31" "37" "34" "39" "10" "00" "02" "e2" "45" "14"

## [137] "00" "0d" "41" "69" "72" "47" "72" "69" "64" "20" "4d" "35" "20" "48" "50"

We still need to run that through strtoi() and turn it into a raw vector (at least for this use-case):

stri\_sub(x, seq(1, stri\_length(x), by = 2), length = 2) %>%

strtoi(base = 16) %>%

as.raw()

## [1] 01 00 00 93 02 00 0a 00 27 22 bc cf 9d b1 26 fa 9a 02 00 0a 00 27 22 bd cf 9d c0 a8 01

## [30] 01 01 00 06 00 27 22 bc cf 9d 0a 00 04 00 00 6a e4 0b 00 0c 62 6a 73 2e 65 72 65 6e 69

## [59] 6c 64 6f 0c 00 06 41 47 35 2d 48 50 0d 00 10 4d 6f 72 72 6f 5f 44 6f 75 72 61 64 6f 5f

## [88] 30 33 0e 00 01 02 03 00 22 58 4d 2e 61 72 37 32 34 30 2e 76 35 2e 36 2e 33 2e 32 38 35

## [117] 39 31 2e 31 35 31 31 33 30 2e 31 37 34 39 10 00 02 e2 45 14 00 0d 41 69 72 47 72 69 64

## [146] 20 4d 35 20 48 50

On one of my systems, an individual use of that full processing pipeline with the sample string takes about 170μs which is not bad. But, what if we have half a million of them (as was the case with the blog post for work)? I mean, *sure*, it’s only about a minute and a half of processing time (with some variance as each bit of input will be of different lengths), but that’s a painful interactive 1.5 minutes and we still need to wrap that bit of code in a function with some vectorization so it can be used easily.

This is a good example of where the complexity introduced by using a little C++/Rcpp may be warranted, especially since the BH package—which brings us a ton of capabilities from the Boost C++ library—has some handy string utilities, including an boost::algorithm::unhex() function.

Here’s one way to attack the problem in C++/Rcpp within a plain ol’ R session:

library(Rcpp)

cppFunction(depends = "BH", '

List dehexify\_cpp(StringVector input) {

List out(input.size()); // make room for our return value

for (unsigned int i=0; i(input[i])); // do the work

std::copy(h.begin(), h.end(), tmp.begin()); // copy it to our raw vector

out[i] = tmp; // save it to the List

} else {

out[i] = NA\_STRING; // bad input

}

}

return(out);

}

', includes = c('#include ')

)

Now, we have a dehexify\_cpp() function in our environment, so we can use it on any valid R data. Let’s see if we get the same results as the stringi R version:

dehexify\_cpp(x)

## [[1]]

## [1] 01 00 00 93 02 00 0a 00 27 22 bc cf 9d b1 26 fa 9a 02 00 0a 00 27 22 bd cf 9d c0 a8 01

## [30] 01 01 00 06 00 27 22 bc cf 9d 0a 00 04 00 00 6a e4 0b 00 0c 62 6a 73 2e 65 72 65 6e 69

## [59] 6c 64 6f 0c 00 06 41 47 35 2d 48 50 0d 00 10 4d 6f 72 72 6f 5f 44 6f 75 72 61 64 6f 5f

## [88] 30 33 0e 00 01 02 03 00 22 58 4d 2e 61 72 37 32 34 30 2e 76 35 2e 36 2e 33 2e 32 38 35

## [117] 39 31 2e 31 35 31 31 33 30 2e 31 37 34 39 10 00 02 e2 45 14 00 0d 41 69 72 47 72 69 64

## [146] 20 4d 35 20 48 50

Apart from it being a list (since we took care of vectorization at the same time) it is, indeed, the same data.

With that tiny bit of fairly straightforward Rcpp/C++ code we get a substantially faster execution time of around 4μs. Yep, that’s not a typo: *four* microseconds.

We’ll give it a real world test with the payload data from work:

# This assumes you have a "~/Data" directory. Put it somewhere

# else if you don't have a "~/Data" directory.

if (!file.exists("~/Data/dehexify-sample-data.txt.gz")) {

download.file(

url = "<https://rud.is/dl/dehexify-sample-data.txt.gz>",

destfile = "~/Data/dehexify-sample-data.txt.gz"

)

}

char\_hex\_lines <- readr::read\_lines("~/Data/dehexify-sample-data.txt.gz")

length(char\_hex\_lines)

## [1] 501926

res <- dehexify\_cpp(char\_hex\_lines)

That took just over a second to run on my main development system. But, did it really work? I chose index 998 at random so let’s poke at it with the tool from the other blog post:

udpprobe::parse\_ubnt\_discovery\_response(res[[998]])

## [Model: N5N; Firmware: XW.ar934x.v5.5.9.21734.140403.1801; Uptime: 13.1 (hrs)

Aye, it did, indeed, work.

**FIN**

It’s still early in 2019 and if you haven’t settled on any resolutions yet or want to substitute out one that isn’t working so well (who wants to drive to the gym anyway?) with another, perhaps add “experiment with Rcpp” to the list since a tiny dose of it can go a very long way into speeding up some tasks.