21:11 Windrose Fingerprinting of Code Architecture

by EVM

Often we come across a firmware from a device that we don't have in hand, and don't know anything about beyond pictures or sales glossies on a vendor website. We'd like to be able to load this firmware into a disassembler and analyze it anyway.

ELF firmware files will happily tell you and your disasssembler the CPU architecture, but what do we do when analyzing a flat binary firmware file? We need a method to determine the architecture by comparing the file to previous samples from known architectures.

Each processor architecture has a unique byte histogram fingerprint, which others have described previously. This is because in machine code some types of opcodes are used more frequently than others (Register/memory move, comparison, jump/branch/call are usually the most common.) This gives each architecture a unique balance of bytes reflecting the designer's choice of representation of common and uncommon opcodes.

What I'm adding to the toolbox here is the concept of visualizing byte histograms as a windrose diagram. Byte histograms can be compared using a chi-squared test, but windrose diagrams may allow for a more-nuanced, visual comparison.

The following diagrams were generated from samples, mostly Linux kernels and Busybox binaries, and the occasional random large firmware file. Linux kernels and Busybox binaries work well because they are very large and contain a mix of lots of different kinds of code.

Here is a Python script that outputs a windrose diagram for a sample that you can compare against the fingerprints shown. This code bins the bytes in groups of four for more readable diagrams, and ignores bytes `0x00`, `0x40`, `0x80` and `0xC0` (to avoid over-representing top address bytes). Note that for best results you need to only map the text section of a binary, and remove any padding. Normally in a flat firmware binary the text section appears before the data section, and depending upon where you make the cut, your mileage may significantly vary on very small binaries.

As an example, Figure 12 presents windrose diagrams from the `.text` section of two 32-bit MIPS binaries. These are the first two MIPS binaries in the ALLSTAR dataset[31] whose `.text` section is greater than 64KB, `7kaa` from Debian's `7kaa` package, and `jmdlx` from Debian's `aajm` package. Notice their largest three spikes (the `0x00`, `0x20`, and `0x8C` bins) match the 32-bit MIPS fingerprint well. The double spike (`0x20` and `0x24` bins) appears in all three prints. `jmdlx` has a shorter spike at `0x00`, and a longer spike at `0x08`, but we can still easily see that its best match is 32-bit MIPS.

[31] `https://allstar.jhuapl.edu`

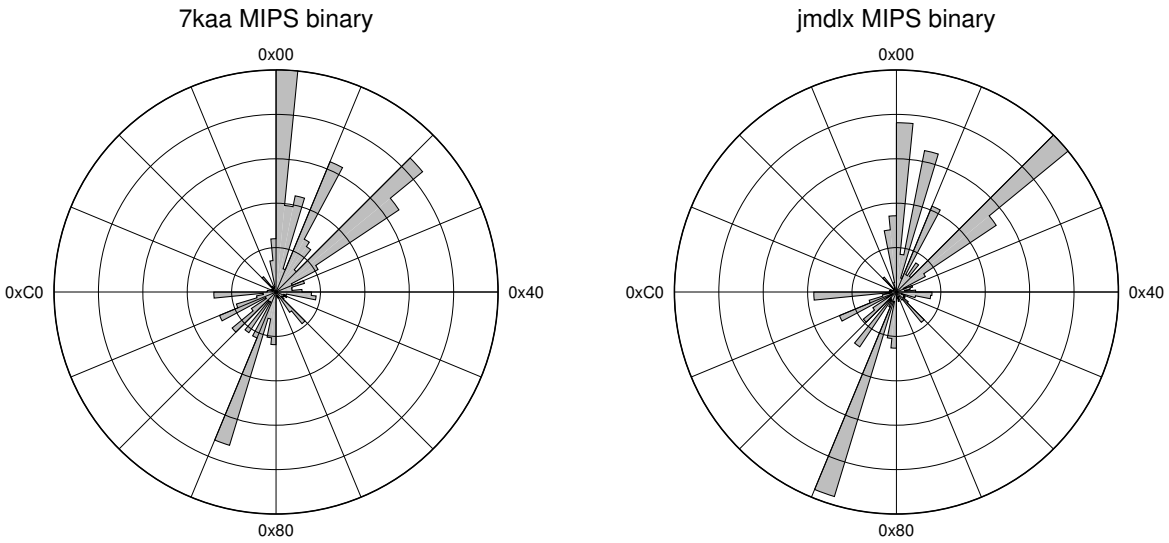
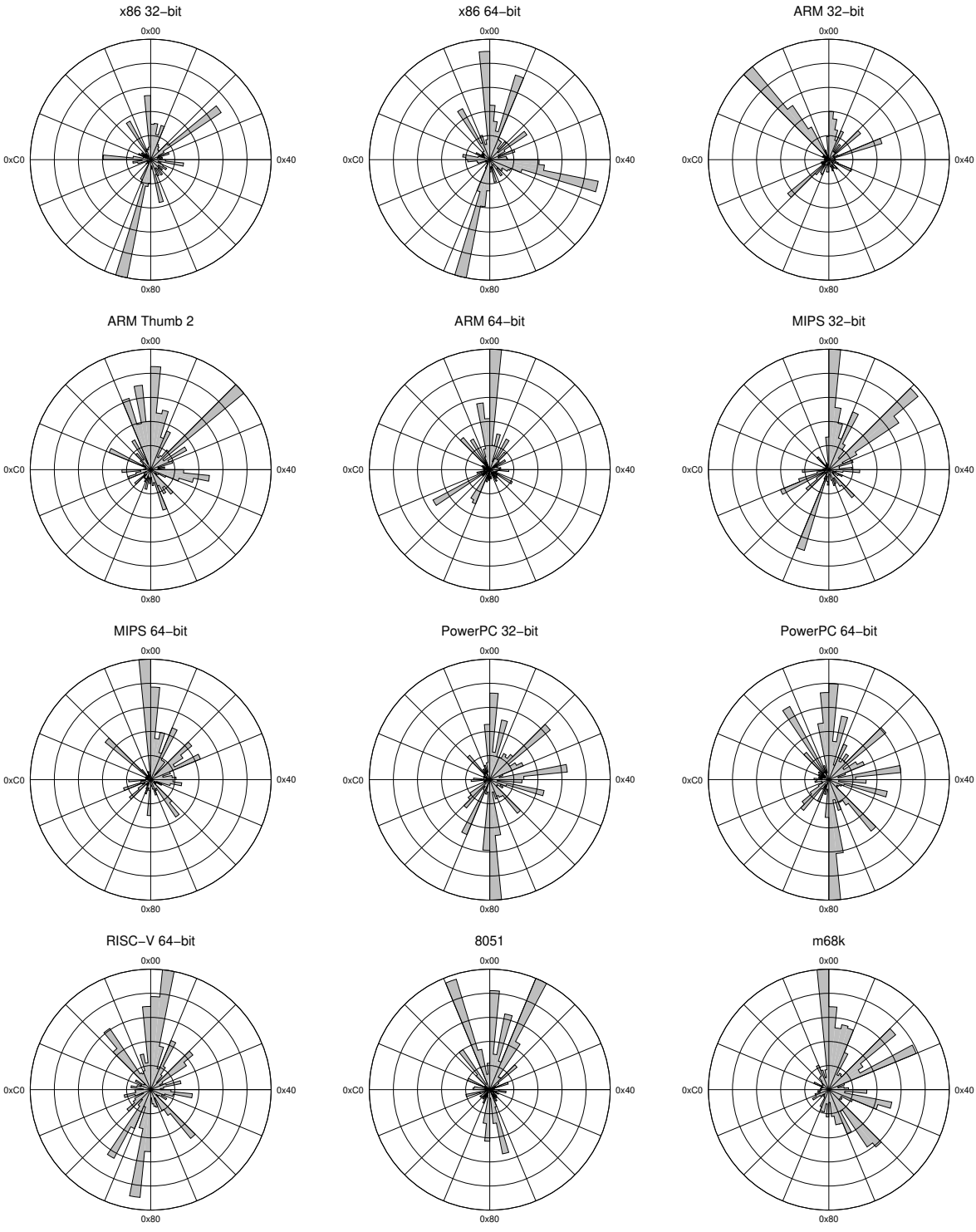
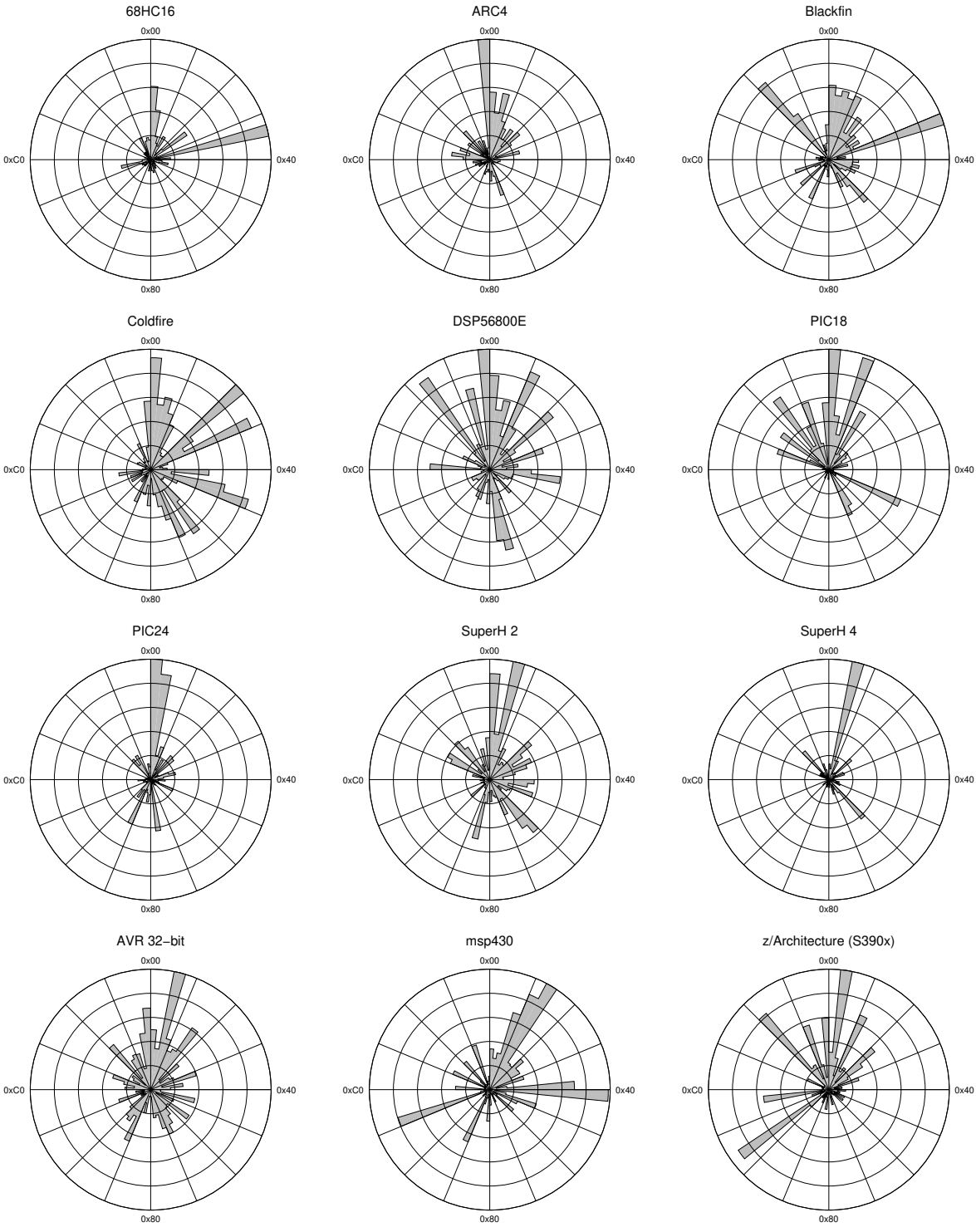


Figure 12: Fingerprints of two sample MIPS executables.

```
#! /usr/bin/python
import sys
import struct

fname = sys.argv[1]

bytes=[]
entries=[]
total=0

for i in xrange(0,256):
  bytes.append(0)

with open(fname,'rb') as f:
  while True:
    b=f.read(1)
    if b=="":
  break
    bint = struct.unpack('<B',b)[0]
    bytes[bint]+=1

for i in xrange(0,256,4):
  entry=0
  for j in xrange(0,4):
    if (((i+j) % 0x40) != 0):
      entry+=int(bytes[i+j])
  entries.append(entry)
  total+=entry

for i in xrange(0,0x40):
  print "%f,%f" % (100*entries[i]/1.0/max(entries), i*360/1.0/0x40)
```


x86 32−bit
x86 64−bit
ARM 32−bit
ARM Thumb 2
ARM 64−bit
MIPS 32−bit
MIPS 64−bit
PowerPC 32−bit
PowerPC 64−bit
RISC−V 64−bit
8051
m68k
0x00
0x40
0x80
0xC0


68HC16
ARC4
Blackfin
Coldfire
DSP56800E
PIC18
PIC24
SuperH 2
SuperH 4
AVR 32−bit
msp430
z/Architecture (S390x)
0x00
0x40
0x80
0xC0