# 16:11 Rescuing Orphans and their Parents with Rules of Thumb2

*by Travis Goodspeed KK4VCZ,*
*concerning Binary Ninja and the Tytera MD380.*

Howdy y'all,

It's a common problem when reverse engineering firmware that an auto-analyzer will recognize only a small fraction of functions, leaving the majority unrecognized because they are only reached through function pointers. In this brief article, I'll show you how to extend Binary Ninja to recognize nearly all functions in a threaded MicroC-OS/II firmware image for ARM Cortex M4. This isn't a polished plugin or anything as fancy as the internal functions of Binary Ninja; rather, it's a story of how to kick a high brow tool with some low level hints to efficiently carve up a target image.

We'll begin with the necessary chore of loading our image to the right base address and kicking off the auto-analyzer against the interrupt vector handlers. That will give us `main()` and its direct children, but the auto-analyzer will predictably choke when it hits the function that kicks off the threads, which are passed as function pointers.

Next, we'll take some quick theories about the compiler's behavior, test them for correctness, and then use these rules of thumb to reverse engineer real binaries. These rules won't be true for every *possible* binary, but they happen to be true for Clang and GCC, the only compilers that matter.

## Loading Firmware

Binary Ninja has excellent loaders for PE and ELF files, but raw firmware images require either conversion or a custom loader script. You can find a full loader script in the `md380tools` repository,[34] but an abbreviated version is shown in Figure 5.

The loader will open the firmware image, as well as blank regions for SRAM and TCRAM. For full reverse engineering, you will likely want to also load an extracted core dump of a live device into SRAM.



## Detecting Orphaned Function Calls

Unfortunately, this loader script will only identify 227 functions out of more than a thousand.[35]

```
>>> len(bv.functions)
227
```

The majority of functions are lost because they are only called from within threads, and the threads are initialized through function pointers that the autoanalyzer is unable to recognize. Given a single image to reverse engineer, we might take the time to hunt down the `init_threads()` function and manually defined each thread entry point as a function, but that quickly becomes tedious. Instead, let's script the auto-analyzer to identify *parents* from known *child* functions, rather than just children from known parent functions.

Thumb2 uses a `bl` instruction, branch and link, to call one function from another. This instruction is 32 bits long instead of the usual 16, and in the Thumb1 instruction set was actually two distinct 16-bit instructions. To redirect function calls, the re-linking script of MD380Tools searches for every 32-bit word which, when interpreted as a `bl`, calls the function to be hooked; it then overwrites those words with `bl` instructions that call the new function's address.

[34] `git clone https://github.com/travisgoodspeed/md380tools`
[35] Hit the backquote button to show the python console, just a like one o' them vidya games.

```
class MD380View(BinaryView):
    """This class implements a view of the loaded firmware, for any image
    that might be a firmware image for the MD380 or related radios loaded
    to 0x0800C000.
    """

    def __init__(self, data):
        BinaryView.__init__(self, file_metadata = data.file, parent_view = data)
        self.raw = data

    @classmethod
    def is_valid_for_data(self, data):
        hdr = data.read(0, 0x160)
        if len(hdr) < 0x160 or len(hdr)>0x100000:
            return False
        if ord(hdr[0x3]) != 0x20:
            # First word is the initial stack pointer, must be in SRAM around 0x20000000.
            return False
        if ord(hdr[0x7]) != 0x08:
            # Second word is the reset vector, must be in Flash around 0x08000000.
            return False
        return True

    def init_common(self):
        self.platform = Architecture["thumb2"].standalone_platform
        self.hdr = self.raw.read(0, 0x100001)

    def init_thumb2(self, adr=0x08000000):
        try:
            self.init_common()
            self.thumb2_offset = 0
            self.arm_entry_addr = struct.unpack("<L", self.hdr[0x4:0x8])[0]
            self.thumb2_load_addr = adr #struct.unpack("<L", self.hdr[0x38:0x3C])[0]
            self.thumb2_size = len(self.hdr);

            codeflags=SegmentFlag.SegmentReadable | SegmentFlag.SegmentExecutable;
            ramflags=codeflags | SegmentFlag.SegmentWritable;

            # Add segment for SRAM, not backed by file contents
            self.add_auto_segment(0x20000000, 0x20000, #128K at address 0x20000000.
                                  0, 0, ramflags)
            # Add segment for TCRAM, not backed by file contents
            self.add_auto_segment(0x10000000, 0x10000, #64K at address 0x10000000.
                                  0, 0, ramflags)
            #Add a segment for this Flash application.
            self.add_auto_segment(self.thumb2_load_addr, self.thumb2_size,
                                  self.thumb2_offset, self.thumb2_size,
                                  codeflags)

            #Define the RESET vector entry point.
            self.define_auto_symbol(Symbol(SymbolType.FunctionSymbol,
                                           self.arm_entry_addr&~1, "RESET"))
            self.add_entry_point(self.arm_entry_addr&~1)

            #Define other entries of the Interrupt Vector Table (IVT)
            for ivtindex in range(8,0x184+4,4):
                ivector=struct.unpack("<L", self.hdr[ivtindex:ivtindex+4])[0]
                if ivector>0:
                    #Create the symbol, then the entry point.
                    self.define_auto_symbol(Symbol(SymbolType.FunctionSymbol,
                                                   ivector&~1, "vec_%x"%ivector))
                    self.add_function(ivector&~1);
            return True
        except:
            log_error(traceback.format_exc())
            return False
    def perform_is_executable(self):
        return True

    def perform_get_entry_point(self):
        return self.arm_entry_addr

class MD380AppView(MD380View):
    """MD380 Application loaded to 0x0800C000."""
    name = "MD380"
    long_name = "MD380 Flash Application"

    def init(self):
        return self.init_thumb2(0x0800c000)

MD380AppView.register()
```

Figure 5. MD380 Firmware Loader for Binary Ninja

To detect orphaned function calls, which exist in the binary but have not been declared as code functions, we can search backward from known function entry points, just as the re-linker in MD380Tools searches backward to redirection function calls!

Let's begin with the code that calculates a `bl` instruction from a source address to a target. Notice how each 16-bit word of the result has an `F` for its most significant nybble. MD380Tools uses this same trick to ignore function calls when comparing functions to migrate symbols between target firmware revisions.

```
def calcbl(adr, target):
    """Calculates the Thumb code to branch
       to a target."""
    offset = target - adr
    offset -= 4  # PC points to next ins.
    offset = (offset >> 1)  # LSBit ignored

    # Hi address setter, but at lower adr.
    hi = 0xF000 | ((offset&0x3ff800)>>11)
    # Low adr setter goes next.
    lo = 0xF800 | (offset & 0x7ff)

    word = ((lo << 16) | hi)
    return word
```

This handy little function let us compare every 32-bit word in memory to the 32-bit word that would be a `bl` from that address to our target function. This works fine in Python because a typical Thumb2 firmware image is no more than a megabyte; we don't need to write a native plugin.

So for each word, we calculate a branch from that address to our function entry point, and then by comparison we have found all of the `bl` calls to that function.

Knowing the source of a `bl` branch, we can then check to see if it is in a function by asking Binary Ninja for its basic block. If the basic block is `None`, then the `bl` instruction is outside of a function, and we've found an orphaned call.

```
prevfuncadr=
  v.get_previous_function_start_before(
    start+i)
prevfunc=
  v.get_function_at(prevfuncadr)
basicblock=
  prevfunc.get_basic_block_at(start+i)
```

To catch data references to executable code, we also look for data words with the function's entry address, which will catch things like interrupt vectors and thread handlers, whose addresses are in a constant pool, passed as a parameter to the function that kicks of a new thread in the scheduler.

See Figure 6 for a quick and dirty plugin that identifies orphaned function calls to currently selected function. It will print the addresses of all orphaned called (those not in a known function) and also data references, which are terribly handy for recognizing the sources of callback functions.[36]

## Detecting Starts of Functions

Now that we can identify orphaned function calls, that is, `bl` instructions calling known functions from outside of any known function, it would be nice to identify where the function call's parent begins. That way, we could auto-analyze the firmware image to identify all *parents* of known functions, letting Binary Ninja's own autoanalyzer identify the other children of those parents on its own.

With a little luck, we can could crawl from a few I/O functions all the way up to the UI code, then all the way back down to leaf functions, and back to all the code that calls them. This is especially important for firmware with an RTOS, as the thread scheduling functions confuse an auto-analyzer that only recognizes child functions.

First, we need to know what functions begin with. To do that, we'll just write a quick plugin that prints the beginning of each function. I ran this on a project with known symbols, to get a feel for how the compiler produces functions.

```
#Exports function prefixes to a file.
def exportfunctionpreambles(view):
  for fun in view.functions:
    print "%08x: %s  %s" % (fun.start,
      hexdump(view.read(fun.start,4)),
      view.get_disassembly(fun.start,
              Architecture["thumb2"]))

PluginCommand.register(
    "Export Function Preambles",
    "Prints four bytes for each function.",
    exportfunctionpreambles);
```

[36] As I write this, Binary Ninja seems to only recognize data references which are themselves used in a known function or that function's constant pool. It's handy to manually search beyond that range, especially when a core dump of RAM is available.

```
def thumb2findorphanedcalls(view, fun):
    if fun.arch.name!="thumb2":
        print "Sorry, this only works for thumb2, not for %s." % fun.arch.name;
        return;
    print "Searching for calls to %s at 0x%x." % (fun.name,fun.start);

    #Fix these to match the image.
    start=view.start;
    count=None;

    #If we're lucky, the branch is in a segment, which we can use as a
    #range.
    for seg in view.segments:
        if seg.start<fun.start and seg.end>fun.start:
            count=seg.end-start;
    if count==None:
        print "Abandoned search for orphaned calls to %s as out of range." % fun.name;

    print "Searching from 0x%08x to 0x%08x." % (start,start+count)
    data=view.read(start,count);
    count=len(data);

    for i in xrange(0,count-2,2):
        word=(ord(data[i])
              |(ord(data[i+1])<<8)
              |(ord(data[i+2])<<16)
              |(ord(data[i+3])<<24));
        if word==calcbl(start+i, fun.start):
            prevfuncadr=view.get_previous_function_start_before(start+i);
            prevfunc=view.get_function_at(prevfuncadr)
            basicblock=prevfunc.get_basic_block_at(start+i);
            if basicblock!=None:
                #We're in a function.
                print "%08x: %s" % (start+i,prevfunc.name);
                if prevfunc.start!=beginningofthumb2function(view,start+i):
                    print "ERROR: Does the function start at %x or %x?" % (
                        prevfunc.start,
                        beginningofthumb2function(view,start+i));
            else:
                #We're not in a function.
                print "%08x: ORPHANED!" % (start+i);
        elif word==((fun.start)|1):
            print "%08x: DATA!" % (start+i);


PluginCommand.register_for_function(
    "Find Orphaned Calls",
    "Finds orphaned thumb2 calls to this function.",
    thumb2findorphanedcalls);
```

Figure 6. This finds all calls from unregistered functions to the selected function.

Running this script shows us that functions begin with a number of byte pairs. As these convert to opcodes, let's play with the most common ones in assembly language!

`fff7 febf` is an unconditional branch-to-self, or an infinite while loop. You'll find this at all of the unused interrupt vector handlers, and as it has no children, we can ignore it for the purposes of working backward to a function definition, as it never calls another function. `7047` is `bx lr`, which simply returns to the calling function. Again, it has no child functions, so we can ignore it.

`80b5` is `push {r7, lr}`, which stores the link register so that it can call a child function. Similarly, `10b5` pushes `r4` and `lr` so that it can call a child function. `f8b5` pushes `r3`, `r4`, `r5`, `r6`, `r7`, and `lr`. In fact, any function that calls children will begin by pushing the link register, and functions generated by a C compiler seem to never push `lr` anywhere except at the beginning.

So we can write a quick little function that walks backward from any `bl` instruction that we find outside of known functions until it finds the entry point. We can also test this routine whenever we have a known function entry point, as a sanity check that we aren't screwing up the calculations somehow.

```
#Identifies the entry point of a function,
#given an address.
def beginningofthumb2function(view, adr):
  """Identifies the start of the thumb2
     function that include adr."""
  print "Searching from %x." % adr

  a=adr;
  while a>view.start:
    dis=view.get_disassembly(a,
                Architecture["thumb2"])
    if "push" in dis:
      if "lr" in dis:
        print "Found entry at 0x%08x"%a;
        return a;
    a-=2;

PluginCommand.register_for_address(
  "Find Beginning of Function",
  "Find the beginning of a thumb2 fn.",
  beginningofthumb2function);
```

This seems to work well enough for a few examples, but we ought to check that it works for every `bl` address. After thorough testing it seems that this is almost always accurate, with rare exceptions, such as `noreturn` functions, that we'll discuss later in this paper. Happily, these exceptions aren't much of a problem, because the false positive in these cases is still the starting address of *some* function, confusing our plugin but not ruining our database with unreliable entries.

– — — – — — — – — — –

So now that we can both identify orphaned calls from parent functions to a child and the backward reference from a child to its parent, let's write a routine that registers all parents within Binary Ninja.

```
#We're not in a function.
print "%08x: ORPHANED!" % (start+i);
#Register that function
adr=beginningofthumb2function(view,start+i);
view.define_auto_symbol(
      Symbol(SymbolType.FunctionSymbol,
             adr, "fun_%x"%adr))
view.add_function(adr);
```

And if we can do this for one function, why not automate doing it for all known functions, to try and crawl the database for every unregistered function in a few passes? A plugin to register parents of one function is shown in Figure 6, and it can easily be looped for all functions.

Unfortunately, after running this naive implementation for seven minutes, only one hundred new functions are identifies; a second run takes twenty minutes, resulting in just a couple hundred more. That is way too damned slow, so we'll need to clean it up a bit. The next sections cover those improvements.

## Better in Big-O

We are scanning all bytes for each known function, when we ought to be scanning for all potential calls and then white-listing the ones that are known to be within functions. To fix that, we need to generate quick functions that will identify potential `bl` instructions and then check to see if their targets are in the known function database. (Again, we ignore unknown targets because they might be false positives.)

Recognizing a `bl` instruction is as easy as checking that each half of the 32-bit word begins with an `F`.

```
def isbl(word):
    """Returns true if the word might be
       a BL instruction."""
    return (word&0xF000F000)==0xF000F000;
```

We can then decode the absolute target of that relative branch by inverting the `calcbl()` function from page 54.

```
def decodebl(adr, word):
    """Decodes a Thumb BL instruction its
       value and address."""

    #Hi and Lo refer to adr components.
    #The Hi word comes first.
    hi=word&0xFFFF;
    lo=(word&0xFFFF0000)>>16

    #Decode the word.
    rhi=(hi&0x0FFF)<<11
    rlo=(lo&0x7FF)
    recovered=rhi|rlo;

    #Sign-extend backward references.
    if (recovered&0x00200000):
        recovered|=0xFFC00000;

    #Apply the offset and strip overflow
    offset=4+(recovered<<1);
    return (offset+adr)&0xFFFFFFFF;
```

With this, we can now efficiently identify the targets of all potential calls, adding them to the function database if they both (1) are the target of a `bl` and (2) begin by pushing the link register to the stack. This finds sixteen hundred functions in my target, in the blink of an eye and before looking at any parents.

Then, on a second pass, we can register three hundred parents that are not yet known after the first pass. This stage is effective, finding nearly all unknown functions that return, but it takes a lot longer.

```
>>> len(bv.functions)
1913
```

## Patriarchs are Slow as Dirt

So why can the plugin now identify children so quickly, while still slowing to molasses when identifying parents? The reason is not the parents themselves, but the false negatives for the *patriarch* functions, those that don't push the link register at their beginning because they never use it to return.

For every call from a function that doesn't return, all 568 calls in my image, our tool is now wasting some time to fail in finding the entry point of every outbound function call.

But rather than the quick fix, which would be to speed up these false calls by pre-computing their failure through a ranged lookup table, we can use them as an oracle to identify the patriarch functions which never return and have no direct parents. They should each appear in localized clumps, and each of these clumps ought to be a single patriarch function. Rather than the 568 outbound calls, we'll then only be dealing with a few not-quite-identified functions, eleven to be precise.

These eleven functions can then be manually investigated, or ignored if there's no cause to hook them.

```
>>> len(bv.functions)
1924
```

– — — –  — — —  — — — –

This paper has stuck to the Thumb2 instruction set, without making use of Binary Ninja's excellent intermediate representations or other advanced features. This makes it far easier to write the plugin, but limits portability to other architectures, which will violate the convenient rules that we've found for this one. In an ideal world we'd do everything in the intermediate language, and in a cruel world we'd do all of our analysis in the local machine language, but perhaps there's a proper middle ground, one where short-lived scripts provide hints to a well-engineered back-end, so that we can all quickly tear apart target binaries and learn what these infernal machines are really thinking?

You should also be sure to look at the IDA Python Embedded Toolkit by Maddie Stone, whose Recon 2017 talk helped inspire these examples.[37]

73 from Barcelona,
–Travis

[37] `git clone https://github.com/maddiestone/IDAPythonEmbeddedToolkit`