21:05 Symgrate: A Web API for Thumb2 Symbol Recovery

by Travis Goodspeed and EVM

Hey folks!

Today we'd like to share with you our nifty little summer project, a publicly accessible server for recovering Thumb2 symbols. You send us the first 18 bytes of a function as a hex-encoded string, and if that function exists in our collection of hundreds of thousands of embedded SDK libraries, we'll give you back the function's name and the library's filename in easily parsed JSON. Client plugins for most interactive disassemblers have already been written.

The first popular symbol recovery tool was IDA Pro's FLIRT technology, which debuted in 1996. FLIRT matches are performed by exact equality between the first 32 bytes of a function, except for those bytes which a linker would relocate. These are stored in a tree structure, to take advantage of overlaps between similar functions to minimize the file size.

FLIRT does handle some of the stranger linking choices of X86, but it does not try to solve the general problem of false positives and collisions that arise from related functions having identical signatures despite different behavior. It won't be putting us reverse engineers out of business anytime soon!

In this article, we'll be implementing a function matcher similar to FLIRT, in that we'll produce blinded signatures of functions which demand that most of the code exactly match, while forgiving differences in the bytes that will be adjusted during linking. We'll implement this both as C code that compares functions within its own address space, and as a PostgreSQL table that can be queried to quickly recover function names.

Thumb2 Instruction Blinding

Before we can build a database, or even compare two functions locally, we need to learn a little bit about the Thumb2 instruction set. We'll do this to create a sort of optimized disassembler, whose only job is to blind out the bytes that don't matter.

Thumb2 is the denser of two instruction sets in 32-bit ARM, and it's the one that's most commonly found in embedded systems. Instructions are either one or two 16-bit instruction words in length; unlike the original Thumb and MIPS16, the 32-bit wide instructions cannot be treated as two independent 16-bit wide instructions.

Thumb2 code uses relative addressing for branches for reasons of efficiency, but it has the nice side effect of making much code accidentally position independent.

This applies to branches and function calls, but not to explicit pointers, such as immediate values. Those are stored in something called a constant pool, which is a group of 32-bit constants that are placed after the `ret` instruction at the end of a function and before the entry point of the next function. These constant pools exist because Thumb2 instructions are too short to include long immediate values, so rather than include these values inside of the instruction, they are referenced by a PC-relative offset to the pool.

So a Thumb2 linker is mostly adjusting (1) relative calls between functions, and (2) absolute addresses held in literal pools at the end of a function. Relative branches within a function aren't adjusted, because they remain the same wherever that function might be loaded. Our basic strategy will be to count from the beginning of a function, enforcing that a minimum number of instructions are *either* identical *or* function calls. And those absolute addresses which might change in the constant pool? Those we don't worry about, because they come late

	hw1																hw2															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
General format	1	1	1	1	0												1															
Branch	1	1	1	1	0	S	offset[21:12]										1	0	J1	1	J2	offset[11:1]										
Branch with link	1	1	1	1	0	S	offset[21:12]										1	1	J1	1	J2	offset[11:1]										
Branch with link, change to ARM	1	1	1	1	0	S	offset[21:12]										1	1	J1	0	J2	offset[11:2]										0
Reserved	1	1	1	1	0												1	1		0												1
Conditional branch	1	1	1	1	0	S	cond				offset[17:12]						1	0	J1	0	J2	offset[11:1]										
Secure Monitor Interrupt	1	1	1	1	0	1	1	1	1	1	1	imm[3:0]					1	0	0	0	imm[11:4]									imm[15:12]		
Move to status from register	1	1	1	1	0	0	1	1	1	0	0	R	Rn				1	0	SBZ	0	field_mask				SBZ							
Change processor state (imod, M != 00,0)	1	1	1	1	0	0	1	1	1	0	1	0	SBO				1	0	SBZ	0	SBZ	imod		M	A		I		F	mode		
No operation, hints	1	1	1	1	0	0	1	1	1	0	1	0	SBO				1	0	SBZ	0	SBZ	0	0	0	hint							
Special control operations	1	1	1	1	0	0	1	1	1	0	1	1	SBO				1	0	SBZ	0	SBO				OP				option			
Branch, Change to Java	1	1	1	1	0	0	1	1	1	1	0	0	Rn				1	0	SBZ	0	SBO				SBZ							
Exception return	1	1	1	1	0	0	1	1	1	1	0	1	(1)	(1)	(1)	(0)	1	0	SBZ	0	SBO				imm8							
Move to register from status	1	1	1	1	0	0	1	1	1	1	1	R	SBO				1	0	SBZ	0	Rd				SBZ							
RESERVED	1	1	1	1	0	1	1	1	1	1	1	1					1	0	1	0												
Permanently **UNDEFINED**	1	1	1	1	0	1	1	1	1	1	1	1					1	0	1	0					1	1	1	1				

Subset of Thumb2 instruction formats from ARM DDI 0308D.
Note well that branches begin with 1111.

in the function, after enough instructions that we'll have a match or not.

The machine language format for branches are described in Section 3.3.6 of the Thumb2 Supplement Reference Manual, ARM DDI 0308D, reproduced here on page 12. J1 and J2 are inverted by the sign-extension bit for reasons of backward compatibility, being 1 when the branch is short. From this table, we can see that a Branch with Link (`bl`) instruction always begins with `F` in its first word, and will also begin with an `F` in its second word except when the target is very far away.

Comparing Instructions from C

If you need to quickly compare short functions for similarity in C, where `src16()` and `dst16` grab 16-bit words from your source and destination address spaces, you might do something like the following, counting 16-bit words which are either exactly equal or are short branches.

```
//! How similar are two functions?
int scorematch(int sadr, int dadr){
  int i=0;

  // Comparing half-words.
  do{
    i+=2;
  }while(
    (
     //Halfwords exactly agree
     src16(sadr+i)==dst16(dadr+i)

     //or halfwords might both be a BL.
     || ( (src16(sadr+i)&0xF000)==0xF000 &&
          (dst16(dadr+i)&0xF000)==0xF000 )
    )
    //stop after a while.
    && i<1024
  );

  return i;
}
```

This scruffy little example is missing range checks, and it will fail to recognize the second word of longer branches, when J1 or J2 might be zero, but it is brutally effective at moving symbols between minor revisions of small firmware images.

We used this code, complete with an embarrassing bug or two, in the MD380Tools project for years. See PoC||GTFO 10:8 and 13:5.

Comparing Instructions from SQL

Having some C code that can quickly compare one function to another is great for porting symbols from one executable to another, but we'd rather have a giant database of functions, on a central server, than any friend or stranger can query freely when useful. For this, we needed to convert our rag tag algorithm into one that was just as scruffy, but could be expressed in terms of SQL for convenient querying.

We decided to implement this as a string that is wildcarded in the style of a SQL `LIKE` clause, ASCII-armored and with two underscores (`__`) to replace any byte which might be changed by the linker.

Our table schema is roughly like this,

```
drop table if exists functions;
create table functions(
  id serial primary key,
  arch varchar(10) not null,
  -- C++ names can be very long.
  name varchar(2048) not null,
  filename varchar(2048) not null,
  raw varchar(100) not null,
  blinded varchar(100) not null,
  unique(blinded) --saves pruning later
);
```

SQL Optimizations

When this scheme was mentioned in passing to a mainframe old-timer by the name of Jim, he panicked! "Why in hell are you traversing every table on every query?" We're not, of course, as that would be much too slow.

The naïve version of this, the one that scared Jim so much, is easily read but even after indexing it is rather slow.

```
-- 70ms. Slow and naive, but easy to read.
select name from functions
  where $1 like blinded;
```

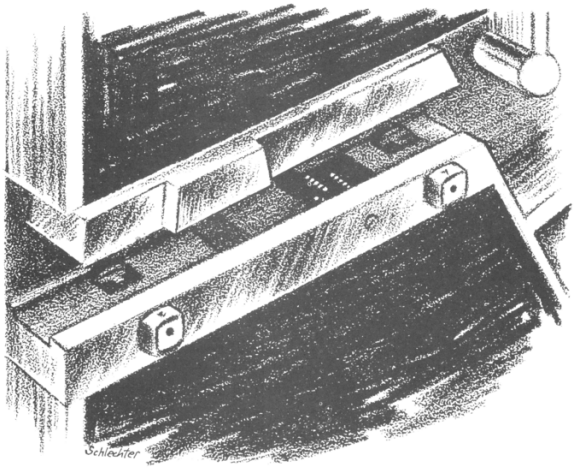
If we ask Postgres to `explain analyze` this query, it takes nearly 70ms because every row of our functions table must be scanned directly, and even split into parallel threads that's a lot of overhead. As our database grows, the overhead will only get worse.

We can speed things up a little more by doing the barest minimum of parsing on the start of the string. See how `10b50446` (`0xb510 0x4604`) does not begin with an `f`, and is not a branch function that might be wild-carded in our database by the

```
% curl -X POST -d 8000beef=02780b78012a28bf9a42f5d16de9044540ea \
  https://symgrate.com/jfns | jq
{
  "8000beef": {
    "Name": "strcmp",
    "Filename": "iccv9cortex/GnuARM/arm-none-eabi/lib/thumb/v7e-m+dp/hard/libg.a"
  }
}
```

instruction format on page 12? We can add a little piece to the `where` clause, such that the first eight characters must *exactly* match our unknown function. With this addition, the `like` operation will only be calculated against a small subset of the total table.

```
-- 7ms when first two are not branches.
select name from functions where
    (    substr($1,3,1)='f'
      or substr($1,7,1)='f'
      or substr($1,1,8)=substr(blinded,1,8)
    ) and $1 like blinded;
```



300 Gigs of Object Files

The one big shortcoming of this technique is that while it is rather robust against changes made by the linker, it is terribly fragile to changes in compiler optimization.

We counter this by recognizing that much device firmware is compiled from static libraries distributed with compiler toolchains as part of an Integrated Development Environment (IDE) from the chip vendor. We've taken to collecting every one of these we can publicly find online, buying the really old ones on eBay.

All told, we're now well above 300GB of `.a`, `.o` and other object files, which are crunched into a SQL table by a bunch of Binary Ninja scripts running in parallel. While Binja offers excellent scripting support that is a joy to use, we're ashamed to admit that we use it here only as a glorified ELF parser, to quickly give us the function prefixes and names.

All told, we have a couple hundred different IDE versions that supply us with 41 unique fingerprints for `strcpy`, 43 for `strlen`, 134 for `strncpy` and 50 for `sprintf`.

Clients and Server

Our server is written in Golang, presenting a few simple API pages that return `json` describing every function that matches our collection. For those in a hurry, results can also be requested in ASCII.

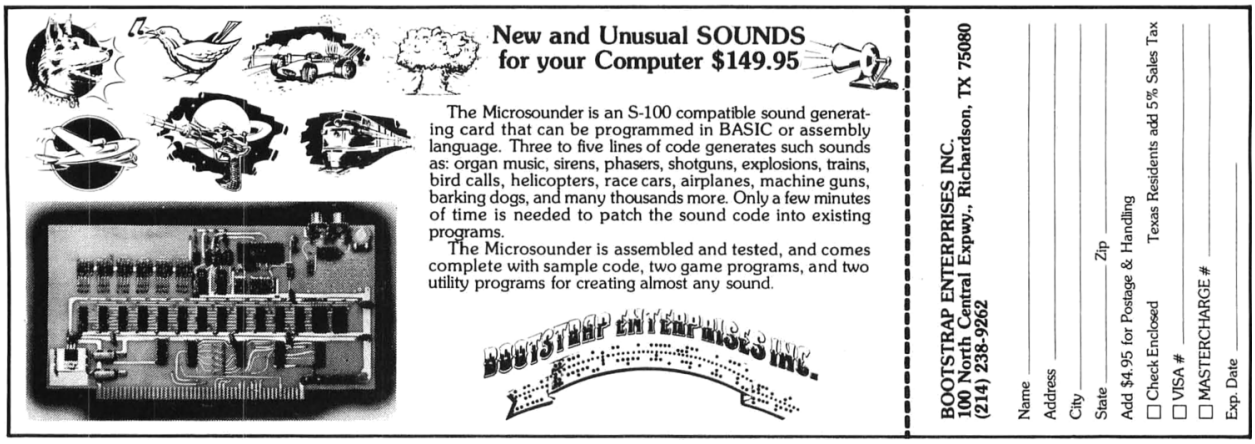
There's some overhead to the HTTPS connection, and some overhead to the searching, so we recommend sending requests in batches of a hundred or so functions.

Let's walk through how the IDA script works, using a fragment in Figure 1. We're going to iterate over the whole program on every function that IDA found in auto-analysis. We'll either grab the boundaries of the `.text` section (if we're in an ELF) with `ida_segment.get_segm_by_name` or we'll just start at memory address `0`, get the next function with `idc.get_next_func(0)` (which will always be the first function in the binary), and work forward to the end of the binary.

The script calls `ida_getfunctionprefix`, a little helper function we wrote to grab the first bytes of the function used as the Symgrate signature, which is currently 18 bytes. We add `(address, function bytes)` pairs to our query string up to 64 times. This allows us to query 64 signatures at a time, with

```
# Iterate over all the functions, querying from the database and printing them.
fnhandled=0;

qstr="";

start=0
end=0
t = ida_segment.get_segm_by_name(".text")
if (t and t.start_ea != ida_idaapi.BADADDR):
    start = t.start_ea
    end = t.end_ea
else:
    start = idc.get_next_func(0)
    end = ida_idaapi.BADADDR

f=start

while (f != ida_idaapi.BADADDR) and (f <= end):
    iname=idc.get_func_name(f)
    adr=f
    adrstr="%x"%f
    res=None

    bstr = ida_functionprefix(f)
    # We query the server in batches of 64 functions to reduce HTTP overhead.
    qstr+="%s=%s&"%(adrstr,bstr)
    f = idc.get_next_func(f)

    if fnhandled&0x3F==0 or f is None:
        res=Symgrate2.queryjfns(qstr)
        qstr=""
        if res!=None:
            Symgrate2.jprint(res)
            #optionally rename functions to the values found in the query
            #ida_renamefunctions(res)

    fnhandled+=1
```

Figure 1: Fragment of `Symgrate2Query.py` from the IDA Pro plugin.



much less network overhead than making a separate HTTPS transaction for each function.

Once we get to 64 functions we submit the query with `Symgrate2.queryjfns`. This function converts marshals the query string into a JSON object and submits the JSON object over HTTP to the server. The server returns `(address, function name)` pairs in a JSON object. By default, the script prints the pairs, but there is also a line that if uncommented will rename functions to the names found by Symgrate. We find it's usually better to see what kind of results you get back first before committing the names to your database.

A Database of SVDs

By this point, we hope to have already convinced you of the value that a Web API server can have for firmware reverse engineering. If you can use our server to recover missing symbols from a firmware image, why not query it for other useful things?

Among our collection of object files, we also have nearly twenty thousand `.svd` files. Each `.svd` file contains an XML description of a Device, the base address of each I/O Peripheral, and the offset after that base address for each I/O Register in that Peripheral.

Querying our server quickly gives all of the registers for the STM32F407. (There's no support for hexadecimal numbers before JSON5; please forgive us if that makes your eyes bleed.)

```
% curl -d STM32F407=STM32F407 -X POST \
  https://symgrate.com/jsvd | jq
{
  "STM32F407": [
    {
      "PeripheralName": "TIM2",
      "Name": "CR1",
      "Adr": 1073741824
    },
    {
      "PeripheralName": "TIM2",
      "Name": "CR2",
      "Adr": 1073741828
    },
    {
      "PeripheralName": "TIM2",
      "Name": "SMCR",
      "Adr": 1073741832
    },
```

But what you if you don't yet know that you are using an STM32F407? Simply send a list of the registers and their access mode, or `u` for undefined, to the server, and it'll give you a list of potentially compatible chips.

```
% curl -d 0x58000148=u -X POST \
  https://symgrate.com/jregs | jq
[
  {
    "Name": "STM32WBxx_CM4",
    "Count": 1
  },
  {
    "Name": "STM32MP1_v0r3",
    "Count": 1
  },
```

– — — – — — — — — — –

We hope to have convinced you find folks that these new-fangled Web APIs are useful, not just for dancing babies and hamster dances, but also to expose valuable databases to otherwise slim plugins of reverse engineering frameworks. Expect some new database functions in the near future, and kindly buy us a beer if the database gives you some useful results.