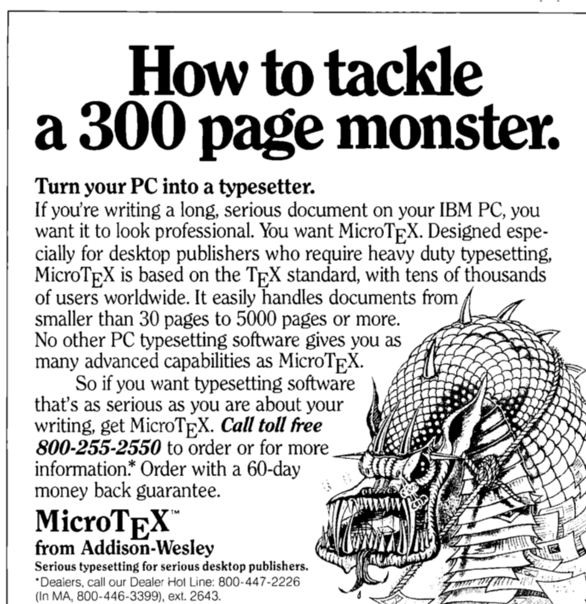
19:04 Undefining the ARM

by Eric Davisson

I'm here today to tell you fine folks about a recent adventure with the ARM architecture, in which I scrambled the undefined bits of instructions to break disassembly without breaking the program's execution.

ARM was something I hadn't touched, so I dug up an old Raspberry Pi and what looked to be a great online resource for learning assembly language, specifically for the Pi. Although it had one handy section on GPIO at the end, this book turned out to be terrible.

Fed up with shallow introductions, I registered with ARM and downloaded the 2,700 page manual. I had to admire the structure and order of the instruction encodings. For the 32-bit form, each instruction is exactly 32 bits, rather than varying from 1 to 15 bytes like x86. Most instructions are conditional, and the first four bits define the conditions. (`0b1110` is the default for unconditional execution.) When browsing the alphabetical instruction list and instruction encoding parts of the manual, I saw that certain bit fields even subdivided instructions into different categories. Some bits then define the specific instruction, and after that, you're pretty much left with the operand data fields.



The Concept

For the register form of the `MOV` command (`MOV Rd, Rm`), we have the 32 bits shown in Figure 1.

As I've mentioned before, those first four bits specify under what condition to execute this `MOV` instruction. The next three bits, `000`, put this instruction into the *Data-processing (register)* category, a fairly common one. Other categories include *Load-/Store*, *Media*, *Branch*, and *Co-processor*. The next five (really four) bits of `1101x` puts us into a sub-sub-category of `Moves, Shifts, and Rotates`. The two bits near the end further divide this into either a `MOV` or `LSL`. The five bits of `00000` is what defines this as a specific instruction of `MOV` (register). We then have the `Rd` and `Rm` fields, which just specify which of the 16 registers to use. Finally the `S` bit defines whether the condition flags are set or not after the instruction is executed.

Well, we skipped a piece! Nothing explained what the `(0)(0)(0)(0)` bits were. So let's flip some and try it out!

In GNU's `as` assembler, you use the `.word` directive to place an arbitrary 32-bit piece of data where an instruction might go.[14] This is a valid instruction of `MOV r0, pc`, defined in `0b` form so that we can see the individual bits.

```
.word 0b11100001101000000000000000001111
```

The Program Counter (PC) register is the 15th (`1111`) register, and it is much like EIP in x86. After stepping through this instruction in `gdb`, I confirmed that the value of `PC+4` is moved into the `r0` register, just as expected. So that is my baseline, my control. Next I flipped one of those `(0)` bits.

```
1 .word 0b11100001101000010000000000001111
```

[14]Editor's Note: All instructions in this article are presented as 32-bit words, rather than as bytes, to better match the ARM manual's descriptions.

[15]`rasm2 -e -a arm -D "e1a0000f e1a1000f"`

I put both of those instructions in my program for comparison, finding that both `gdb` and `objdump` failed to disassemble it.[15]

```
0x10420 main+24 mov      r0, pc
0x10424 main+28 ;<UNDEFINED> ins: 0xe1a1000f
```

Even though the disassembler shows the second instruction as undefined, both of them behave identically, moving `PC+4` into `r0`!

At this point, a false prophet might declare that wherever an instruction matches one with undefined bits, we can flip these bits without changing the behavior of the program. And like many things a false prophet might say, this is *almost* true, but lacking one or two important details. Here, the details matter.

ARM Wrestling

I call my PoC ARMaHYDAN, to pay tribute to the 2004 HYDAN stego tool for x86 by El-Khalil and Keromytis.[16] Like many readers of this fine journal, I am not interested in steganography as a tool to hide information; rather, I love the idea that file formats–and also instruction sets!–have hidden nooks and crannies ignored by their interpreters.

First I cataloged all of the instructions that had these optional bits. From four hundred or so instructions, ignoring conditional codes, only 141 instructions had these bits.

The first script I wrote flipped the last optional bit for all valid instructions in an executable. I did this to `/usr/games/worm` in the bsdgames package, because I like that game. My script used `readelf` to locate and parse the offset and size of the `.text` section; as I only wanted to flip the bits for the code of the program.

About a quarter of the output's `.text` section appeared to be undefined! I then ran the game, and it worked flawlessly. At this point the generalizations seem to hold, but I had only tested against one program.

Still, I wondered if by changing this bits from one instruction, I might convert it to some other instruction. To assure myself, I checked by having a script definitively investigate every encoding. Based on the encodings in the ARM manual, there should be no overlap here.

Just for safe measure I tested a few other programs. My favorite was modifying a quarter of `objdump`, then feeding it itself as an argument to show it report that quarter of its own instructions are undefined.

When it Literally isn't Code!

So now that I was executing modified code, I still needed to know whether these invalid instructions ever occurred naturally in the wild. So I loosened up the parsing for my profiler script to not just match on the valid instruction encodings, but invalid ones too.

The answer to my question was disturbing: there were many of these illegal instructions in the wild! I later found the rate of this occurrence to be evenly distributed from 0-13%. It would get much higher for libraries. I knew something was off about this, as it just wouldn't make sense for assemblers to do this on purpose. Something else was going on.

I finally got a hint when my script began to break, and the breaking change was that I was now matching on *all* forms of instructions, and not just the validly defined ones. Why would it be safe to change any valid instruction, but not these ten percent of already-invalid ones? It turns out I made one of the biggest assumption of all, that the `.text` section is pure code!

So here's what happened: In fixed-width instruction sets like ARM and PowerPC, there is no room in the instruction for a register-wide pointer. ARM solves this problem by placing a pool of literals into

[16] `unzip pocorgtfo19.pdf hydan.pdf hydan-0.13.tar.gz`

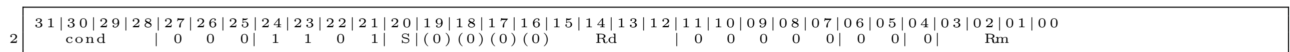


Figure 1. Bitfields of the `MOV` Instruction.

the code, then referencing that location with fewer bits, relative to the program counter.

So when you see `ldr r2, =0xdeadbeef` in the disassembly, you will also see `0xdeadbeef` as a literal *later in the code.* These four bytes are not an instruction, but they are in the `.text` section, and its important not to damage them.

Not Solving the Code/Data Problem

This means I ran into a very old problem, the code versus data problem. My early tests worked out of luck, but that luck ran out when I loosened up the parser can began modifying words in the `.text` section that were not code.

I noticed these false positive instructions did not show up in a consistent frequency; some of them occurred way more than others. For a while it only seemed that two or three problem instructions seemed to show up, so I took them out of my script and everything worked after that. But still, only for the small subset of programs I was modifying and testing.

SS-50 Computing tm
• THE OTHER ALL 6800 COMPUTER MAGAZINE
Devoted to the 6800-6809 enthusiast...Software, fixes, hardware, reviews and more!
Charter Subscription
$12.00-1 Year $22.00-2 Years
VISA MC
FREE SAMPLE ISSUE
(60¢ in stamps for 1st Class)
SS-50 Computing
P.O. Box 402K
Logan, Utah 84321
321

To really understand the situation, I wrote a profiler script to run against my entire Raspbian installation. It showed that these false positives were distributed across more than half the possible instruction set! It was also evenly distributed enough to not be able to justify blacklisting a couple of instructions and hoping for the best.

Well, that's in the context of statically blacklisting some instructions. I considered running an initial profiling pass of the program I'm trying to modify to tally the invalid instructions (most likely data) and keep track of this as a blacklist and store it as metadata. The dynamically blacklisted instructions could be ignored for injecting data into, and the extracting routine could look to the blacklist in metadata to not extract data from those instructions. One downside to this is that more metadata is at the cost of how much data I can inject.

Then I realized that I could encode the entire blacklist in just one byte, by prioritizing the instructions. The byte would simply be the number of high-trouble matches to skip.

I profiled my whole system for a list of instructions based on frequency in a few contexts. The first is just the occurrence of instructions period. This found the top five instructions with optional bits to be `MOV` (register), `CMP` (immediate), `MOV` (immediate), `CMP` (register), and `LSL` (immediate). The top fife for false positives, that are actually data, with option bits are `LDRD` (register), `STRD` (register), `STRH` (register), `MUL`, and `MRS`.

We aren't so lucky that the full lists are mutually exclusive, but they are certainly dissimilar enough to truly minimize the second data loss problem. This is because the instructions I'm actually blacklisting are in the minority of instructions that are actually valid and therefore used. We are losing only a marginal amount of storage space for our injection!

Comparing my top ten lists, the `MUL` instruction is the only one in both my top ten lists, ranked fourth for false positives but tenth for popularity, making up less than one percent of valid instructions. By choosing the right threshold, these lists oughtn't conflict or get in the way of our storage.

DEF CON
Voice Bridge
801-855-3326
Free VMBs - 2 Voice BBS Sections - 5 Voice Bridges
Up to 8 people on a bridge at once/Daily meetings start around 6pm PST
A good place to meet before you start your evening activities

Steganalysis

As I said in the very beginning, using rare machine encodings to inject data for steganography is easily detectable. The concept in HYDAN was that there are different (valid) ways to encode the same assembly instruction, partly because of how messed up things get with x86's `MODRM/SIB` tables and redundancies introduced with not being able to do memory to memory operand instructions. (These are just two basic reasons; there are more.)

Take `xor eax, eax` for example. There is an encoding for `xor r32m32, r32` and also one for `xor r32, r32m32`. In other words, there's a variation for a pointer being the first or second operand depending on the encoding, even though you can choose a register for both. So if you did just choose a register for both, which encoding do you use? Assemblers will prefer only one in this kind of situation, even though both execute in a valid way. A steganographer could use this information to call one encoding a 1, and the other a 0, and encode data with this method. But knowing that, if I suspect an x86 program to be stego'd, the first thing I would check for is the uncommonly encoded instructions like that.

The situation is no different for ARMaHYDAN. Invalid instructions, whether data or stego, ought to be less than 13% of all 32-bit words in the `.text` section, and by carefully observing which ones are executed, it oughtn't be hard to identify the existence of hidden content.

Cut out the NULLs!

Another nifty result of this project is that many of the null bytes in ARM machine code contain at least a bit or two that the CPU will ignore. Take a moment to reread the brilliant Phrack 66:12, in which Yves Younan and Pieter Philippaerts used a dozen clever tricks to make alphanumeric self-modifying shellcode in a creole dialect of both ARM and Thumb,[17] then consider how much easier it might be if so many of their blacklisted instructions[18] could be smuggled in by flipping a bit here or there.

Native	Assembly	Modified
e10100d0	ldrd r0, [r1, -r0]	e10101d0
e10100f0	strd r0, [r1, -r0]	e10101f0
e10100b0	strh r0, [r1, -r0]	e1010fb0
e0100090	muls r0, r0, r0	e0101090
e11000d0	ldrsb r0, [r0, -r0]	e11001d0
e11000b0	ldrh r0, [r0, -r0]	e11001b0
e11000f0	ldrsh r0, [r0, -r0]	e11001f0
e1100080	tst r0, r0, lsl #1	e1101080
e3100080	tst r0, #128	e3101080
e1500080	cmp r0, r0, lsl #1	e1501080
e1300080	teq r0, r0, lsl #1	e1301080
e1700080	cmn r0, r0, lsl #1	e1701080
e3700080	cmn r0, #128	e3701080
e3300080	teq r0, #128	e3301080
e1100010	tst r0, r0, lsl r0	e1101010
e3500080	cmp r0, #128	e3501080
e1400090	swpb r0, r0, [r0]	e1400190
e1700010	cmn r0, r0, lsl r0	e1701010
e1500010	cmp r0, r0, lsl r0	e1501010
e1300010	teq r0, r0, lsl r0	e1301010
f1010000	setend le	f1010401
e1200050	qsub r0, r0, r0	e1200150
e03000b0	ldrht r0, [r0], -r0	e03001b0
e03000d0	ldrsbt r0, [r0], -r0	e03001d0
e03000f0	ldrsht r0, [r0], -r0	e03001f0
e12000a0	smulwb r0, r0, r0	e12010a0
...	...	...

Figure 2. ARM Instructions with a Null Byte

Final Thoughts

This project is not ground breaking, but by reading the ARM manual and chasing down the unexplained bitfields, I managed to learn a lot about the architecture and have some fun in the process.

As you read my code,[19] please remember that the fun is in the journey and not the destination. Don't just theorize about what new tricks might be done! Read the documentation, and when the inspiration hits, run the experiments that will teach you the facts you need to write a nifty proof of concept.

[17] `unzip pocorgtfo19.pdf phrack6612.txt`
[18] Ibid, §2.3.
[19] `git clone https://github.com/XlogicX/ARMaHYDAN || unzip pocorgtfo19.pdf ARMaHYDAN.zip`