15:09 Detecting Emulation with MIPS16 Delay Slots

by Ryan Speers and Travis Goodspeed
with the kindest of thanks to Thorsten Haas.

Howdy y'all,

Let's begin with a joke that I once heard at a conference: *David Patterson and John Hennessy walk into a bar. Everyone gathers to listen to the two heroes who built legendary machines. The entire bar spends the night multiplying fractions, and then everyone has that terrible hangover you get when you realize you had no fun and learned nothing new, even though your night started out so promising.*

But let's tell the joke differently: *Patterson and Hennessy walk into a bar in another town, but this time, Greg Peterson is behind the bar. The two of them begin a long-winded story about weighted averages, lashing out at "RISC-deniers" who aren't even in the room. Just as folks begin to get bored, and begin to sip their drinks too quickly out of nervousness, Peterson jumps in and saves the day. Because he knows that these fine folks build real machines that really shipped, he redirects the conversation to war stories and practical considerations.*

Patterson tells how the two-stage pipeline in the RISC 1 chip was the first design with a branch delay slot, as there's no point in throwing away the staged instruction that has already finished execution. Hennessy jumps in with a tale of dual instruction sets on MIPS, allowing denser code without abandoning the spirit of the RISC faith. Then Peterson, the bartender, serves up a number of Xilinx devkits to bar patrons, who begin collaborating on a five-stage pipeline design of their own, with advice on specific design choices from David and John. The next morning, they've built a working CPU and suffered no hangovers.

If your Computer Architecture class was more like the former than the latter, I hope that this brief article will show you some of the joy of this fine subject.

In PoC||GTFO 6:6, Craig Heffner discussed a variety of methods for detecting Qemu emulation of MIPS hardware. We'll be discussing one more way to detect emulation, but we'll be using the MIPS16 instruction set and a clever trick of delay slots to detect the emulation.

We wanted to craft a capability that is (a) able to differentiate hardware from an emulation environment, and also (b) able to confuse static analysis. We picked used standard tools: Qemu as an emulation environment and IDA Pro as a disassembler.[34]

The first criterion leads us to want something that both: (a) works in userland, and (b) is not trivial for an emulator developer to patch. Moving to userland meant that hardware registry inspection, as discussed in Section 6.1 of Heffner's article, would not work. Similarly, the technique of reading `cpuinfo` in Section 6.2 would be easily patchable, as Craig noted. Here, we instead seek a capability more similar to Section 6.3, where cache incoherency is exploited to differentiate real hardware and Qemu.

MIPS16e

SSH'ing to a newly acquired MIPS box, we find the same nifty line of `cpuinfo` that struck our fancy in Craig's article. MIPS16 is an extension to the classic MIPS instruction set that fills the same niche as Thumb2 does on ARM. The instructions word is 16 bits wide, a subset of the full register set is directly available, and a core tenet of RISC is violated: some instructions are more than one word long.

```
$ cat /proc/cpuinfo
system type           : BCM7358A1 STB platform
cpu model             : Broadcom BMIPS3300 V3.2
cpu MHz               : 751.534
tlb_entries           : 32
isa                   : mips1 mips2 mips32r1
ASEs implemented      : mips16
```

Just like ARM, this alternate instruction set is used whenever the least significant bit of the program counter is set. Function pointers work as expected between the two instruction sets, and the calling conventions are compatible.

[34]We will happily buy the drinks in celebration of Radare2 issue 1917 and Capstone issue 241 being closed.

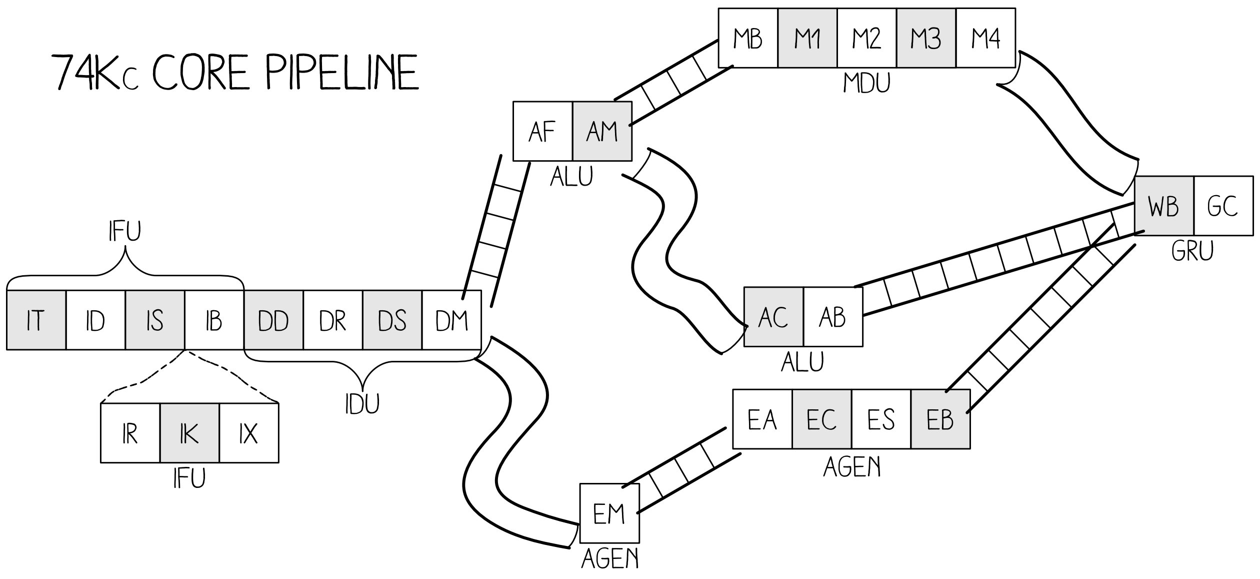


Figure 7. MIPS 74Kc Pipeline

Despite careful work to maintain compatibility between MIPS16 and MIPS32, there are inevitable differences. MIPS16 only has direct access to eight registers, rather than the 32 of its larger cousin.

CPU Pipelines

In Hennessy and Patterson's books, a five-stage pipeline is described and hammered into the poor reader's head. This classic RISC pipeline isn't what you'll find in modern chips, but it's a lot easier to keep in mind while working on them. The stages in order are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).

Each pipeline stage can only hold one instruction at a time, but by passing the instructions through as a queue, multiple instructions can exist in *different stages* at the same time. When a branch is mis-predicted, the pipeline will be "flushed," which is to say that the partially-completed instructions from the incorrectly guessed branch are blown to the wind and replaced with harmless NOP instructions, which are sometimes called "bubbles."

Bubbles are also one way to avoid "data hazards," which are dependencies between instructions that run at the same time. For example, if you were to use a value just after loading it, the CPU would have to either insert a bubble to delay the second instruction until the value is ready or it would "forward" the register result.[35]

The MIPS 74Kc on one of our target machines has 14 or 15 pipeline stages, depending upon how you count, plus three additional stages for MIPS16e instruction decoding.[36] These stages are quite well documented, but to ease the explanation a bit, we won't bore you with the details of exactly what happens where. The stages themselves are shown in Figure 7, helpfully illustrated by Ange Albertini.

Extended (Wide) Instructions

We mentioned earlier that MIPS16 instructions are usually just one instruction word, but that sometimes they are two. That's a bit vague and hand-wavy, so we'd like to clear that up now with a concrete example.

There is an Extend Immediate instruction which allows us to enlarge the immediate field of another MIPS16 instruction, as its immediate field is smaller than that in the equivalent 32-bit MIPS instruction. This instruction is itself two bytes, and is placed directly before the instruction which it will extend, making the "extended instruction" a total of four bytes.

[35]Very early MIPS machines made the hazard the compiler's responsibility, in what was called the "load delay slot." It is separate from the "branch delay slot" that we'll discuss in a later section, and is no longer found in modern MIPS designs.

[36]`unzip pocorgtfo15.pdf mips74kc.pdf`

For example, the opcode for adding an immediate value of 1 to `r2` is `0x4a01`. (`r2` is the register for both the first argument to a function and its return value.) Because MIPS16 only encodes room for five immediate bits in this instruction, it allows for an extension word before the opcode to include extra bits. These can of course be zero, so `0xF000 0x4a01` also means `addi r2, 1`.

Some combinations are illegal. For example, extending the immediate bits of a NOP isn't quite meaningful, so trying to execute `0xF008 0x6500` (Extended Immediate NOP) will trigger a bus error and the process will crash.

The Extended Shift instruction shown along with a regular Shift in Figure 8. Now how the prefix word changes the meaning of the subsequent instruction word.

However, thinking of these two words as a single instruction isn't quite right, as we'll soon see.

Delay Slots

Unlike ARM and Thumb, but like MIPS32 and SPARC, MIPS16 has a branch delay slot. The way most folks think of this, and the way that it is first explained by Patterson and Hennessy,[37] is that the very next instruction after a branch is executed regardless of whether the branch is taken.

Sometimes this is hidden by an assembler, but a disassembler will usually show the instructions in their physical order. IDA Pro helpfully groups the delay-slot instruction into the proper block, so in graph view you won't mistake it for being conditionally executed.

Extended Instructions in a Delay Slot

So what happens if we put a multi-word instruction into the delay slot? IDA Pro, being first written for X86, assumes that X86 rules apply and the whole chunk is one instruction. Qemu agrees, and a quick test of the following code reveals that the full instruction is executed in the delay slot.

We can test this as we see that on both real hardware and Qemu, extending an instruction like a `NOP` that shouldn't be extended will trigger a bus error. However, when we put this combination after a return, it will only crash Qemu. In this case in hardware, only the extension word was fetched, which didn't cause an issue.

```
0xE820 //Return.
0xF008 //Extension word.
0x6500 //NOP, will crash if extended.
```

This is a known issue with the MIPS16e instruction set.[38] To quote page 30, *"There is only one restriction on the location of extensible instructions: They may not be placed in jump delay slots. Doing so causes UNPREDICTABLE results."*

Making Something Useful

We can now crash an emulator while allowing hardware to execute, but let's improve this technique into something that can be used effectively for evasion. We'll replace the `NOP` which caused the crash when extended with an instruction which is intended to be extended, specifically an add immediate, `addi`.

```
0x6740 // First we zero r2, the
       // return value.
0xE820 // jr $ra   (Return)
0xF000 // Extended immediate of 0.
0x4A01 // Add immediate 1 to r2.
       // (only executed in Qemu)
```

If we take that shellcode and view the IDA disassembly for it, you will see that, as above, IDA groups the delay-slot instruction into the function block so it looks like one is added to the return value. See Figure 9, being careful to remember that `$v0` means `r2`.

[37]Page 444 of Computer Organization and Design, 2nd ed.
[38]`unzip pocorgtfo15.pdf mips16e-isa.pdf`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHIFT					rx			ry			sa[a]			f	

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTEND					sa 4:0					s5[a]	0	0	0	0	0	SHIFT					rx			ry			0	0	0	f	

Figure 8. MIPS16 Regular and Extended Shift Instructions

But hang on a minute, that delay slot holds two instruction words, and as we learned earlier, these can be thought of as separate instructions!

In fact, IDA only shows the instruction bytes on the left if you explicitly request a number of bytes from the assembly be shown. Without these being shown, a reverse engineer might forget that the program assembled a double-length instruction and thus that this behavior will occur.

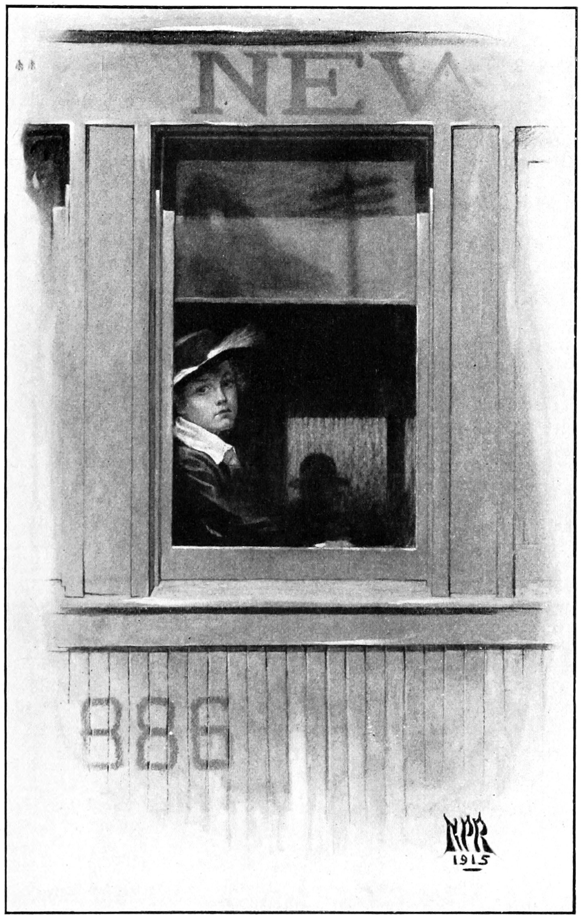
This shows how we can confuse static analysis tools, which disassemble without taking into account this special case.

Let's now look at what happens when we take the above shellcode and execute it as a function from a program. We print the return value from the function in the below sample output.

```
int exec16(int (*fptr16)(int),
           int verbose){
  uint32_t res;
  uint8_t* bytes;
  int (*functionPtr)(int);
  functionPtr=(void*) (((int)fptr16)|1);
  return functionPtr(0xdeadbeef);
}

uint16_t amiemulated16[]={
0x6740, // First we zero r2, the
        // return value.
0xE820, // jr $ra   (Return)
0xF000, // Extended immediate of 0.
0x4A01  // Add immediate 1 to r2.
        // (only executed in Qemu)
};

int main() {
  printf("I am running %s.\n",
    exec16((void*) amiemulated16, 0)
    ? "in Qemu"
    : "on real hardware");
  return 0;
}
```



"ONE DOES NOT TRAVEL ELEVEN THOUSAND MILES WITHOUT ACQUIRING THE RIGHT TO BE TIRED."

We've discussed how IDA sees the extended addition as a single instruction, when in fact they are two separate MIPS instructions. But how is this handled in an emulator versus real MIPS hardware?

On the real hardware, when the return instruction is processed, the next instruction in the pipeline is `0xF000` (the extension instruction) and this is executed in the branch delay slot. That instruction, however, becomes a `NOP` in hardware.

```
ROM:0000 .set mips16
ROM:0000 # ===== SUBROUTINE =====
ROM:0000 amiemulated:
ROM:0000 67    40          move  $v0, $zero # Clear return value to zero.
ROM:0002 E8    20          jr    $ra        # Return
ROM:0004 F0    00 4A 01    addiu $v0, 1     # Adds 1 to return value in Qemu.
ROM:0004 # End of function amiemulated      # This becomes a NOP on real hardware.
```

Figure 9. MIPS16 Machine Code abusing the Delay Slot

THE GODS ARE ATHIRST

By ANATOLE FRANCE

A Translation by ALFRED ALLINSON

Demy 8vo. 6s.

SOME PRESS OPINIONS.

STANDARD.—"'Les Dieux ont Soif' has appeared in English, and a new section of our public has the opportunity of learning what the sagest writer of to-day thinks of the French Revolution. Only supreme genius could have given him the power to enter so fully into the minds of the diverse beings with whom he peoples his narrative. Here we have him as a lover of the whole human race, even though he laughs gently at their strange ways. It is a wonderful book."

MANCHESTER GUARDIAN.—"Most spiritedly translated in this fine edition, it is a sane book about a mad year. His attitude is so finely pondered, so sensitively balanced, so penetrating and ironic that the Terror slips into the natural order of things."

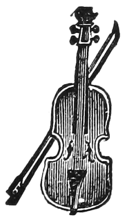
SUNDAY TIMES.—"The tale reveals an extraordinarily fine grasp of history, an insight that is almost uncanny into the thoughts and phrases of the Jacobin doctrinaires and a genuine sense of drama and climax. This calm, scholarly, worldly-wise, ironic philosopher setting himself to evoke by the magic of his art the days in which the fanatics of the French Revolution tried to make a nation virtuous by mere enactments—could we have a more piquant experiment in fiction."

BOOKMAN.—"In this brilliant and fascinating book Anatole France creates for us the atmosphere of the Revolution. Here, as elsewhere, Anatole France displays a wealth of minute learning. He is a master of unobtrusive detail, exquisite preciseness and finish of style."

ACADEMY.—"In this case the translation has been well and simply achieved, with the result that very little of the grim power, humour and pathos of the genius of Anatole France has been lost."

GUARDIAN.—"In this brilliant vivid study, the French Revolution appears as a more vital, because more vividly imagined, movement than in most of the histories of the period."

JOHN LANE, THE BODLEY HEAD, VIGO ST., W.



ALL KINDS OF

Stringed Instruments,

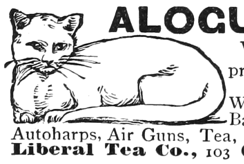
Parts thereof.

FINE STRINGS, High-Grade Repairing.

BREITKOPF & HÄRTEL, I.

39 E. 19th St., New York.

Write for Catalogue.



ALOGUE FREE!

We give the following premiums with **TEA.**

Watches, Solid Gold Rings Banquet Lamps, Banjos, Autoharps, Air Guns, Tea, Dinner and Toilet Sets.

Liberal Tea Co., 103 Cross St., Boston, Mass.

```
~$ uname -a
Linux target 3.12.1 #1 mips GNU/Linux
~$ ./hello
I am running on real hardware.
```

The reason this detection works, we hypothesize, is because Qemu doesn't actually have a pipeline, and thus it is emulated by knowing that it should run the instruction following a branch, to "correctly" handle the branch-delay slot. When it reads that next instruction, it reads the two instructions that it sees as a single extended instruction, instead of just reading the extension.

```
~$ mips-linux-gnu-gcc -static -std=gnu99 \
   hello.c  -o hello
~$ qemu-mips -L /usr/mips-linux-gnu hello
I am running in Qemu.
```

In hardware, we should note, the instruction isn't exactly tossed away because it's broken in half. The extension word, as the first half of the pair, never really gets executed on its own; rather, it hangs around in the pipeline to modify the subsequent instruction word. As the pipeline flows, the first word becomes a bubble as the second word becomes the single, unified instruction, but that unified instruction is too late to be executed. Instead, it is cruelly flushed from the MIPS16 pipeline while the bible ahead of it becomes a worthless NOP.

Thus, with just the eight byte function `0x6740 0xe820 0xf000 0x4a01`, we can reliably detect emulation of MIPS16. As an added bonus, IDA Pro will agree with the simulation behavior, rather than the hardware behavior.

– – – – – – – – – – –

Kind thanks are due to Thorsten Haas for lending us a MIPS shell account on impossibly short notice. If you'd like to play around with more differences between hardware and emulation, we'll note that in MIPS32, `0x03E00008 0x03E00008` is a clean return to `$ra` on hardware, but crashes Qemu. To crash on hardware and return normally in Qemu, use `0x03e0f809 0x8fe20001`.

Cheers from Hanover, New Hampshire,
Travis and Ryan