Reverse engineering a SmartNIC

NVIDIA/Mellanox ConnectX-5

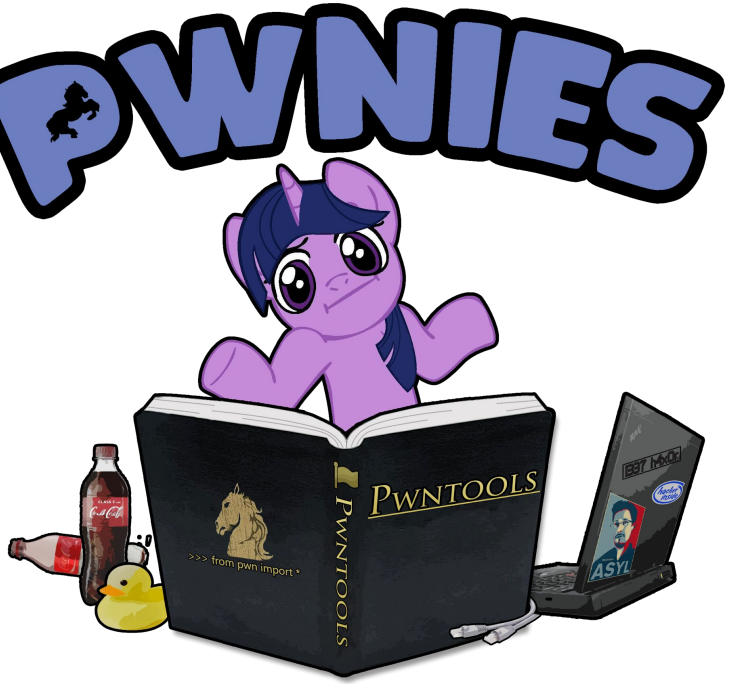
The plan

1. What is Pwnies?
2. What is it a ConnectX-5? What can it do?
3. Firmware layout & Tooling
4. Embedded CPU(s) (iRISC?)
5. Reversing an instruction set
6. Writing a ghidra processor plugin
7. Firmware patching and code execution
8. Fuzzing an instruction set
9. Writing a PCIe driver in rust
10. Future work?

What is Pwnies?

- We are a “student” organization at DIKU
- We made Pwntools

Come join us



What is a ConnectX-5?

- 100Gb/s or 2x100Gb/s QSFP
- Overlay offloading(VXLAN/GENEVE/…)
- SR-IOV/eSwitch (Switching between VMs on the same HV)
- Newer generations: CX6, CX7: TLS offloading, remote management, …
- Price on Ebay: ~1000dkk
- Good open source support(Drivers, tooling, …)
- Some public documentation

Firmware: Tooling & Layout

Mellanox has published tools! (https://github.com/Mellanox/mstflint)

… and firmware images

Firmware: Tooling & Layout

- MAIN_CODE?
- BOOT2?
- IRON_PREP_CODE?
- Public keys? Signatures?

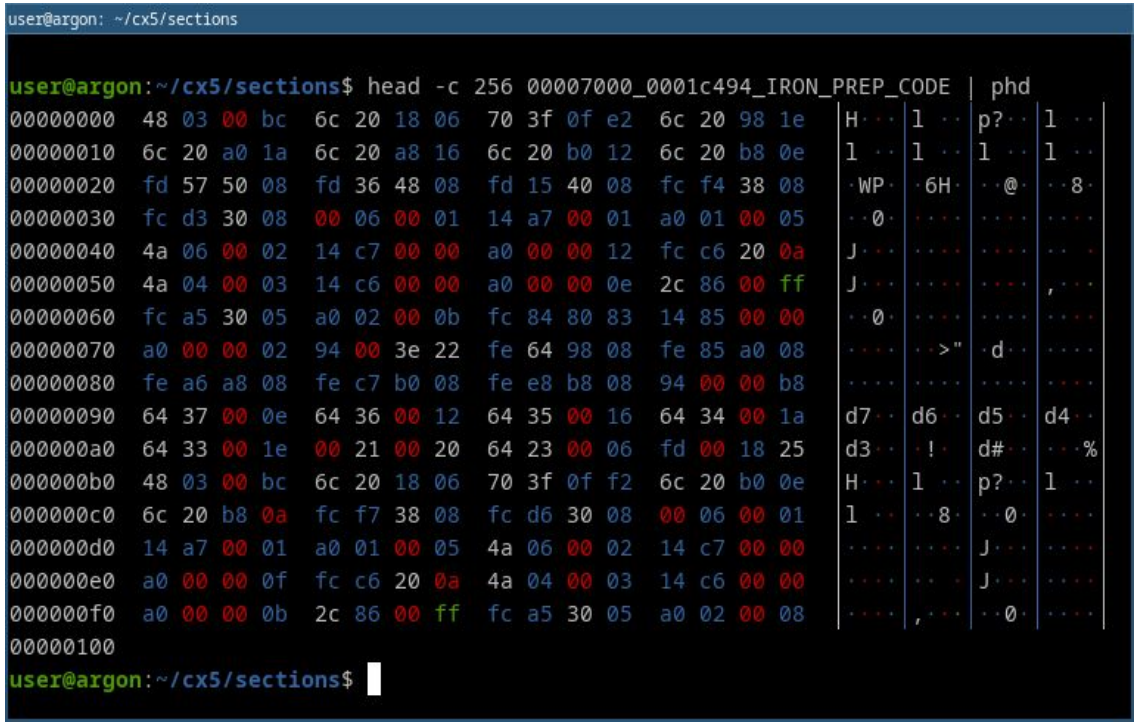
```
# mstflint -i orig_cx5_firmware.patch verify
FS4 failsafe image

/0x00000018-0x0000001f (0x000008)/ (HW_POINTERS) - OK
...
/0x00000090-0x00000097 (0x000008)/ (HW_POINTERS) - OK
/0x00000500-0x0000053f (0x000040)/ (TOOLS_AREA) - OK
/0x00001000-0x00003137 (0x002138)/ (BOOT2) - OK
/0x00004000-0x0000401f (0x000020)/ (ITOC_HEADER) - OK
/0x00006000-0x0001902b (0x01302c)/ (IRON_PREP_CODE) - OK
/0x0001902c-0x0001912b (0x000100)/ (RESET_INFO) - OK
/0x0001915c-0x003ac65b (0x393500)/ (MAIN_CODE) - OK
/0x003ac65c-0x003bfeab (0x013850)/ (PCIE_LINK_CODE) - OK
/0x003bfeac-0x003c09db (0x000b30)/ (POST_IRON_BOOT_CODE) - OK
...
/0x0040fd24-0x0040fe63 (0x000140)/ (IMAGE_SIGNATURE_256) - OK
/0x0040fe64-0x00410763 (0x000900)/ (PUBLIC_KEYS_2048) - OK
/0x00410764-0x004107f3 (0x000090)/ (FORBIDDEN_VERSIONS) - OK
/0x004107f4-0x00410a33 (0x000240)/ (IMAGE_SIGNATURE_512) - OK
/0x00410a34-0x00411b33 (0x001100)/ (PUBLIC_KEYS_4096) - OK
...

-I- FW image verification succeeded. Image is bootable.
```

Firmware: Tooling & Layout

- Patterns?
- Endianness?
- Instruction width?
- Function prolog/epilog?



Embedded CPU: iRISC

- Docs/tooling: 11 cores?, RISC!, 32bit?, tracing over PCIe?
- Patent search: Not really anything useful…
- Almost no information available. :(

What can we do with this little information?

Is reverse engineering possible?

Reverse engineering an ISA

- Guess instruction layout!
  - Width of instructions: 32 bits
  - Width of opcode: 6 bits
  - Number of register: 32 registers(5 bits)
- Guess opcodes based on common patterns
  - Prolog/epilog of functions: Matched blocks of loads/stores, matched stack adjustments
  - Calls: Should target start of functions.
  - Related instructions are ‘near’ opcode-wise

Reverse engineering an ISA: Disassembler/Guessing

user@argon: ~/cx5/presentation

```
from pwn import *
context(endian="big")

code = read("00007000_0001c494_IRON_PREP_CODE")

for i, inst in enumerate(map(u32, group(4, code))):
    op = (inst >> 26) & 0x3f
    rs = (inst >> 21) & 0x1f
    rd = (inst >> 16) & 0x1f
    rt = (inst >> 11) & 0x1f
    imm16 = inst & 0xffff
    imm11 = inst & 0x7ff

    opcode = {
        0x1b: f"st.d {rt}, {rs}, {imm11:#03x}",
        0x19: f"ld.d {rd}, {rs}, {imm11:#03x}",
    }.get(op, f"unk.{op:02x} {rd}, {rs}, {imm16:#04x}")

    print(f"{4*i:08x}:\t{inst:08x}\t{opcode}")
~
                                    19,1          All
```

user@argon: ~/cx5/presentation

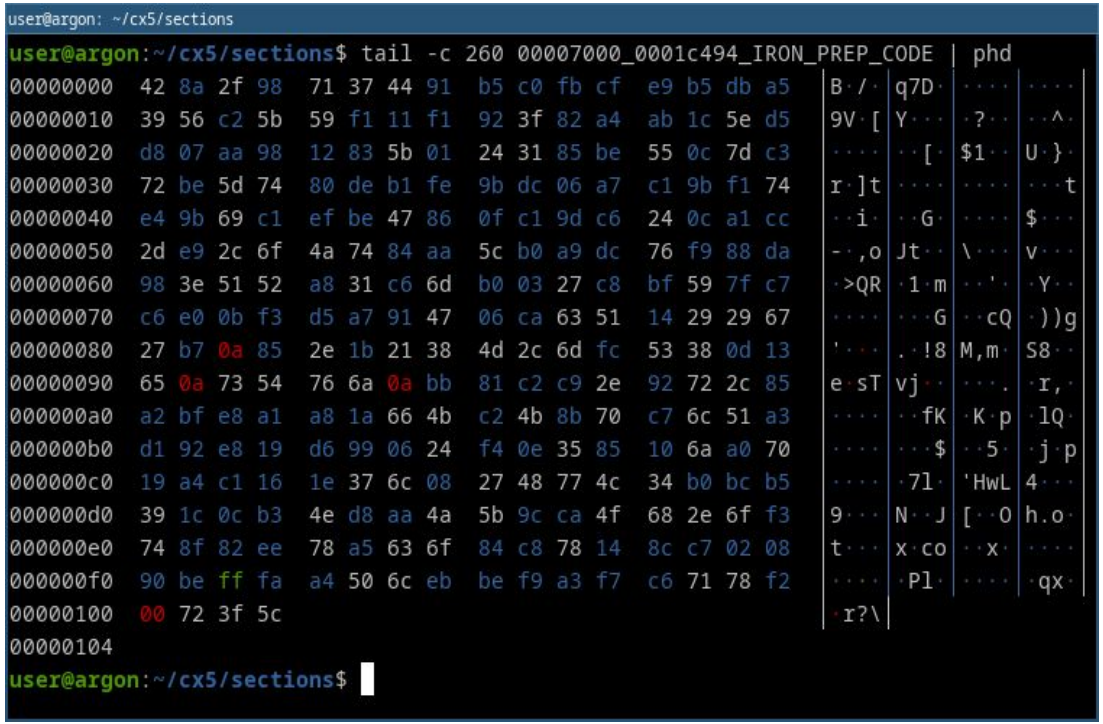
```
00000000:       unk.12 3, 0, 0xbc       <- ???, load r3 with something
00000004:       st.d 3, 1, 0x6          <- store to stack, r1 stackpointer?
00000008:       unk.1c 31, 1, 0xfe2     <- stack adjustment? store instruction?
0000000c:       st.d 19, 1, 0x1e        <- save regs, unaligned offset?
00000010:       st.d 20, 1, 0x1a
00000014:       st.d 21, 1, 0x16
00000018:       st.d 22, 1, 0x12
0000001c:       st.d 23, 1, 0xe
... lots of unknown instructions ...
00000090:       ld.d 23, 1, 0xe         <- restore regs, same offset/reg pairs
00000094:       ld.d 22, 1, 0x12
00000098:       ld.d 21, 1, 0x16
0000009c:       ld.d 20, 1, 0x1a
000000a0:       ld.d 19, 1, 0x1e
000000a4:       unk.00 1, 1, 0x20       <- add r1, r1, 0x20 ???
000000a8:       ld.d 3, 1, 0x6          <- load r3, return address ??
000000ac:       unk.3f 0, 8, 0x1825     <- return ???, 0x18 -> r3?

000000b0:       unk.12 3, 0, 0xbc       <- New function prolog
000000b4:       st.d 3, 1, 0x6
000000b8:       unk.1c 31, 1, 0xff2
000000bc:       st.d 22, 1, 0xe
000000c0:       st.d 23, 1, 0xa
...
                                                9,36          All
```

Reverse engineering an ISA: Rosetta stone

- High entropy?
- Magic bytes?
- What is this?

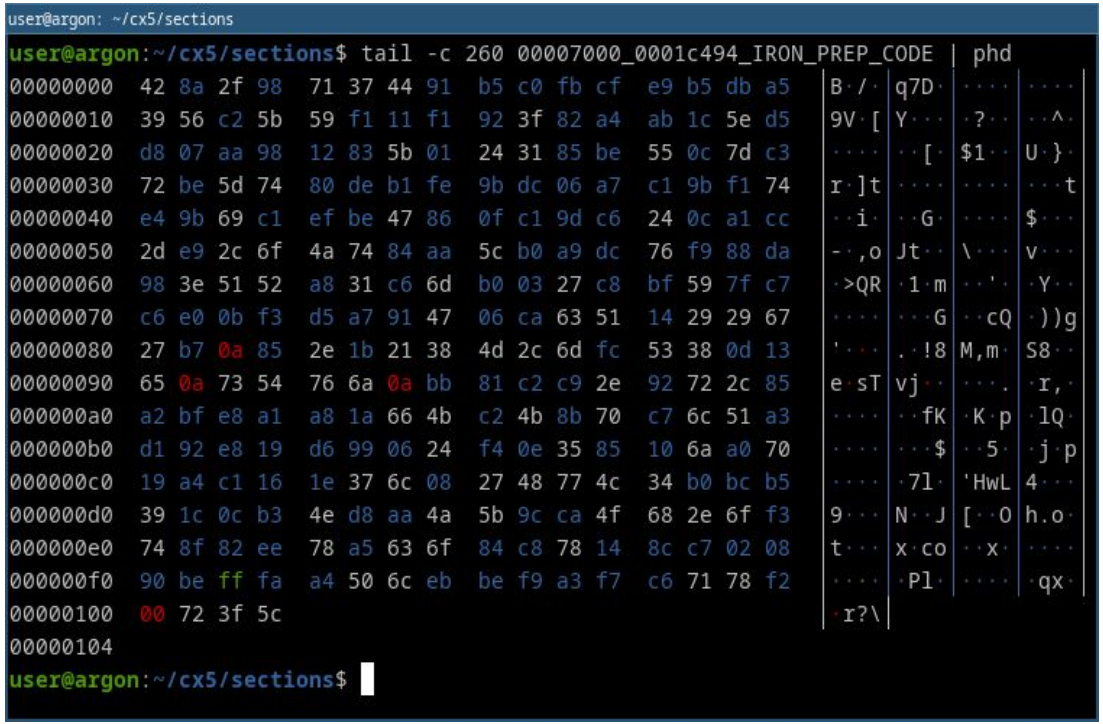
Let’s ask google?



Reverse engineering an ISA: Rosetta stone

- These are the ‘K’ constants from SHA256
- Firmware implements SHA256 somewhere!!
- Operations in SHA256:
  - add/sub/xor/and/or/shifts
  - loads/stores of different sizes
  - Lots of register use => caller/callee saved registers
  - loops/branches/compares

Can we find the SHA256 implementation in firmware?



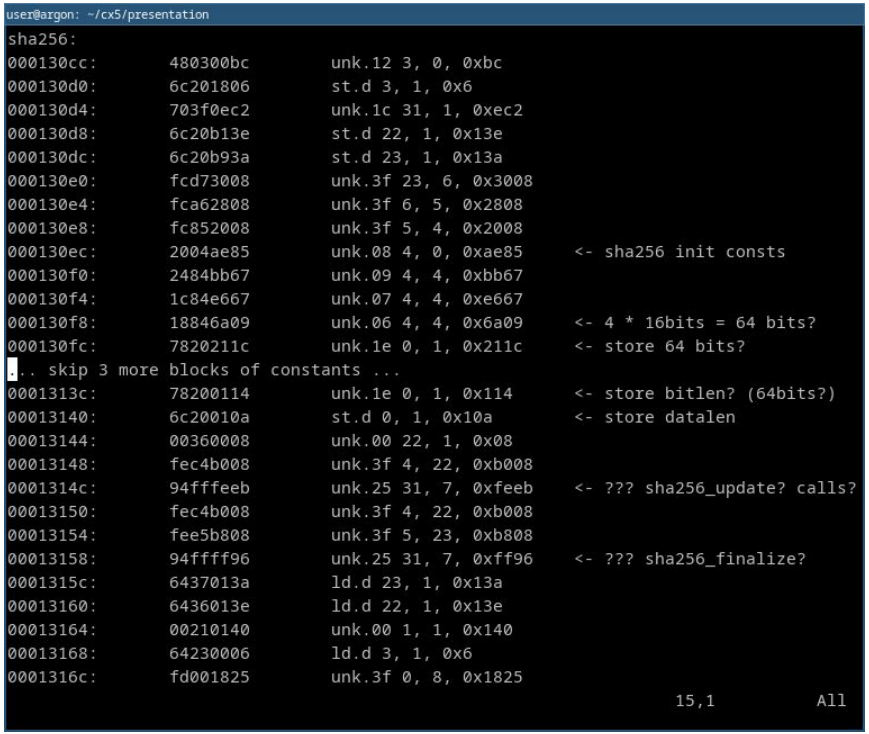
Reverse engineering an ISA: Finding SHA256

- SHA256 is usually implemented in 4 functions:
  - **sha256_init:** has initialization constants
  - **sha256_transform:** uses ‘K’ constants
  - **sha256_update:** calls **sha256_transform**
  - **sha256_finalize:** calls **sha256_transform**
- Searching for specific constants
- Constants should be in code
- Plan:
  - Find **sha256_init** by searching for constants
  - Find calls to **sha256_init**
  - Assume that **sha256_update** and **sha256_finalize** is called nearby

```
void sha256_init(SHA256_CTX *ctx)
{
        ctx->datalen = 0;
        ctx->bitlen = 0;
        ctx->state[0] = 0x6a09e667;
        ctx->state[1] = 0xbb67ae85;
        ctx->state[2] = 0x3c6ef372;
        ctx->state[3] = 0xa54ff53a;
        ctx->state[4] = 0x510e527f;
        ctx->state[5] = 0x9b05688c;
        ctx->state[6] = 0x1f83d9ab;
        ctx->state[7] = 0x5be0cd19;
}
```

Reverse engineering an ISA: Reversing SHA256

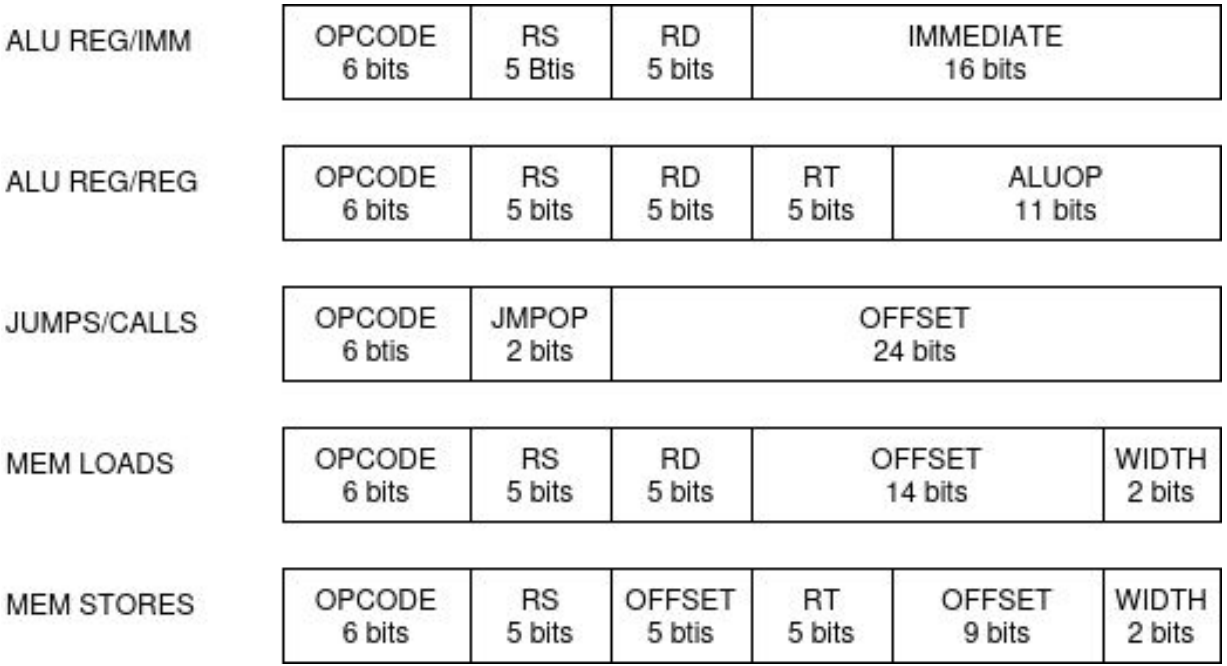
- Found sha256 constants!
- Looks like it calls **sha256_update** and **sha256_finalize** right after.
- Looks like we have 64 bits registers?



Result: Found **sha256_transform**, a lot more known instructions.

What we know so far: iRISC instruction layout

- Many alu ops
- loads/stores
- Calls/jumps
- Branches
- Register size
- Calling convention



Custom Ghidra Processor module

- Reverse engineering using python disassembler is annoying
- Ghidra has support for custom architectures
- Sleigh: DSL for describing ISAs
- Already >30 ISAs implemented in Sleigh, lots of examples.
- Table based, highly flexible.
- + Few XML files to describe:
  - Basic facts: name, link to docs, level of support, compiler differences, …
  - Calling convention
  - Special registers: stack pointer, program counter, …

Ghidra processor module: Sleigh (Address spaces)

- Two address spaces
  - RAM (code/data)
  - Registers
- Bind names to specific addresses
- Multiple bindings to same address (eg. r5 = r5h || r5l)

```
define endian=big;

define alignment=4;

define space ram type=ram_space size=4 default;
define space register type=register_space size=4;

define register offset=0x00000 size=8 [
    zero r1 r2 r3 r4 r5 r6 r7
    r8 r9 r10 r11 r12 r13 r14 r15
    r16 r17 r18 r19 r20 r21 r22 r23
    r24 r25 r26 r27 r28 r29 r30 r31
];

define register offset=0x00000 size=4 [
    zeroh zerol r1h r1l r2h r2l r3h r3l r4h r4l r5h r5l r6h r6l r7h r7l
    r8h r8l r9h r9l r10h r10l r11h r11l r12h r12l r13h r13l r14h r14l r15h r15l
    r16h r16l r17h r17l r18h r18l r19h r19l r20h r20l r21h r21l r22h r22l r23h r23l
    r24h r24l r25h r25l r26h r26l r27h r27l r28h r28l r29h r29l r30h r30l r31h r31l
];
```

Ghidra processor module: Sleigh (Instruction fields)

- Define slices of bits
- Attach registers to bits of instructions.

```
define token instr(32)
    op=(26, 31)
    rs=(21, 25)
    rshi=(21, 25)
    rslo=(21, 25)
    jmpop=(24, 25)
    imm24=(0, 23)
    simm24=(0, 23) signed
    rd=(16, 20)
    rdhi=(16, 20)
    rdlo=(16, 20)
    cmpop=(16, 20)
    cmpshamt=(16, 20)
    rt=(11, 15)
    rthi=(11, 15)
    rtlo=(11, 15)
    shamt=(11, 15)
    imm16=(0, 15)
    simm16=(0, 15) signed
    funct=(0, 8)
```

```
attach variables [ rd rs rt ] [
    zero r1 r2 r3 r4 r5 r6 r7
    r8 r9 r10 r11 r12 r13 r14 r15
    r16 r17 r18 r19 r20 r21 r22 r23
    r24 r25 r26 r27 r28 r29 r30 r31
];

attach variables [ rdhi rshi rthi ] [
    zeroh r1h r2h r3h r4h r5h r6h r7h
    r8h r9h r10h r11h r12h r13h r14h r15h
    r16h r17h r18h r19h r20h r21h r22h r23h
    r24h r25h r26h r27h r28h r29h r30h r31h
];

attach variables [ rdlo rslo rtlo ] [
    zerol r1l r2l r3l r4l r5l r6l r7l
    r8l r9l r10l r11l r12l r13l r14l r15l
    r16l r17l r18l r19l r20l r21l r22l r23l
    r24l r25l r26l r27l r28l r29l r30l r31l
];
```

Ghidra processor module: Sleigh (Instructions/tables)

```
RD: rd is rd { export rd; }
RDsrc: rd is rd { export rd; }
RDsrc: rd is rd & rd=0 { export 0:8; }

RSsrc: rs is rs { export rs; }
RSsrc: rs is rs & rs=0 { export 0:8; }

RTsrc: rt is rt { export rt; }
RTsrc: rt is rt & rt=0 { export 0:8; }

RDlo: rdlo is rdlo { export rdlo; }
RDlosrc: rdlo is rdlo { export rdlo; }
RDlosrc: rdlo is rdlo & rdlo=0 { export 0:4; }

RSlo: rslo is rslo { export rslo; }
RSlosrc: rslo is rslo { export rslo; }
RSlosrc: rslo is rslo & rslo=0 { export 0:4; }

RTlosrc: rtlo is rtlo { export rtlo; }
RTlosrc: rtlo is rtlo & rtlo=0 { export 0:4; }
```

```
:unk.^op RD, RSsrc, RTsrc, imm16 is op & RD & RSsrc & RTsrc & imm16 {
    RD = unkOp(op:1, RSsrc, RTsrc);
}

:addi RD, RSsrc, simm16         is op=0x00 & RD & RSsrc & simm16 {
    RD = RSsrc + simm16;
}

:addi RD, RSsrc, simm16         is op=0x01 & RD & RSsrc & simm16 {
    RD = RSsrc + simm16;
}

:addi.hi RD, RSsrc, simm16      is op=0x02 & RD & RDsrc & RSsrc & simm16 {
    RD = RSsrc + (simm16 << 16);
}

:addi.hi RD, RSsrc, simm16      is op=0x03 & RD & RDsrc & RSsrc & simm16 {
    RD = RSsrc + (simm16 << 16);
}

:subi RD, RSsrc, simm16         is op=0x04 & RD & RSsrc & simm16 {
    RD = RSsrc - simm16;
}
```

Ghidra processor module: Sleigh (PCode)

- PCode defines the semantic of instruction
- Tables can emit pcode
- Ordering of emitted PCode described by `build` keyword. (we did not need it)
- No conditional emitting of PCode.
- All flow control is described by `call`/`goto’
- PCode gives us decompiler for almost free.

The Ghidra developers who made the Sleigh DSL were very clever.

iRISC: Ghidra Decompiler

```
00a7ede4 48 03 00 bc  csr.r    r3,zero,retaddr     r3 = CALLOTHER "RDCSR", retaddr
00a7ede8 6c 20 18 06  st.d     r3l,r1l,0x4         $U3180:4 = INT_ADD r1l, 4:4
                                                   STORE ram($U3180:4), r3l
00a7edec 70 3f 0f ea  enter    r1l,r1l,-0x18       $U3200:4 = INT_ADD r1l, 0xffffffe8:4
                                                   STORE ram($U3200:4), r1l
                                                   r1l = INT_ADD r1l, 0xffffffe8:4
00a7edf0 6c 20 a8 16  st.d     r21l,r1l,0x14       $U3180:4 = INT_ADD r1l, 20:4
                                                   STORE ram($U3180:4), r21l
00a7edf4 6c 20 b0 12  st.d     r22l,r1l,0x10       $U3180:4 = INT_ADD r1l, 16:4
                                                   STORE ram($U3180:4), r22l
00a7edf8 6c 20 b8 0e  st.d     r23l,r1l,0xc        $U3180:4 = INT_ADD r1l, 12:4
                                                   STORE ram($U3180:4), r23l
00a7edfc fc d7 30 08  mv       r23,size            r23 = COPY r6
00a7ee00 fc b6 28 08  mv       r22,data            r22 = COPY r5
00a7ee04 fc 95 20 08  mv       r21,st              r21 = COPY r4
00a7ee08 16 e4 00 00  subsi    st,r23,0x0          r4 = INT_SUB r23, 0:8
                                                   C64 = INT_LESS 0:8, r23
                                                   C32 = INT_LESS 0:4, r23l
                                                   r4 = INT_SUB 0:8, r23
                                                   N64 = INT_SLESS r4, 0:8
                                                   Z64 = INT_EQUAL r4, 0:8
                                                   N32 = INT_SLESS r4l, 0:4
                                                   Z32 = INT_EQUAL r4l, 0:4
00a7ee0c a0 00 00 15  b.t.e...  zero,LAB_00a7ee60  CBRANCH *[ram]0xa7ee60:8, Z32
```

```
1
2  void sha256_update(sha256_state *st,byte *data,int size)
3
4  {
5    undefined8 uVar1;
6    uint i;
7    undefined4 uStack00000004;
8
9    uVar1 = RDCSR(retaddr);
10   uStack00000004 = (undefined4)uVar1;
11   if (size != 0) {
12     i = st->datalen;
13     do {
14       st->data[i] = *data;
15       i = st->datalen + 1;
16       st->datalen = i;
17       if (i == 0x40) {
18         sha256_transform(st);
19         st->bitlen = st->bitlen + 0x200;
20         st->datalen = 0;
21         i = 0;
22       }
23       data = data + 1;
24       size = size + -1;
25     } while (size != 0);
26   }
27   return;
28 }
29
```

Firmware patching and code execution

- Linux kernel driver / sysfs debug command interface
- Patch target, what should we patch?
- FNP pin hack/Flash recovery mode => No signature checks

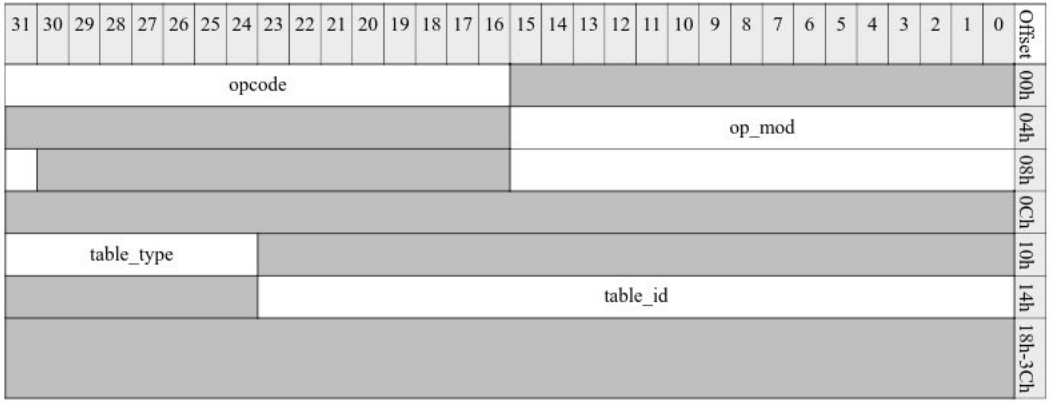
Firmware patching: Command interface

- NIC accepts commands over PCIe using DMA
- QUERY_FLOW_TABLE will be our patch target, linux kernel driver does not use it
- Some structure: opcode/op_mod
- What should we make the code do?

12.14.5 QUERY_FLOW_TABLE - Query Flow Table

The command returns a Flow Table context, as it was created by "CREATE_-FLOW_TABLE - Allocate a New Flow Table".

Table 401 - QUERY_FLOW_TABLE Input Structure Layout



Firmware patching: patching QUERY_FLOW_TABLE

- Read command data to stack
- Call data on stack
- Write response data from stack
- Fix some fields: seems important/wont work without
- Fixup all kinds of checksum errors everywhere in the firmware image :'(
- Lots of trial and error, much reflashing of the NIC

We have code execution on the NIC on demand

```
undefined4 cmdif_query_flow_table(toc *cmdif)

{
  uint uVar1;
  undefined8 uVar2;
  code *pcVar3;
  undefined4 uStack00000004;
  undefined4 auStack_d8 [8];
  undefined auStack_b8 [184];

  uVar2 = RDCSR(retaddr);
  uStack00000004 = (undefined4)uVar2;
  uVar1 = cmdif->gvmi;
  pcVar3 = (code *)cmdif->cmdif_io_cb;
                    /* Recv request */
  (*pcVar3)(1,uVar1,&cmdif->io_cb_state,0,0xc0,auStack_d8,0);
                    /* Set errorcode = 0x0 */
  auStack_d8[0] = 0;
                    /* Call shellcode */
  (*(code *)auStack_b8)(auStack_d8,cmdif);
                    /* Send response */
  (*pcVar3)(0,uVar1,&cmdif->io_cb_state,0,0xc0,auStack_d8,0);
  cmdif->missions = cmdif->missions & 0xffff0000 | 4;
  return 0;
}
```

Instruction set fuzzing

Plan:

- Generate experiment shellcode
- Send to NIC for execution
- Get response
- Analyse response
- Profit?!?

Result: We know a lot of opcodes, lots of details: Flags, data widths, edge cases, side effects, …

test_unki.asm

```
1   lbl entry
2       set64 r5, {{ r5 }}
3       set64 r6, 0
4       set64 r7, 0
5
6   lbl test
7       unk.i {{ opcode }}, r7, r5, {{ simm16 }}
8
9   lbl result
10      st.q r0, r4, r5, 0x08
11      st.q r0, r4, r6, 0x10
12      st.q r0, r4, r7, 0x18
13
14  lbl exit
15      ret.d
```

Userspace PCIe Driver

- Linux kernel driver is annoying
  - Often we break something
  - NIC stops responding to commands when we break something
  - Kernel hangs when NIC does not respond
  - Can’t Ctrl+C linux kernel, ain’t how a kernel works.
  - Kernel won’t even shutdown, as it can’t shutdown NIC nicely
  - We must yank power manually/echo b > /proc/sysrq-trigger
  - Really annoying. :(
  - Only 300exec/s
- This is a software problem, PCIe/thunderbolt is very SOLID
- Conclusion: We need a ‘Ctrl+C’-able driver in userspace
- PCIe/Driver interface/IOMMU/VFIO-PCI
- Rust Userspace driver: 30000exec/s very fast

Opensourcing it all!

- https://github.com/irisc-research-syndicate
  - mlx5cmd: userspace vfio/pcie driver for primarily for executing shellcode
  - irisc-asm: Basic assembler for the iRISC architecture, templating support
  - ghidra-processor: A ghidra module for decompilation support

Come by our tent to get a hand on introduction for working with iRISC and ghidra!

We would like to talk with some NVIDIA/Mellanox people!

Future work?

- Possible exploits (Packet parsing, VM escape, kernel driver bug?)
- Newer generations of ConnectX NICs (TLS offloading?)
- Other vendors (Broadcom, Intel, …)
- NICs/WIFI/Switches are interesting targets as they are directly accessible over the network, and often are OS independent.