

# Reverse engineering a SmartNIC

NVIDIA/Mellanox ConnectX-5

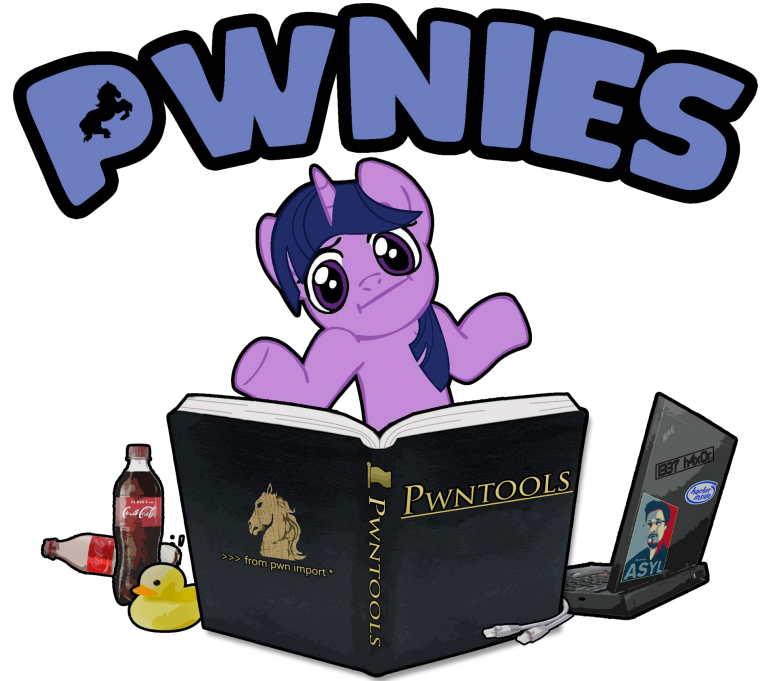
# The plan

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?

```
user@argon: ~/cx5/sections
user@argon:~/cx5/sections$ head -c 256 00007000_0001c494_IRON_PREP_CODE | phd
00000000 48 03 00 bc 6c 20 18 06 70 3f 0f e2 6c 20 98 1e | H... 1... p?... 1...
00000010 6c 20 a0 1a 6c 20 a8 16 6c 20 b0 12 6c 20 b8 0e | l... 1... 1... 1...
00000020 fd 57 50 08 fd 36 48 08 fd 15 40 08 fc f4 38 08 | .WP. .6H. .@. .8.
00000030 fc d3 30 08 00 06 00 01 14 a7 00 01 a0 01 00 05 | .0. ....
00000040 4a 06 00 02 14 c7 00 00 a0 00 00 12 fc c6 20 0a | J... ....
00000050 4a 04 00 03 14 c6 00 00 a0 00 00 0e 2c 86 00 ff | J... .... ,...
00000060 fc a5 30 05 a0 02 00 0b fc 84 80 83 14 85 00 00 | .0. ....
00000070 a0 00 00 02 94 00 3e 22 fe 64 98 08 fe 85 a0 08 | .... .>" .d...
00000080 fe a6 a8 08 fe c7 b0 08 fe e8 b8 08 94 00 00 b8 | ....
00000090 64 37 00 0e 64 36 00 12 64 35 00 16 64 34 00 1a | d7.. d6.. d5.. d4..
000000a0 64 33 00 1e 00 21 00 20 64 23 00 06 fd 00 18 25 | d3.. !. d#.. .%.
000000b0 48 03 00 bc 6c 20 18 06 70 3f 0f f2 6c 20 b0 0e | H... 1... p?... 1...
000000c0 6c 20 b8 0a fc f7 38 08 fc d6 30 08 00 06 00 01 | l... .8. .0. ....
000000d0 14 a7 00 01 a0 01 00 05 4a 06 00 02 14 c7 00 00 | .... .... J... ....
000000e0 a0 00 00 0f fc c6 20 0a 4a 04 00 03 14 c6 00 00 | .... .... J... ....
000000f0 a0 00 00 0b 2c 86 00 ff fc a5 30 05 a0 02 00 08 | .... ,... .0. ....
00000100
user@argon:~/cx5/sections$
```

# 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?

```
user@argon: ~/cx5/sections
user@argon:~/cx5/sections$ tail -c 260 00007000_0001c494_IRON_PREP_CODE | phd
00000000 42 8a 2f 98 71 37 44 91 b5 c0 fb cf e9 b5 db a5 | B / · q7D · ··· ···
00000010 39 56 c2 5b 59 f1 11 f1 92 3f 82 a4 ab 1c 5e d5 | 9V [ Y ··· ·?· ··^·
00000020 d8 07 aa 98 12 83 5b 01 24 31 85 be 55 0c 7d c3 | ··· ·· [ $1 ·· U } ··
00000030 72 be 5d 74 80 de b1 fe 9b dc 06 a7 c1 9b f1 74 | r ] t ··· ··· ··· t
00000040 e4 9b 69 c1 ef be 47 86 0f c1 9d c6 24 0c a1 cc | ·· i ·· G ··· $ ···
00000050 2d e9 2c 6f 4a 74 84 aa 5c b0 a9 dc 76 f9 88 da | ·· , o J t ·· \ ··· v ···
00000060 98 3e 51 52 a8 31 c6 6d b0 03 27 c8 bf 59 7f c7 | ·> QR ·1 m ··· ·Y ··
00000070 c6 e0 0b f3 d5 a7 91 47 06 ca 63 51 14 29 29 67 | ··· ··· G ··· cQ ·· ) g
00000080 27 b7 0a 85 2e 1b 21 38 4d 2c 6d fc 53 38 0d 13 | ! ··· ·· ! 8 M , m S 8 ··
00000090 65 0a 73 54 76 6a 0a bb 81 c2 c9 2e 92 72 2c 85 | e · s T v j ··· ··· ·r , ·
000000a0 a2 bf e8 a1 a8 1a 66 4b c2 4b 8b 70 c7 6c 51 a3 | ··· ·· f k ·K · p ·l Q ·
000000b0 d1 92 e8 19 d6 99 06 24 f4 0e 35 85 10 6a a0 70 | ··· ··· $ ·· 5 ·· j · p
000000c0 19 a4 c1 16 1e 37 6c 08 27 48 77 4c 34 b0 bc b5 | ··· ·· 7 l · ' H w L 4 ···
000000d0 39 1c 0c b3 4e d8 aa 4a 5b 9c ca 4f 68 2e 6f f3 | 9 ··· N ·· J [ ·· 0 h . o ·
000000e0 74 8f 82 ee 78 a5 63 6f 84 c8 78 14 8c c7 02 08 | t ··· x · co ·· x ···
000000f0 90 be ff fa a4 50 6c eb be f9 a3 f7 c6 71 78 f2 | ··· · P l ··· ··· q x ·
00000100 00 72 3f 5c | ·r ? \
00000104
user@argon:~/cx5/sections$
```

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/comparisons

Can we find the SHA256 implementation in firmware?

```
user@argon: ~/cx5/sections
user@argon:~/cx5/sections$ tail -c 260 00007000_0001c494_IRON_PREP_CODE | phd
00000000 42 8a 2f 98 71 37 44 91 b5 c0 fb cf e9 b5 db a5 | B / · q7D · ··· ···
00000010 39 56 c2 5b 59 f1 11 f1 92 3f 82 a4 ab 1c 5e d5 | 9V [ Y ··· ·?· ··^·
00000020 d8 07 aa 98 12 83 5b 01 24 31 85 be 55 0c 7d c3 | ··· ·· [ · $1· ·U}·
00000030 72 be 5d 74 80 de b1 fe 9b dc 06 a7 c1 9b f1 74 | r ]t ··· ··· ··t
00000040 e4 9b 69 c1 ef be 47 86 0f c1 9d c6 24 0c a1 cc | ··i· ··G· ··· $·
00000050 2d e9 2c 6f 4a 74 84 aa 5c b0 a9 dc 76 f9 88 da | ··,o Jt· ·\·· v··
00000060 98 3e 51 52 a8 31 c6 6d b0 03 27 c8 bf 59 7f c7 | ·>QR ·1·m ··· ·Y·
00000070 c6 e0 0b f3 d5 a7 91 47 06 ca 63 51 14 29 29 67 | ··· ···G· ··cQ ·)·)g
00000080 27 b7 0a 85 2e 1b 21 38 4d 2c 6d fc 53 38 0d 13 | '··· ··!8 M,m S8·
00000090 65 0a 73 54 76 6a 0a bb 81 c2 c9 2e 92 72 2c 85 | e·sT vj· ··· ·r·
000000a0 a2 bf e8 a1 a8 1a 66 4b c2 4b 8b 70 c7 6c 51 a3 | ··· ··fk ·K·p ·lQ·
000000b0 d1 92 e8 19 d6 99 06 24 f4 0e 35 85 10 6a a0 70 | ··· ··$ ··5 ··j·p
000000c0 19 a4 c1 16 1e 37 6c 08 27 48 77 4c 34 b0 bc b5 | ··· ·7l ·'HwL 4·
000000d0 39 1c 0c b3 4e d8 aa 4a 5b 9c ca 4f 68 2e 6f f3 | 9·· N·J [·0 h.o·
000000e0 74 8f 82 ee 78 a5 63 6f 84 c8 78 14 8c c7 02 08 | t·· x·co ··x· ···
000000f0 90 be ff fa a4 50 6c eb be f9 a3 f7 c6 71 78 f2 | ··· ·Pl· ··· ·qx·
00000100 00 72 3f 5c | ·r?\
00000104
user@argon:~/cx5/sections$
```

# 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.

```
user@argon: ~/cx5/presentation
sha256:
000130cc:      480300bc      unk.12 3, 0, 0xbc
000130d0:      6c201806      st.d 3, 1, 0x6
000130d4:      703f0ec2      unk.1c 31, 1, 0xec2
000130d8:      6c20b13e      st.d 22, 1, 0x13e
000130dc:      6c20b93a      st.d 23, 1, 0x13a
000130e0:      fcd73008      unk.3f 23, 6, 0x3008
000130e4:      fca62808      unk.3f 6, 5, 0x2808
000130e8:      fc852008      unk.3f 5, 4, 0x2008
000130ec:      2004ae85      unk.08 4, 0, 0xae85      <- sha256 init consts
000130f0:      2484bb67      unk.09 4, 4, 0xbb67
000130f4:      1c84e667      unk.07 4, 4, 0xe667
000130f8:      18846a09      unk.06 4, 4, 0x6a09      <- 4 * 16bits = 64 bits?
000130fc:      7820211c      unk.1e 0, 1, 0x211c      <- store 64 bits?
... skip 3 more blocks of constants ...
0001313c:      78200114      unk.1e 0, 1, 0x114      <- store bitlen? (64bits?)
00013140:      6c20010a      st.d 0, 1, 0x10a      <- store datalen
00013144:      00360008      unk.00 22, 1, 0x08
00013148:      fec4b008      unk.3f 4, 22, 0xb008
0001314c:      94fffeeb      unk.25 31, 7, 0xfeeb      <- ??? sha256_update? calls?
00013150:      fec4b008      unk.3f 4, 22, 0xb008
00013154:      fee5b808      unk.3f 5, 23, 0xb808
00013158:      94ffff96      unk.25 31, 7, 0xff96      <- ??? sha256_finalize?
0001315c:      6437013a      ld.d 23, 1, 0x13a
00013160:      6436013e      ld.d 22, 1, 0x13e
00013164:      00210140      unk.00 1, 1, 0x140
00013168:      64230006      ld.d 3, 1, 0x6
0001316c:      fd001825      unk.3f 0, 8, 0x1825

15,1 All
```

# What we know so far: iRISC instruction layout

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

ALU REG/IMM

OPCODE 6 bits	RS 5 Btis	RD 5 bits	IMMEDIATE 16 bits	
------------------	--------------	--------------	----------------------	--

ALU REG/REG

OPCODE 6 bits	RS 5 bits	RD 5 bits	RT 5 bits	ALUOP 11 bits
------------------	--------------	--------------	--------------	------------------

JUMPS/CALLS

OPCODE 6 btis	JMPOP 2 bits	OFFSET 24 bits		
------------------	-----------------	-------------------	--	--

MEM LOADS

OPCODE 6 bits	RS 5 bits	RD 5 bits	OFFSET 14 bits	WIDTH 2 bits
------------------	--------------	--------------	-------------------	-----------------

MEM STORES

OPCODE 6 bits	RS 5 bits	OFFSET 5 btis	RT 5 bits	OFFSET 9 bits	WIDTH 2 bits
------------------	--------------	------------------	--------------	------------------	-----------------

# 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 zero l 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   r31,r11,0x4      $U3180:4 = INT_ADD r11, 4:4
                                STORE ram($U3180:4), r31
00a7edec 70 3f 0f ea  enter  r11,r11,-0x18    $U3200:4 = INT_ADD r11, 0xffffffe8:4
                                STORE ram($U3200:4), r11
                                r11 = INT_ADD r11, 0xffffffe8:4
00a7edf0 6c 20 a8 16  st.d   r211,r11,0x14  $U3180:4 = INT_ADD r11, 20:4
                                STORE ram($U3180:4), r211
00a7edf4 6c 20 b0 12  st.d   r221,r11,0x10  $U3180:4 = INT_ADD r11, 16:4
                                STORE ram($U3180:4), r221
00a7edf8 6c 20 b8 0e  st.d   r231,r11,0xc   $U3180:4 = INT_ADD r11, 12:4
                                STORE ram($U3180:4), r231
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, r231
                                r4 = INT_SUB 0:8, r23
                                N64 = INT_SLESS r4, 0:8
                                Z64 = INT_EQUAL r4, 0:8
                                N32 = INT_SLESS r41, 0:4
                                Z32 = INT_EQUAL r41, 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**

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	Offset
opcode																											00h					
																op_mod											04h					
																											08h					
																											0Ch					
table_type																											10h					
								table_id																			14h					
																											18h-3Ch					

# 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**

```
2 undefined4 cmdif_query_flow_table(toc *cmdif)
3
4 {
5     uint uVar1;
6     undefined8 uVar2;
7     code *pcVar3;
8     undefined4 uStack00000004;
9     undefined4 auStack_d8 [8];
10    undefined auStack_b8 [184];
11
12    uVar2 = RDCSR(retaddr);
13    uStack00000004 = (undefined4)uVar2;
14    uVar1 = cmdif->gvmi;
15    pcVar3 = (code *)cmdif->cmdif_io_cb;
16                /* Recv request */
17    (*pcVar3)(1,uVar1,&cmdif->io_cb_state,0,0xc0,auStack_d8,0);
18                /* Set errorcode = 0x0 */
19    auStack_d8[0] = 0;
20                /* Call shellcode */
21    (*(code *)auStack_b8)(auStack_d8,cmdif);
22                /* Send response */
23    (*pcVar3)(0,uVar1,&cmdif->io_cb_state,0,0xc0,auStack_d8,0);
24    cmdif->missions = cmdif->missions & 0xffff0000 | 4;
25    return 0;
26 }
27
```



# 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, ...

```
ASM 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.