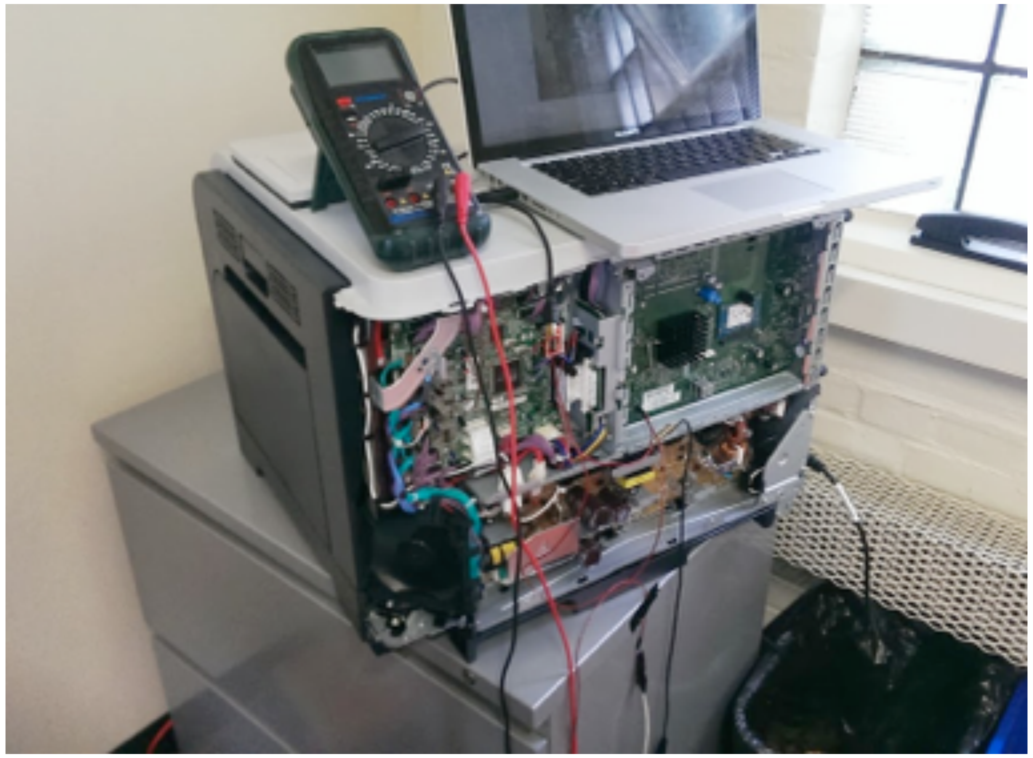
TANDON SCHOOL OF ENGINEERING

Challenges and Techniques for Emulating and Instrumenting Embedded Systems



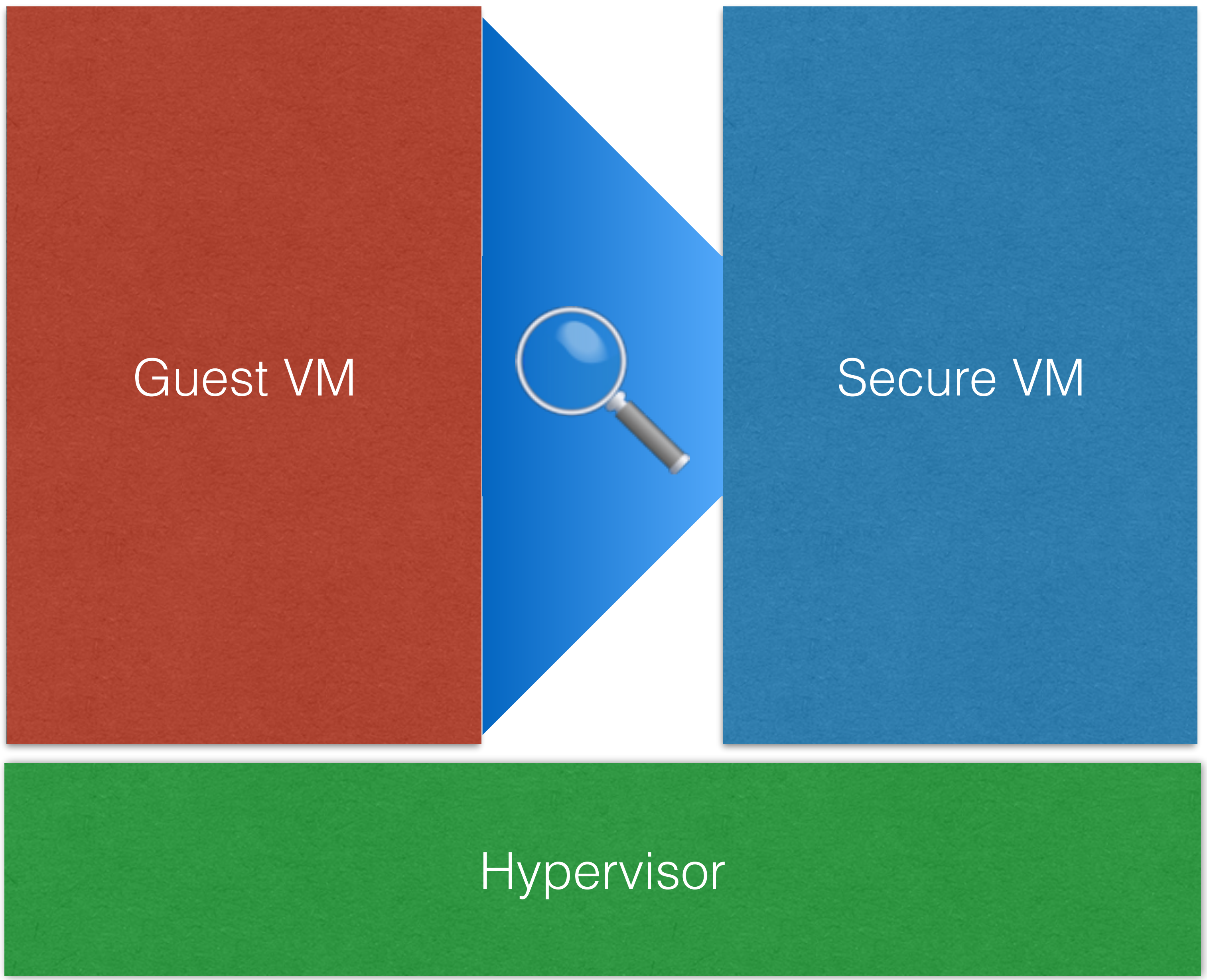
Brendan Dolan-Gavitt

NYU Tandon

Missing Capabilities

- The embedded world is sometimes thought of as just another computing platform
- However, unlike the desktop world, we currently lack key capabilities:
  - Secure virtualization – the ability to isolate security-relevant functionality in its own VM
  - Emulation – the ability to recreate a hardware platform in software

Virtualization Security



Embedded Security

- Currently, evaluation of security of embedded devices is almost entirely *manual*
- Extract firmware, open it in a disassembler, look for security flaws by hand
- In some cases, can use automated static analysis tools, but typically only for small portions of code or with access to source code

Dynamic Analysis for Security

- *Dynamic analysis*, i.e. analysis of code *in vivo* is critical to many kinds of security analysis:
  - Finding Vulnerabilities: fuzzing, concolic execution
  - Detecting Privacy Violations: dynamic taint analysis / information flow

Embedded Emulation

- We currently lack the ability to do dynamic analysis of embedded systems
- CPU support is there (e.g., QEMU), but each device has *embedded peripherals* that must be modeled and emulated
- Modeling requires painful manual reverse engineering

Why Emulation?

- **Scale**: we can test many thousands of virtual instances of devices, vs a relatively small number of physical devices
- **Safety**: emulated devices can be tested without fear of damaging expensive hardware
- **Instrumentation**: we can do much finer-grained instrumentation in a virtual environment, and even implement sophisticated features like taint analysis & record/replay
- **Flexibility & Convenience**: easy to test different configurations, software changes, etc.

Goals

- Create tools to *assist creation of embedded device peripheral models* in QEMU
- Try not to break the device!
  - We may only have one of them
  - This rules out some “invasive” techniques, like fuzzing the device directly
- If possible, avoid relying on things like JTAG that may be disabled or inaccessible in real targets

Assumptions

- Some basics are available:
  - CPU architecture known
  - Firmware available
  - Firmware load address / RAM location
- These are usually not too difficult to get manually – orthogonal to our work

Embedded Device I/O

- An embedded CPU communicates with its peripherals using 3 mechanisms:
  - Memory-mapped I/O
  - Interrupts
  - Direct Memory Access (DMA)

Idea: Infer Models from Execution

- We have code that queries the devices (i.e. device firmware)
- Giving the “wrong answers” to this code produces errors (i.e., an oracle)
- Therefore: can use code to infer correct answers to queries (i.e. device models)

Breadcrumbs

```
seg002:8005AAD0 70 40 2D E9                 STMFD   SP!, {R4-R6,LR}
seg002:8005AAD4 00 50 A0 E1                 MOV     R5, R0
seg002:8005AAD8 74 31 9F E5                 LDR     R3, =0xBF030100
seg002:8005AADC A0 10 95 E5                 LDR     R1, [R5,#0xA0]
seg002:8005AAE0 68 01 9F E5                 LDR     R0, =aRamSize0x08x ; "RAM Size=0x%08x\r\n"
seg002:8005AAE4 00 20 93 E5                 LDR     R2, [R3]
seg002:8005AAE8 FF EC A0 E3                 MOV     LR, #0xFF00
seg002:8005AAEC FF 30 8E E3                 ORR     R3, LR, #0xFF
seg002:8005AAF0 03 60 02 E0                 AND     R6, R2, R3
seg002:8005AAF4 D8 16 00 EB                 BL      DbgPrintf
seg002:8005AAF8 4C 01 9F E5                 LDR     R0, =aAsicid0x08x ; "asicID=0x%08x\r\n"
seg002:8005AAFC 06 10 A0 E1                 MOV     R1, R6
seg002:8005AB00 D5 16 00 EB                 BL      DbgPrintf
```

```
WindowsCE on ARM

ARM Cortex-A8 CPU
Stack Pointer: 0x00007ffc
Windows CE Kernel for ARM (Thumb Enabled) Built on Jan 26 2012 at 21:54:55
ProcessorType=0c08  Revision=1 CpuId=0x412fc081
OEMAddressTable = 8005923c
OEMInit (db)
Trace system activeRAM Size=0x40cbe000
asicID=0x00002600
Jedi Memory Pool: Size=0x2D000000 Start=0x12D88000
```

Breadcrumbs

```
seg002:8005AAD0 70 40 2D E9         STMFD   SP!, {R4-R6,LR}
seg002:8005AAD4 00 50 A0 E1         MOV     R5, R0
seg002:8005AAD8 74 31 9F E5         LDR     R3, =0xBF030100
seg002:8005AADC A0 10 95 E5         LDR     R1, [R5,#0xA0]
seg002:8005AAE0 68 01 9F E5         LDR     R0, =aRamSize0x08x ; "RAM Size=0x%08x\r\n"
seg002:8005AAE4 00 20 93 E5         LDR     R2, [R3]
seg002:8005AAE8 FF EC A0 E3         MOV     LR, #0xFF00
seg002:8005AAEC FF 30 8E E3         ORR     R3, LR, #0xFF
seg002:8005AAF0 03 60 02 E0         AND     R6, R2, R3
seg002:8005AAF4 D8 16 00 EB         BL      DbgPrintf
seg002:8005AAF8 4C 01 9F E5         LDR     R0, =aAsicid0x08x ; "asicID=0x%08x\r\n"
seg002:8005AAFC 06 10 A0 E1         MOV     R1, R6
seg002:8005AB00 D5 16 00 EB         BL      DbgPrintf
```

```
WindowsCE on ARM

ARM Cortex-A8 CPU
Stack Pointer: 0x00007ffc
Windows CE Kernel for ARM (Thumb Enabled) Built on Jan 26 2012 at 21:54:55
ProcessorType=0c08  Revision=1 CpuId=0x412fc081
OEMAddressTable = 8005923c
OEMInit (db)
Trace system activeRAM Size=0x40cbe000
asicID=0x00002600
Jedi Memory Pool: Size=0x2D000000 Start=0x12D88000
```

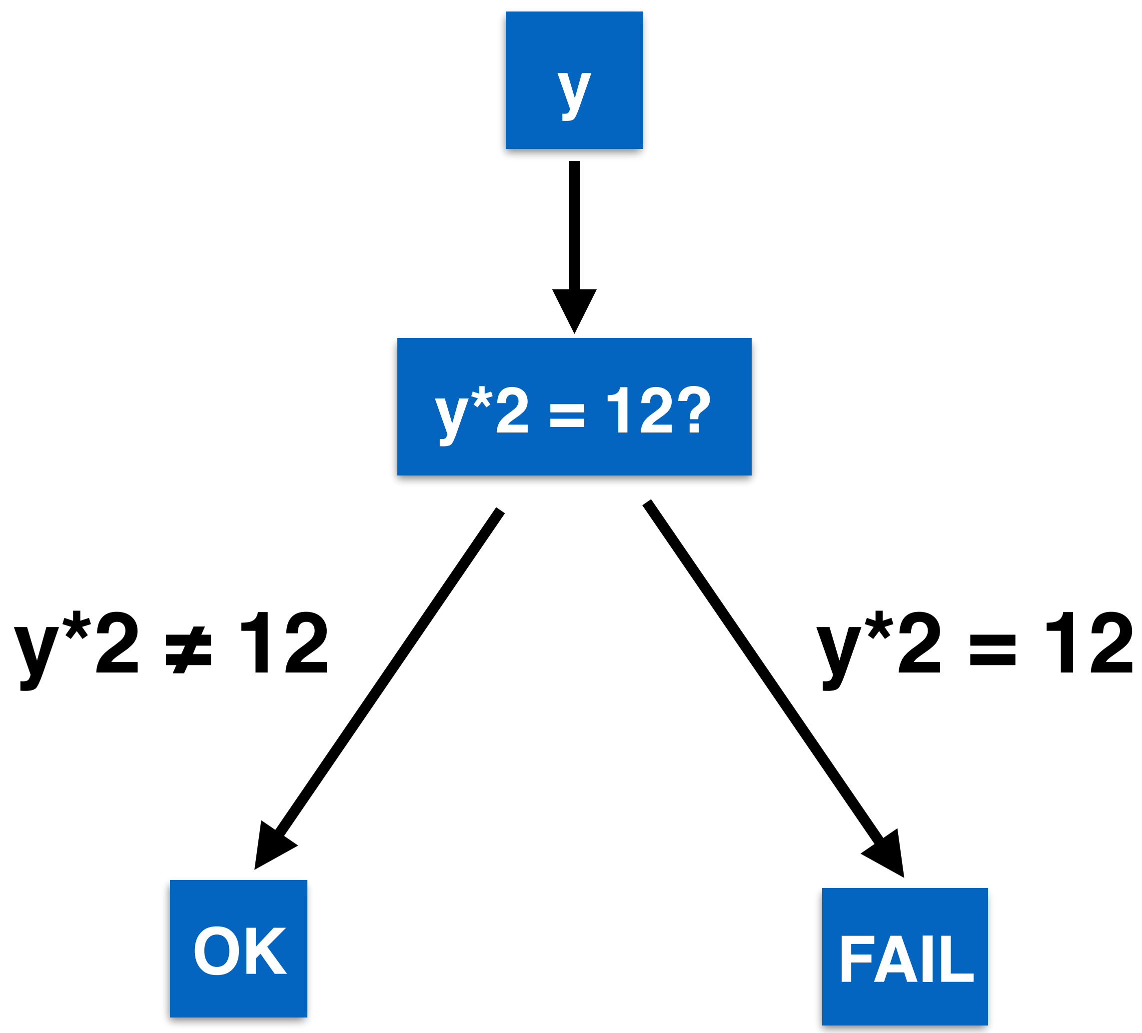
Breadcrumbs

```
seg002:8005AAD0 70 40 2D E9          STMFD   SP!, {R4-R6,LR}
seg002:8005AAD4 00 50 A0 E1          MOV     R5, R0
seg002:8005AAD8 74 31 9F E5          LDR     R3, =0xBF030100
seg002:8005AADC A0 10 95 E5          LDR     R1, [R5,#0xA0]
seg002:8005AAE0 68 01 9F E5          LDR     R0, =aRamSize0x08x ; "RAM Size=0x%08x\r\n"
seg002:8005AAE4 00 20 93 E5          LDR     R2, [R3]
seg002:8005AAE8 FF EC A0 E3          MOV     LR, #0xFF00
seg002:8005AAEC FF 30 8E E3          ORR     R3, LR, #0xFF
seg002:8005AAF0 03 60 02 E0          AND     R6, R2, R3
seg002:8005AAF4 D8 16 00 EB          BL      DbgPrintf
seg002:8005AAF8 4C 01 9F E5          LDR     R0, =aAsicid0x08x ; "asicID=0x%08x\r\n"
seg002:8005AAFC 06 10 A0 E1          MOV     R1, R6
seg002:8005AB00 D5 16 00 EB          BL      DbgPrintf
```

```
WindowsCE on ARM

ARM Cortex-A8 CPU
Stack Pointer: 0x00007ffc
Windows CE Kernel for ARM (Thumb Enabled) Built on Jan 26 2012 at 21:54:55
ProcessorType=0c08  Revision=1 CpuId=0x412fc081
OEMAddressTable = 8005923c
OEMInit (db)
Trace system activeRAM Size=0x40cbe000
asicID=0x00002600
Jedi Memory Pool: Size=0x2D000000 Start=0x12D88000
```

Symbolic Execution




```
y = read()
y = 2 * y
if (y == 12)
    fail()
print("OK")
```

OK => (y*2 ≠ 12) => y ≠ 6

FAIL => (y*2 = 12) => y = 6

NYU

Example

```
void OutputDebugChar(int c) {
    // Wait until serial port is ready
    while (ReadDword(0xbd370404) & 0x4 == 0) {
        // busy loop
    }
    WriteDword(0xbd370414, c);
}
```

Symbolic Input

Goal

```
IN:
0x0005cb08:  ldr   r0, [pc, #44]
0x0005cb0c:  bl    0x5cc24
----------------
IN:
0x0005cc24:  ldr   r0, [r0]
0x0005cc28:  bx    lr
----------------
IN:
0x0005cb10:  tst   r0, #4   ; 0x4
0x0005cb14:  beq   0x5cb08
```

```
=========== tcg-llvm-tb-40-5cb10 ===========
(Eq false
    (Eq 0
        (And w32 (ReadLSB w32 0 device memory @0xbd370404.0)
                 4)))
True:
device memory @0xbd370404.0: 04000000
```

Automatically Generated QEMU Devices


```
{
    "dev_0xbd370404" : {
        "description": "Automatically generated device at 0xbd370404",
        "base": 3174499328,
        "memory": [
            {"address": 3174499332, "values": [4], "size": 4}
        ]
    }
}
```

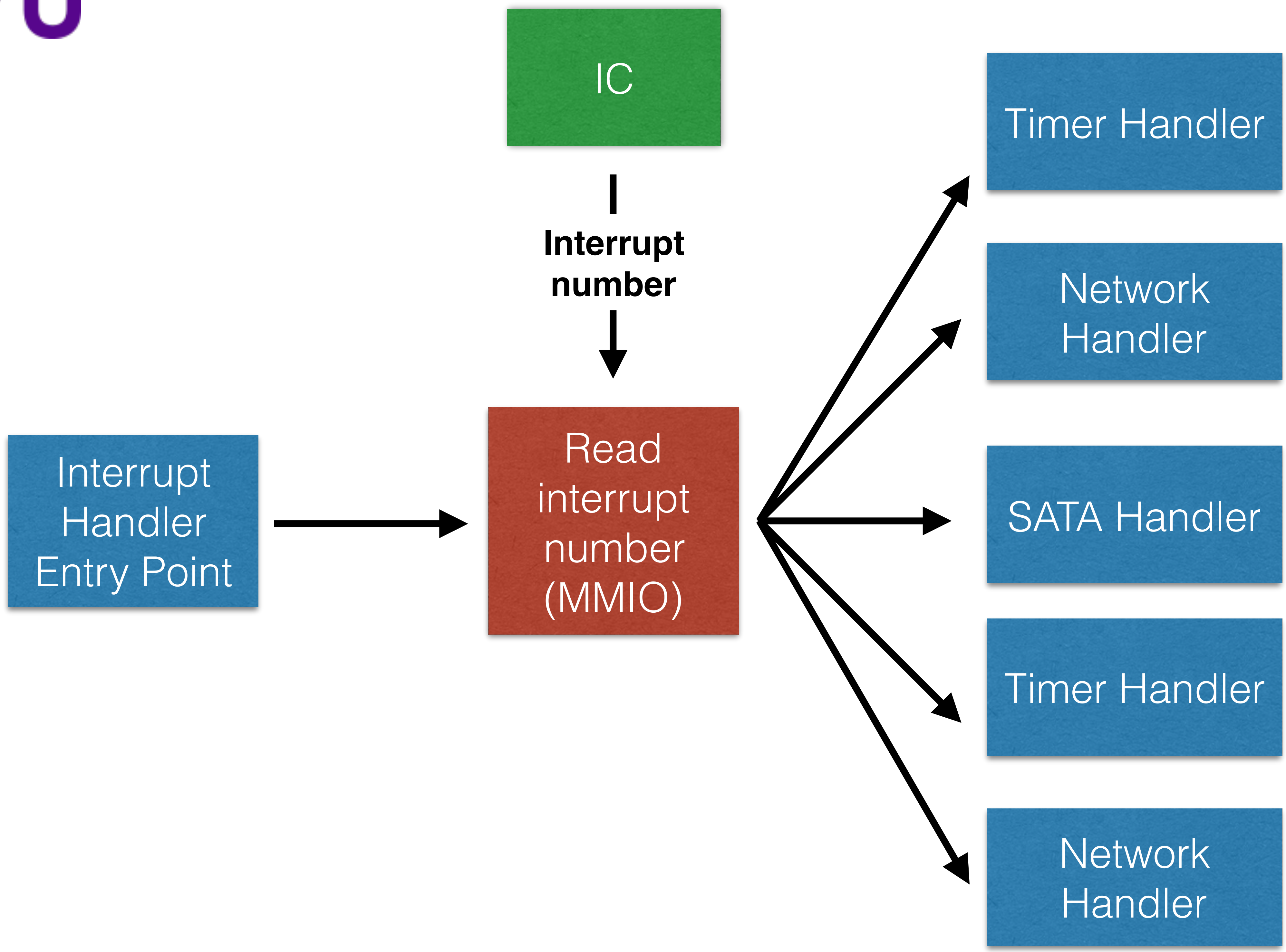
Limitations of MMIO Solving

- Only simple devices can be modeled this way – more complex devices may have complicated state
- Doesn't help uncover the *semantics* of peripherals, only what value is needed to continue
- Currently useful for devices like simple sensors, serial I/O, NVRAM / boot arguments

Understanding Interrupts

- At the CPU level, there is really only one "interrupt"
- When it fires, the interrupt controller is consulted to find out interrupt number
- OS then dispatches into specific interrupt handler that talks to the peripheral
- Interrupts drive execution:
  - **System timer** fires periodically to let scheduler run
  - **Network controller** interrupts when packet arrives

Interrupt Controller Code



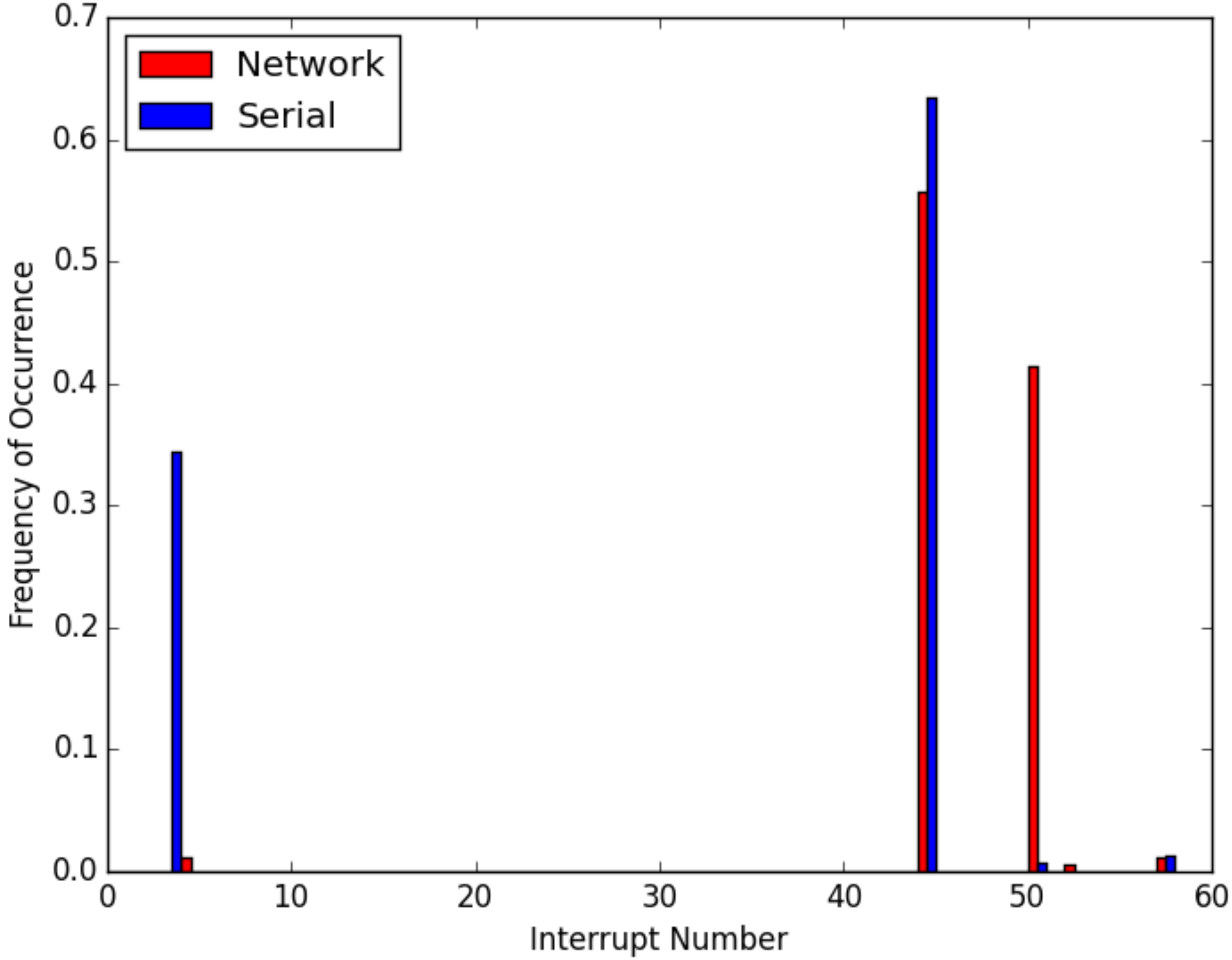
Mapping Interrupts Automatically

- Given this common structure we can *automatically* map out interrupts and associate each with the code for its handler
- Start symbolically executing at the *architecturally defined* interrupt handler
  - On ARM this is at 0xFFFF0018
- Look for **conditional branches** controlled by **device input**
- Ask solver to enumerate all possible targets controlled by that device input

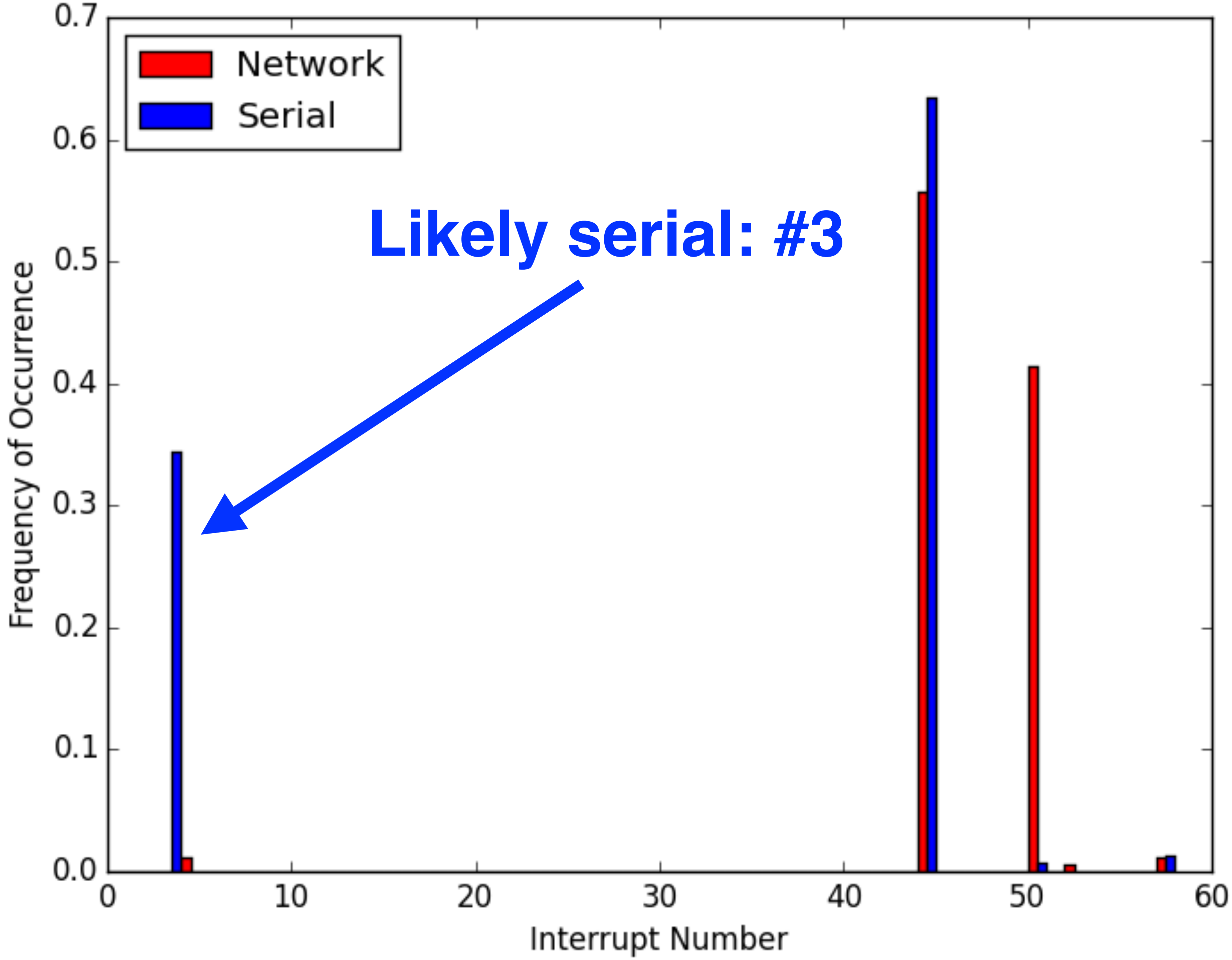
Observing Interrupts

- If we allow ourselves to modify firmware, we can log each interrupt, its number, and a timestamp
- By sending particular I/O workloads we can then associate interrupt numbers to actual peripherals
- Hint: the interrupt that fires every 100ms is probably the timer!

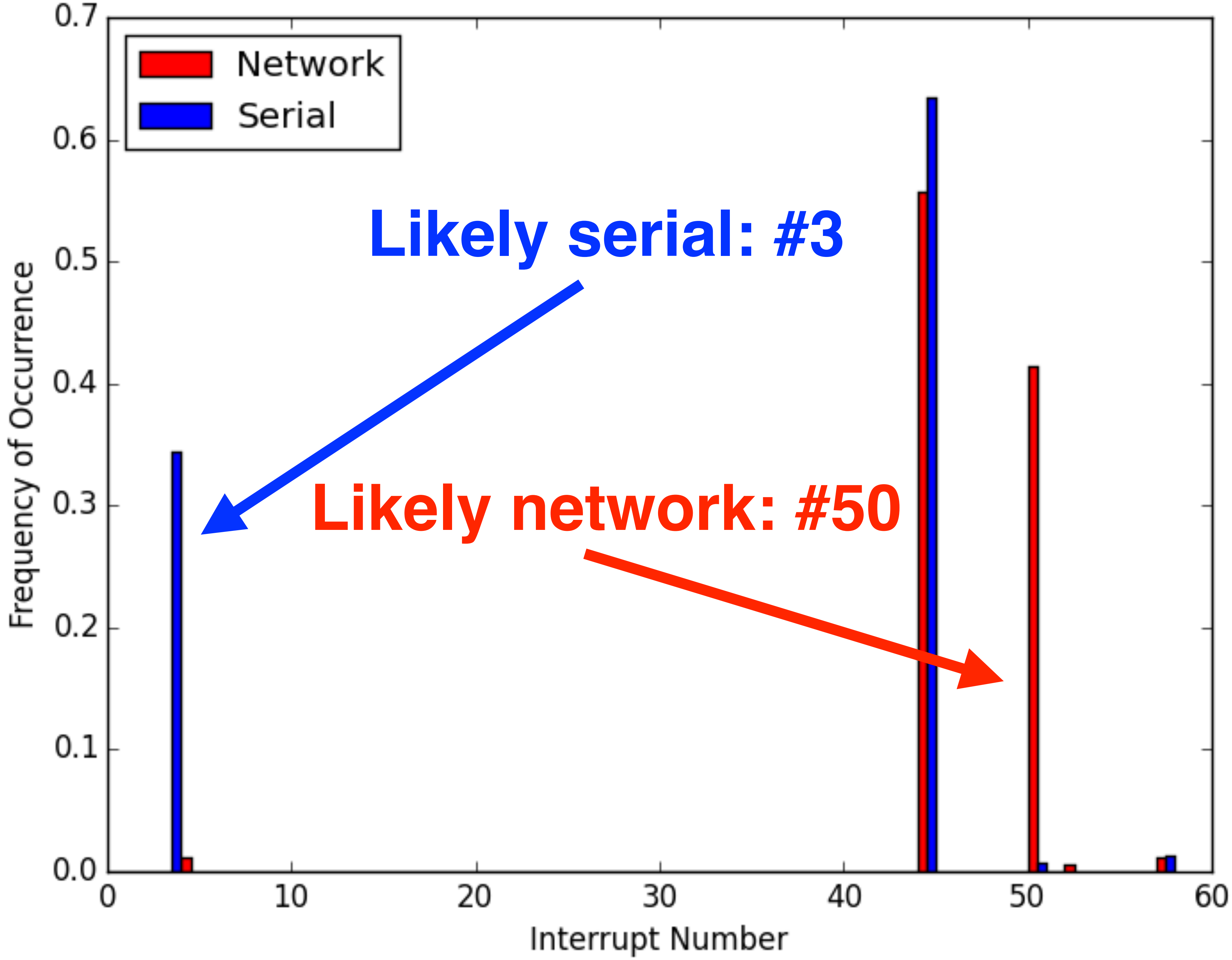
Observing Interrupts



Observing Interrupts

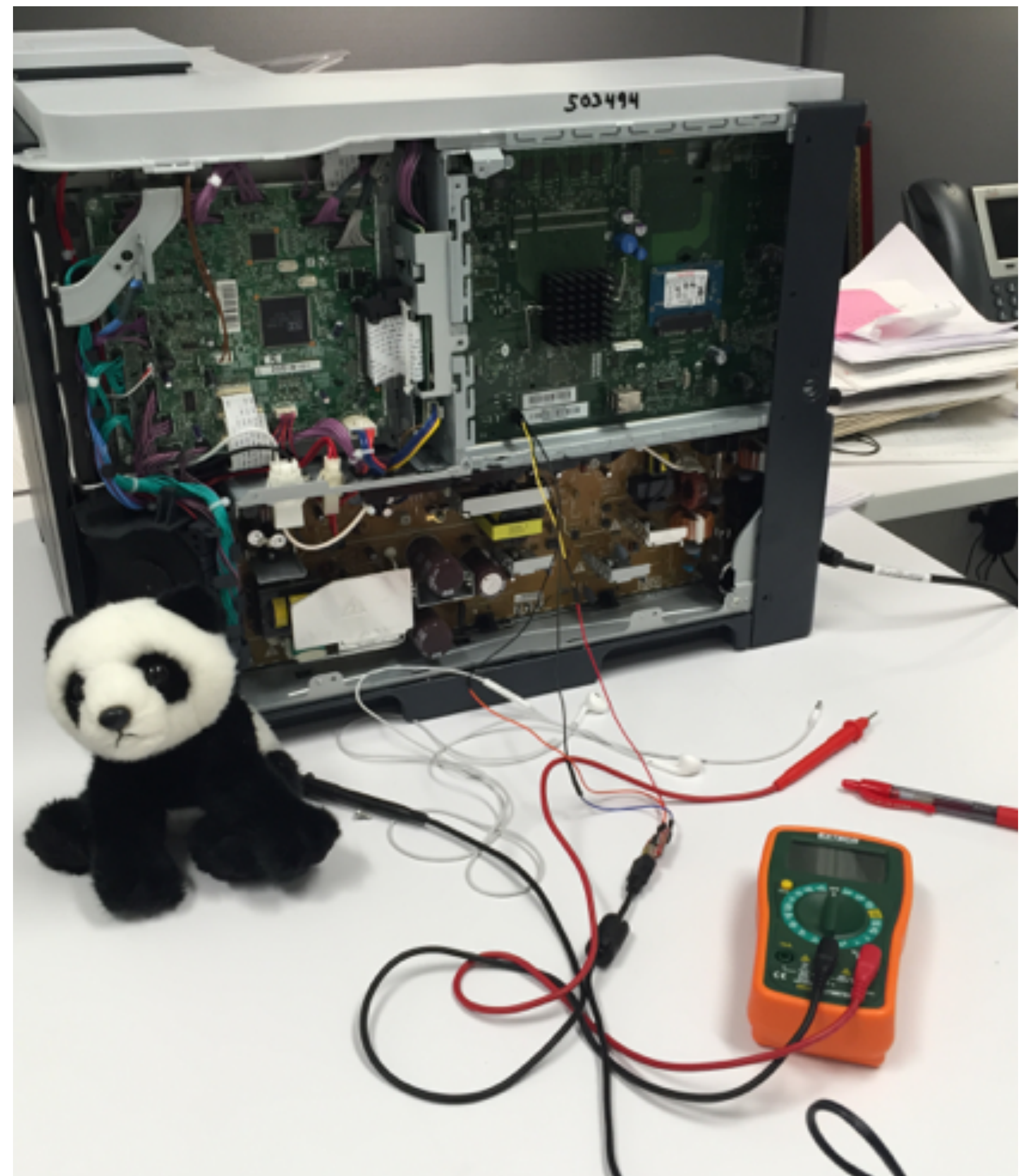


Observing Interrupts



HP Printer Firmware

- Our current model system: **HP m551dn**
- High-end color laserjet printer
- **Hardware**: ARM Cortex A8, 1GB RAM, USB, PCIe, SATA SSD, Broadcom Ethernet
- **Software**: Windows CE 6.0
  - With an emulated VxWorks userland library???



HP Printer Firmware

```
WindowsCE on ARM

ARM Cortex-A8 CPU
Stack Pointer: 0x00007ffc
Windows CE Kernel for ARM (Thumb Enabled) Built on Jan 26 2012 at 21:54:55
ProcessorType=0c08  Revision=0 CpuId=0x410fc080
OEMAddressTable = 8005923c
OEMInit (db)
Trace system activeRAM Size=0x40cbe000
asicID=0x00002600
Jedi Memory Pool: Size=0x2D000000 Start=0x12D88000
Adjusting Jedi Memory Pool for Acpi=0x3fd5e000 Memory Pool Start=0x12D88000
[...]
BOOT CHECKPOINT: START APP FAILED: \Windows\HP.Platform.Services.PartitionManager.exe 0x00980001
BOOT CHECKPOINT: BOOT COMPLETE
BOOT CHECKPOINT: SHELL BOOT COMPLETE
ErrorDialog::OnInitDialog
ErrorDialog::OnInitDialog width = 800, height = 600
```

HP Printer Firmware

We can boot!

```
WindowsCE on ARM

ARM Cortex-A8 CPU
Stack Pointer: 0x00007ffc
Windows CE Kernel for ARM (Thumb Enabled) Built on Jan 26 2012 at 21:54:55
ProcessorType=0c08  Revision=0 CpuId=0x410fc080
OEMAddressTable = 8005923c
OEMInit (db)
Trace system activeRAM Size=0x40cbe000
asicID=0x00002600
Jedi Memory Pool: Size=0x2D000000 Start=0x12D88000
Adjusting Jedi Memory Pool for Acpi=0x3fd5e000 Memory Pool Start=0x12D88000
[...]
BOOT CHECKPOINT: START APP FAILED: \Windows\HP.Platform.Services.PartitionManager.exe 0x00980001
BOOT CHECKPOINT: BOOT COMPLETE
BOOT CHECKPOINT: SHELL BOOT COMPLETE
ErrorDialog::OnInitDialog
ErrorDialog::OnInitDialog width = 800, height = 600
```

HP Printer Firmware

We can boot!

```
WindowsCE on ARM

ARM Cortex-A8 CPU
Stack Pointer: 0x00007ffc
Windows CE Kernel for ARM (Thumb Enabled) Built on Jan 26 2012 at 21:54:55
ProcessorType=0c08  Revision=0 CpuId=0x410fc080
OEMAddressTable = 8005923c
OEMInit (db)
Trace system activeRAM Size=0x4[illegible]cbe000
asicID=0x00002600
Jedi Memory Pool: Size=0x2D[illegible]00000 Start=0x12D88000
Adjusting Jedi Memory Pool for Acpi=0x3fd5e000 Memory Pool Start=0x12D88000
[...]
BOOT CHECKPOINT: START APP FAILED: \Windows\HP.Platform.Services.PartitionManager.exe 0x00980001
BOOT CHECKPOINT: BOOT COMPLETE
BOOT CHECKPOINT: SHELL BOOT COMPLETE
ErrorDialog::OnInitDialog
ErrorDialog::OnInitDialog width = 800, height = 600
```

...but missing USB, SATA, network, ...

Instrumentation

- Many of the techniques we've already looked at would be enhanced by improved *instrumentation*
- Similarly, there are many security techniques (e.g., virtualization security) that rely on secure instrumentation
- How can we achieve this on an embedded system?

The Problem of Virtualization

- Many embedded devices do not have support for hardware virtualization
    - E.g., ARM virtualization standard exists but not widely implemented
- Classic "trap and emulate" and para-virtualization often require changes to the guest OS
- Neither of these are suitable for trying to instrument already-deployed devices

Dynamic Binary Instrumentation

- One possible solution: *dynamic binary instrumentation*
  - Essentially a JIT for binary code – can be very efficient since most instructions do not need to be rewritten
- Create a *small* DBI implementation that can be run on the device itself to perform arbitrary instrumentation
- Worth noting: this is the strategy VMWare used to virtualize x86

Open Questions

- ***Fidelity***: how can we ensure the inferred model faithfully represents real hardware?
- ***Completeness***: how do we know when we have covered all hardware behavior?
- ***Physical Effects***: how can we incorporate physical constraints on the behavior of hardware into our automatically generated models?

Conclusions

- Full emulation is a necessary baseline capability for efficiently testing embedded and industrial control systems
- Peripheral modeling is difficult, but symbolic execution can cover some simple cases
- Further research is needed to fully automate emulation of embedded systems – particularly in the area of efficient and secure instrumentation