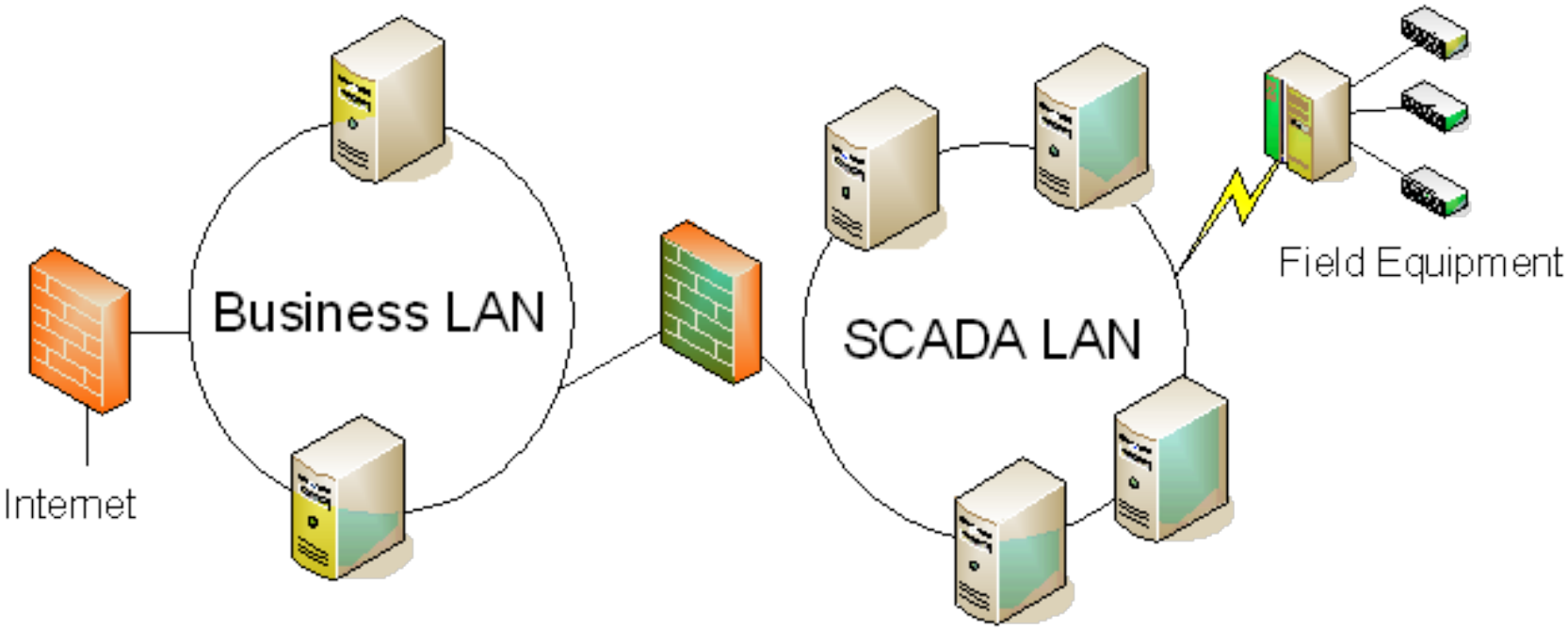
CyberSecurity of Field Equipment of SCADA

Jason Larsen
TCIPG 2012

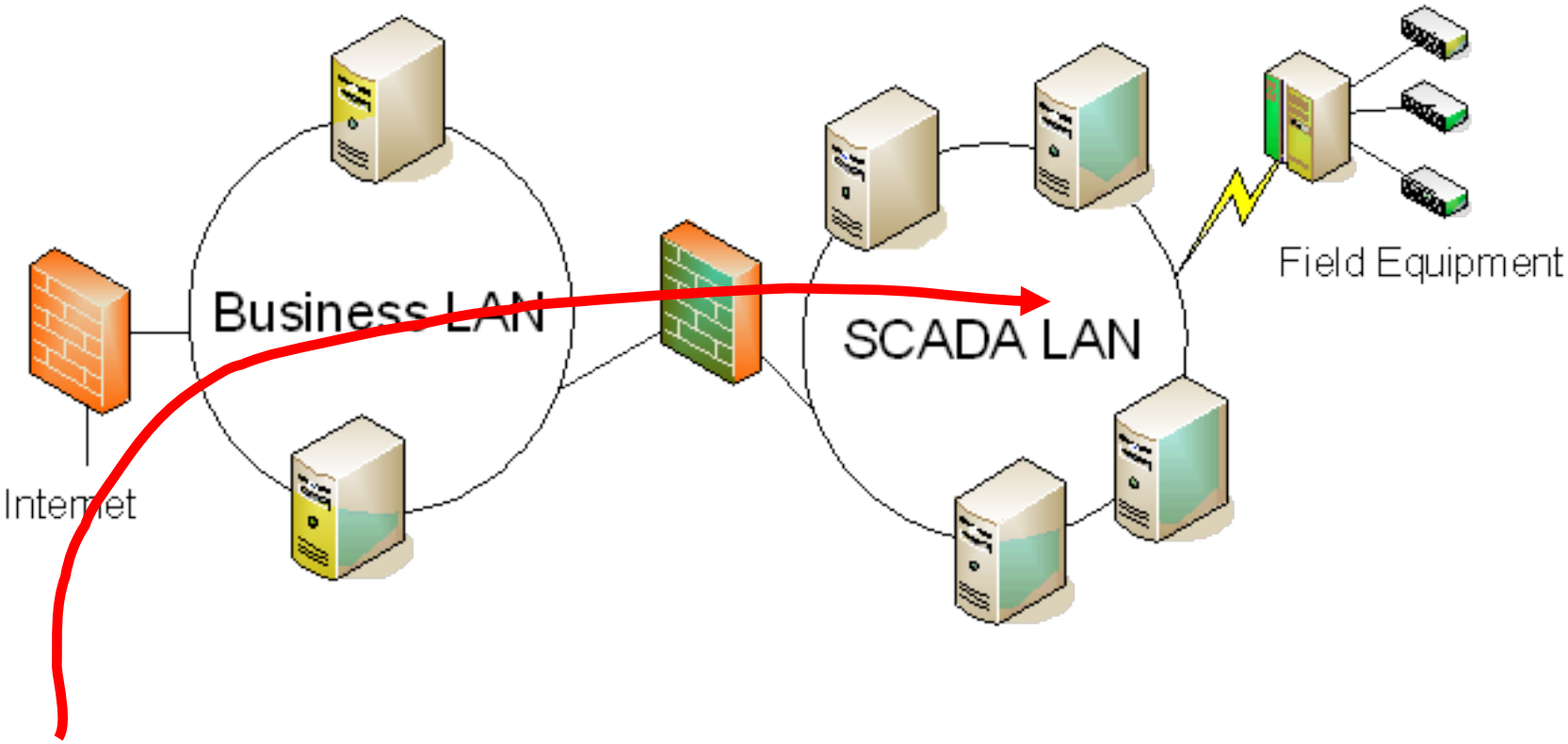
Who Am I?

- Jason Larsen
  - CyberSecurity Researcher at the Idaho National Labs
  - Specialize in the technical attacks on critical infrastructure
- Warning: Bits and Bytes Guy
  - If I get too far in the weeds. Give me a heads up.

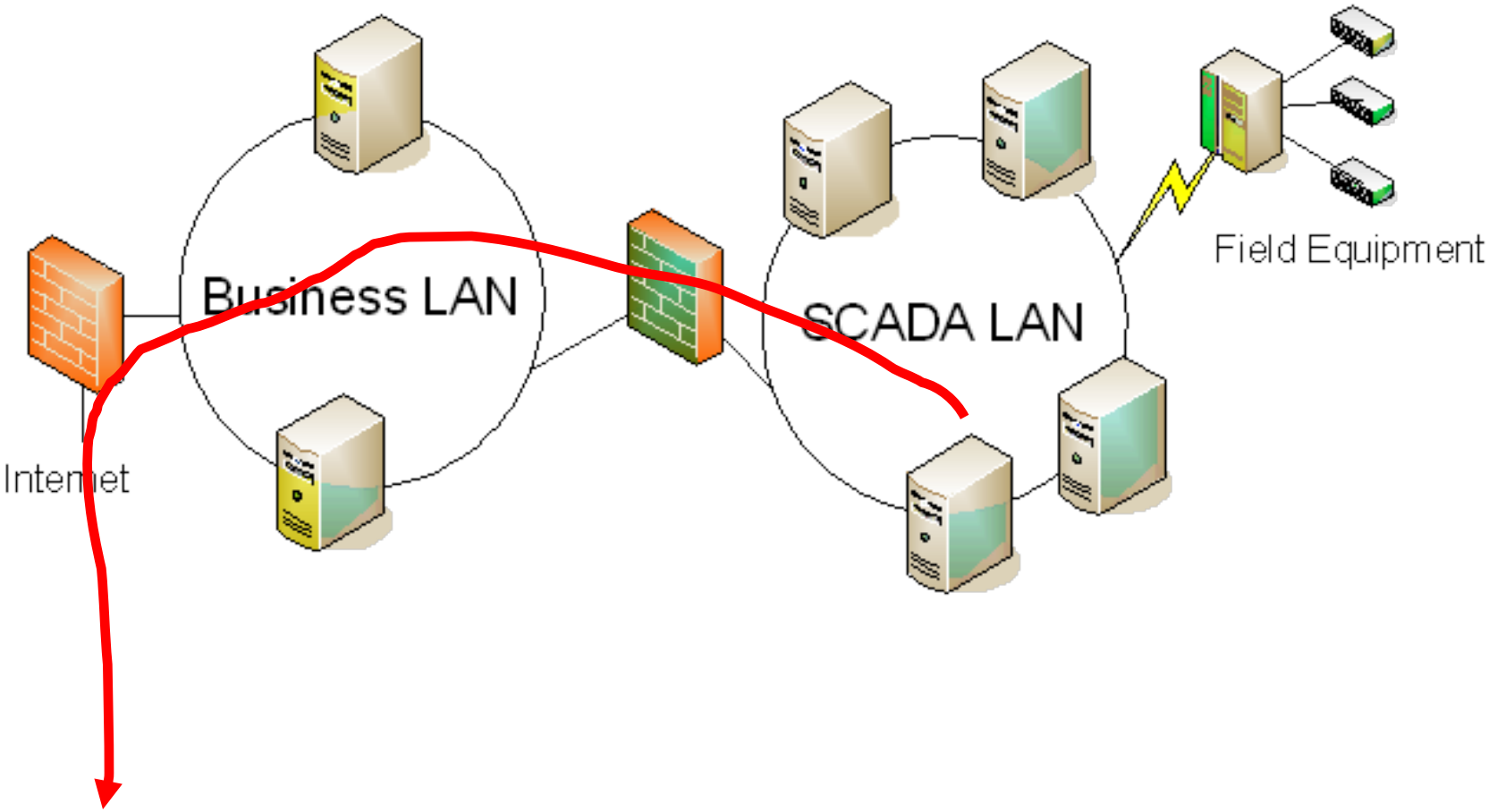
Traditional SCADA Attack Pathways



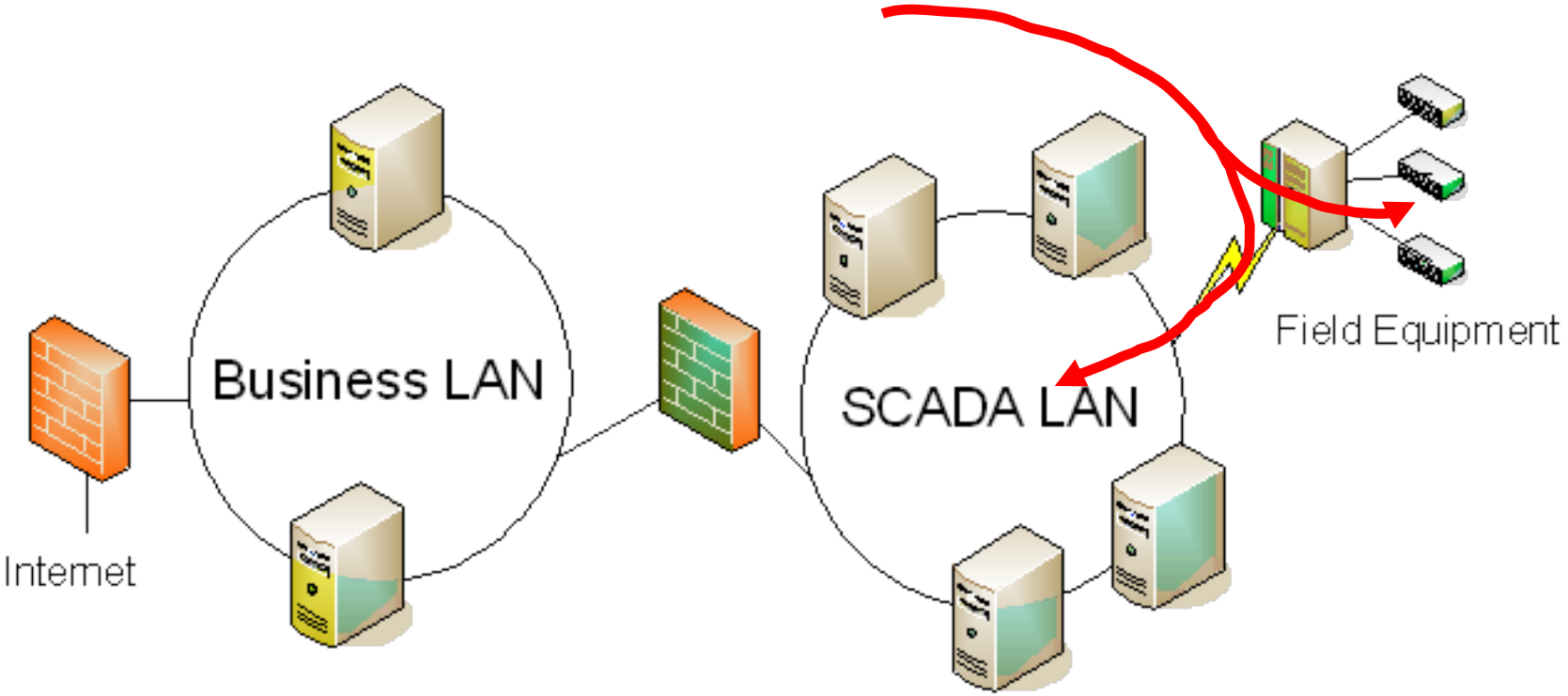
Full Frontal Assault



USB Key/Supply Chain/Vendors/Anti-Virus



Field Communications



The data you need

- Most attackers will need to compromise the main SCADA network
- A map of the real-world equipment isn't contained in the SCADA system
- Where is the data they need?
  - Operator Screen Files
  - Realtime Database
  - Change Management and Maintenance Orders

Why Attack Field Equipment?

- Bypass sanity checks
  - Logic in the field equipment may not allow the equipment to be driven to failure
  - Intermediate equipment may enforce interlocks or other semantics
- The attacker can physically touch the field equipment
  - SMART Grid
  - Devices outside the facility
- Interconnections lead to field equipment
  - Weather sensors
  - Badge readers
  - Cameras
  - Door Alarms

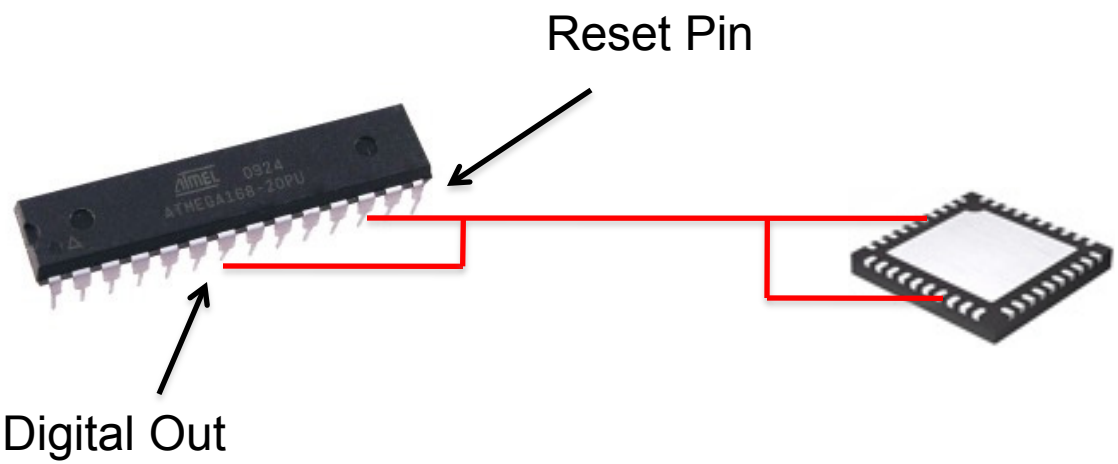
Let's hack like it's 1999

- Lack of a kernel
- No memory protection
- No virtual address space means no address space randomization
- VxWorks, Phar Lap, Green Hills, eCos,…
  - Not really an "operating system" in the computer sciencey definition

Problems: Reboot-y-ness

- Embedded systems mantra “If all else fails, reboot yourself”
- Watchdog Timers
- Common Reset Lines
- The attacker must not let the victim reboot during exploitation
- Shellcode cannot take over the CPU

Common Reset Lines



"Dave, I'm invoking the suicide pact"

Rewriting From scratch

- I’ve tried to rewrite firmware from scratch
- I don’t think this is possible even with an almost unlimited budget
  - The new firmware never quite behaves like the original
  - Embedded engineers don’t fix bugs as much as work around them
- In most cases attackers must modify and not rewrite

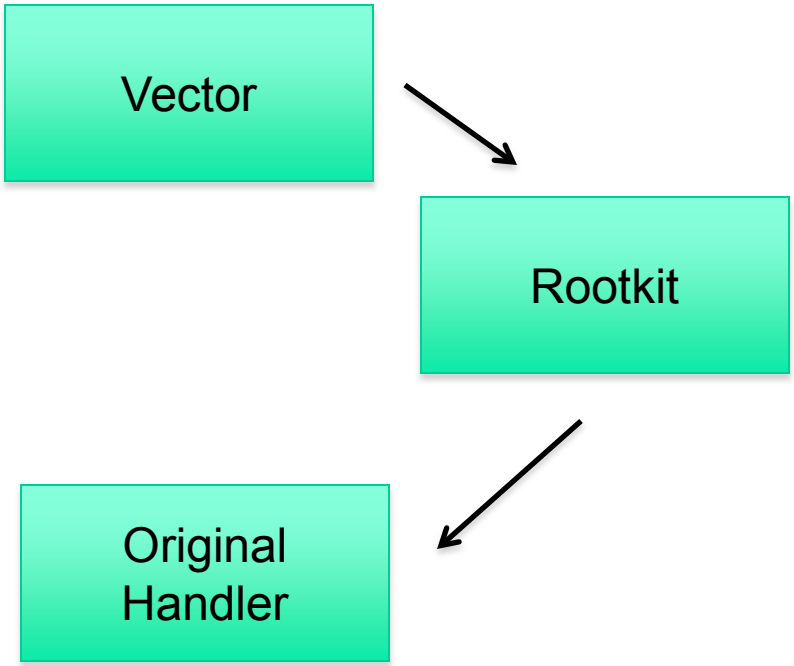
Problems:Firmware Patches

- How do I give my rootkit CPU cycles?
  - Vector Tables
  - Hooking the Main Loop

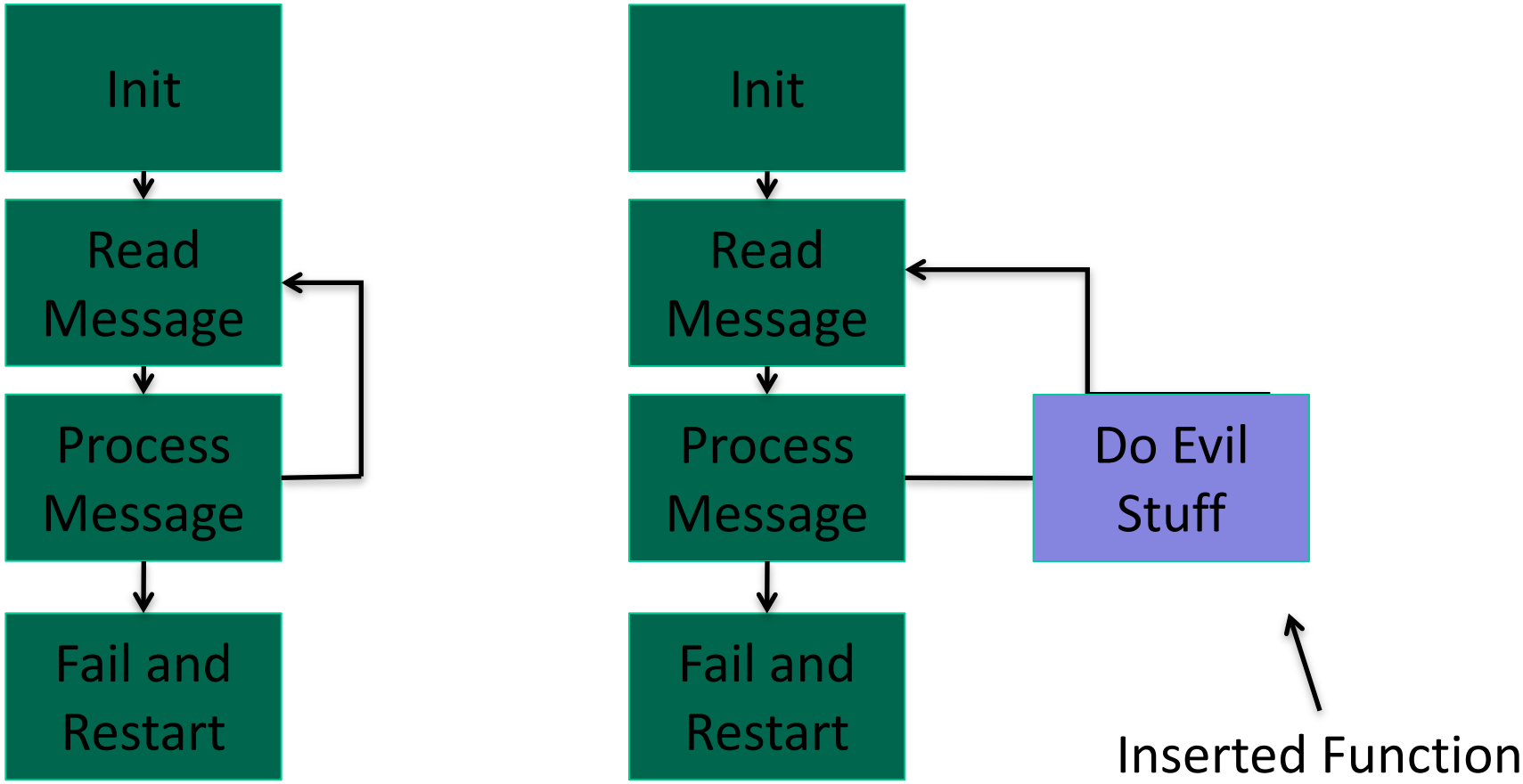
Vector Tables

- In most microcontrollers the Vector table is mapped at the top or bottom of memory

0x0000	Reset
0x0002	Timer Tick
0x0004	Math Exception
0x0008	PortA



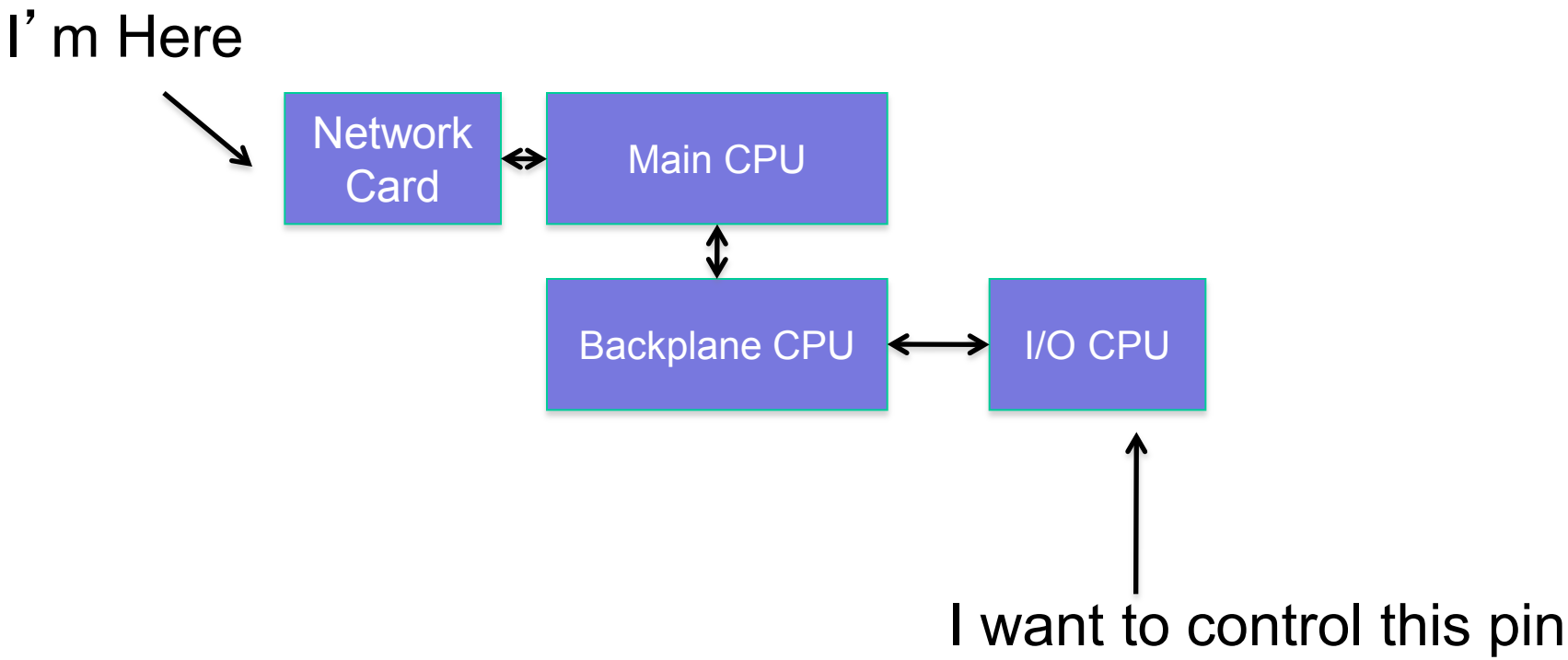
Hooking the Main Loop



Problems:Inter-Micro Communications

- Writing shellcode for serial connections is just like writing findsockshellcode
- Serial ports are opened via an open() call
- File descriptor equivalents may be walked to find the serial port
- UARTs can be accessed via memory-mapped I/O

Inter-Micro Communications

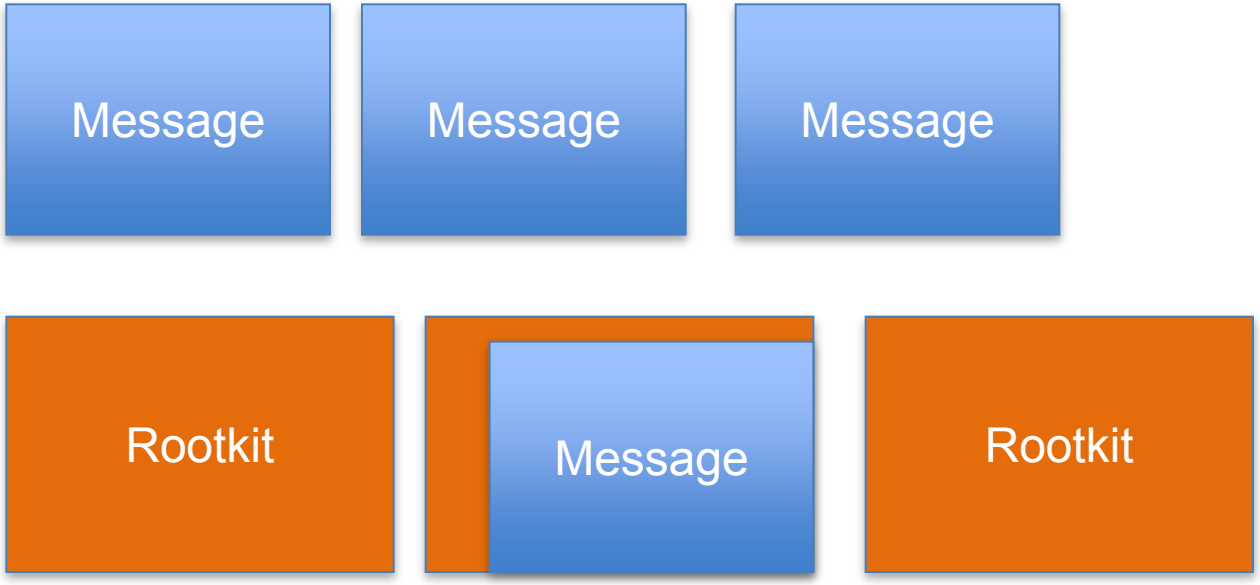


Exploiting Serial Connections

- There's a problem
- If the attacker takes over the serial connection, the serial port doesn't perform its normal function
  - See Reboot-y-ness
- The purpose of a serial connection must be preserved for long-term control
- Both sides must transition to the rootkit protocol at the same time

Exploiting Serial Connections

- Original Messages must be tunneled over the new rootkit protocol



Communications Protocol

- Protocol Interleaving

Communications Protocols

- A powerful attacker could do better
- Rudimentary control channel

16 17 Magic
00 12 Length
04 Message Type
00 04 Data Blob Length
01 02 03 04 Data Blob

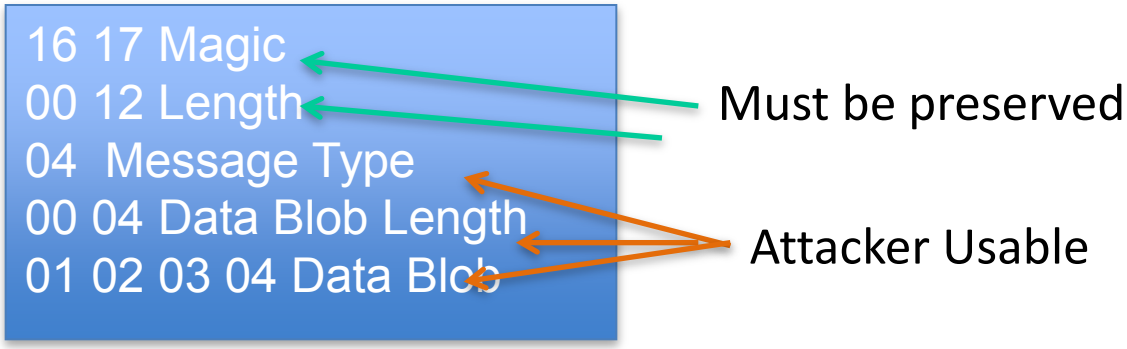
16 17 Magic
00 12 Length
FF Attacker Message
00 06 Data Blob Length
reboot

Communications Protocols

- Certain artifacts must be maintained
  - Otherwise reboot/reinitialize/error
- If the attacker doesn't want to spend every waking moment re-coding communications channels, he'll need something that can be used in multiple protocols

Communications Protocols

- Redundancy coding can preserve the artifacts of the serial protocol while supplying arbitrary data in an efficient manner
- Complex algorithms don't bulk up the rootkit



Communications Protocols

- Message + Offsets -> Coding
- Coding schemes make detection by analyzers or heuristic IDSs unlikely

Reboot

16 17 Magic
00 12 Length
3e Message Type
70 71 Data Blob Length
03 d9 24 97 21 39 da 70
cc 03 a3 0b 0f a0 82

Problem:Where do I put the rootkit?

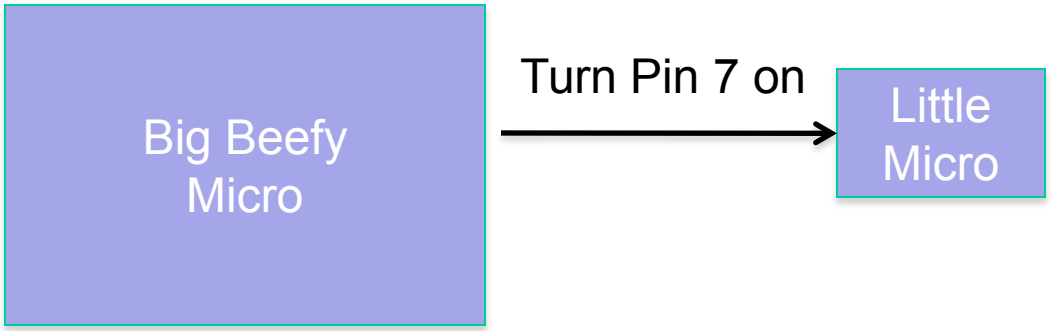
- Most vendors see extra flash as wasted revenue
  - That flash could have been used for a feature that would have sold more units
- Feature creep
  - Why do I need a web server on my nic?
- Since the flash is full, the attacker must make room for his functionality

Where do I put my rootkit?

- Lazy - Slaving a microcontroller
- Basic – Dump functionality
- Intermediate – Local Optimizations and Compression Sleds
- Advanced – Refactoring the Firmware

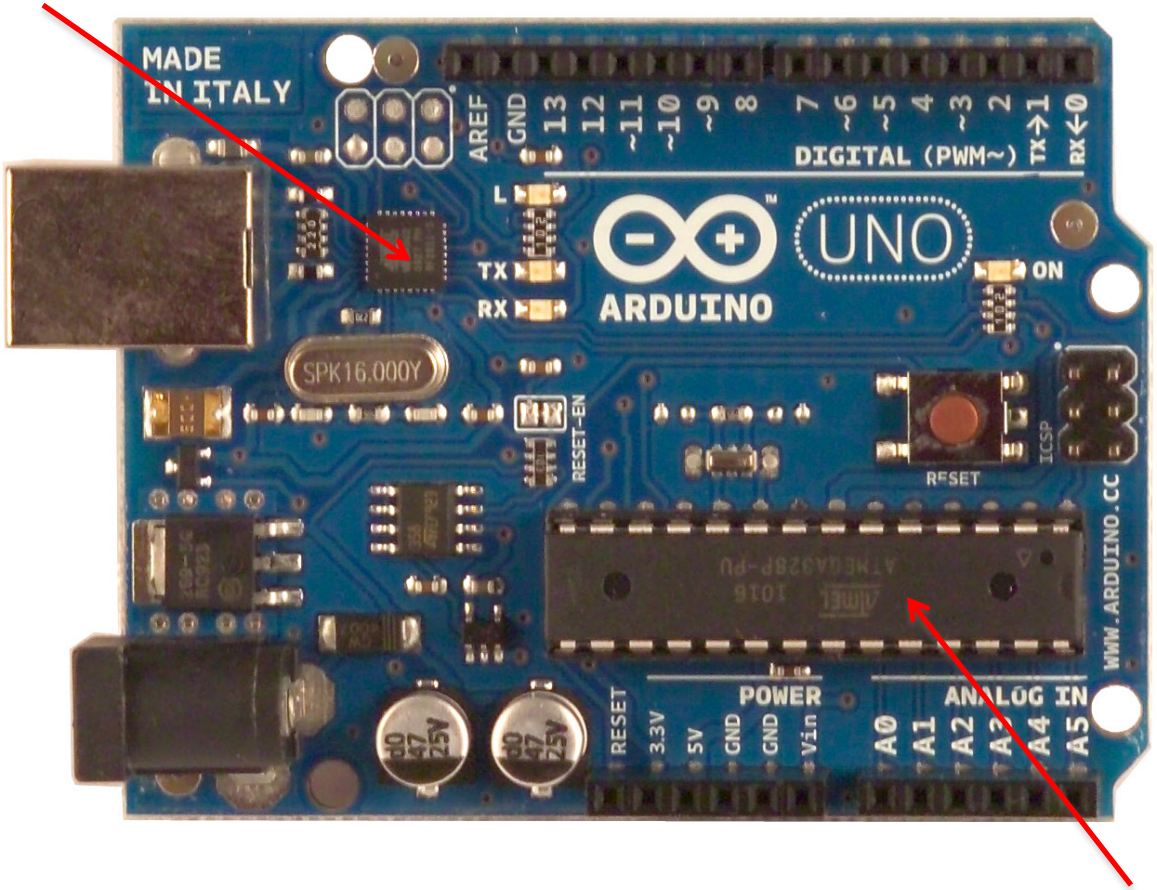
Lazy - Slaving the microcontroller

- For the lazy and unskilled
- If two microcontrollers are connected and the one needed for control doesn't have enough space, it can be lobotomized and run from an adjacent microcontroller
- This approach works well in motherboard rootkits



Uno Example

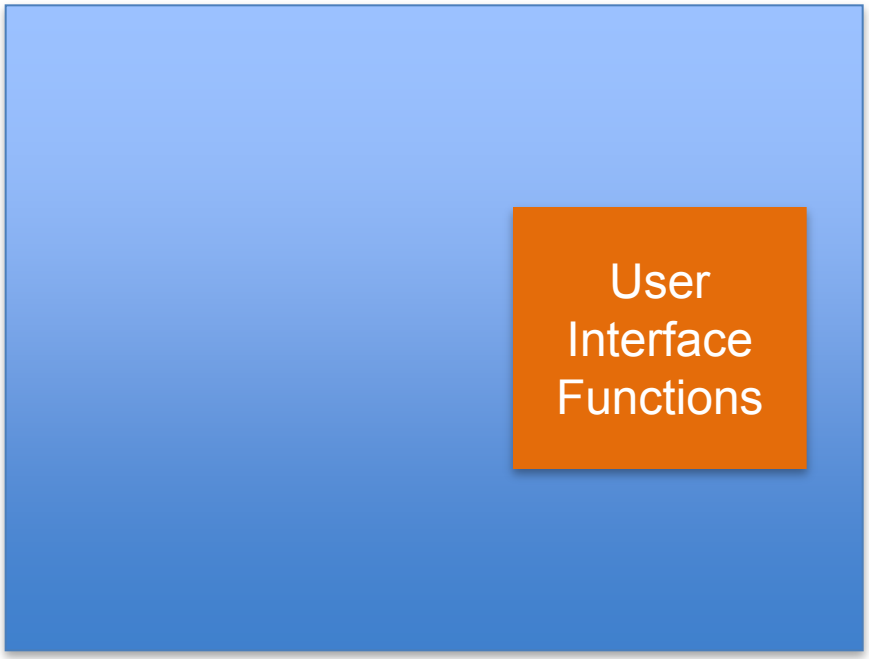
USB Processor (4k Flash)



Main Processor (32k Flash)

Basic Approach

- The attacker can dump functionality
- Little-used functions can be replaced with the rootkit



Basic Approach

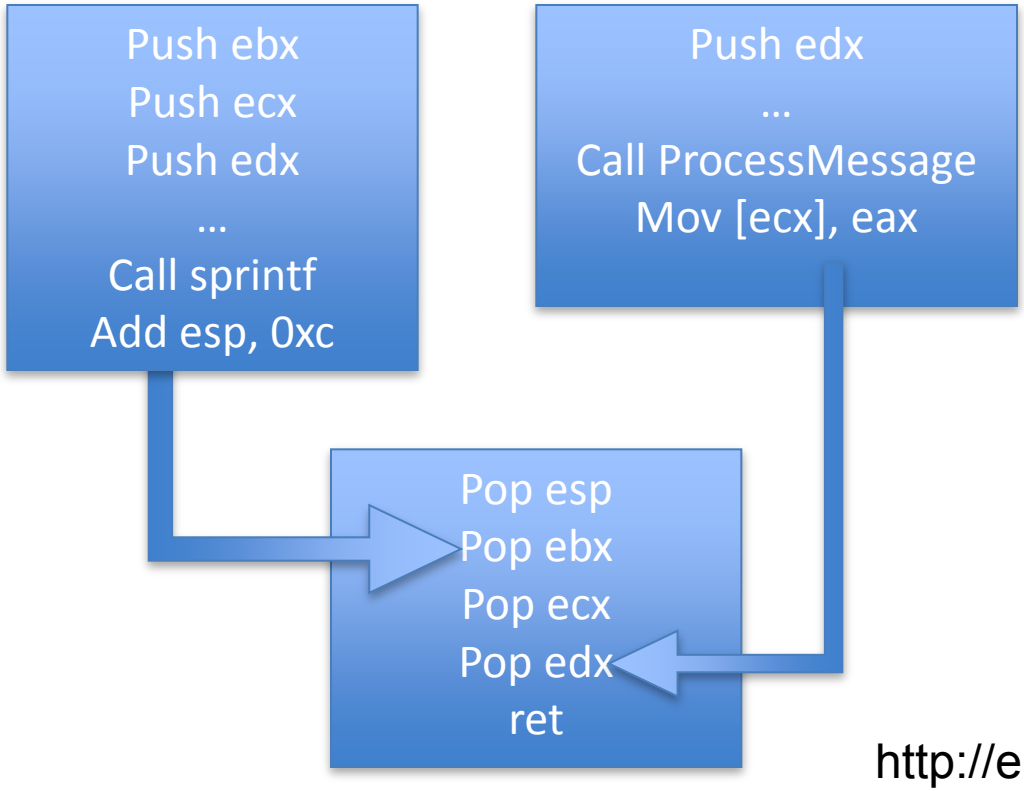
- Flash is flash
- Attacker can replace human-only data
  - String tables
  - images

Intermediate Approach Local Compression

- Most compilers do a poor job of inter-function optimization
- Since the assembly is poorly optimized, gaps can be created in the image while preserving total functionality

Intermediate Approach

- Function Tailing – A simple inter-function optimization



http://en.wikipedia.org/wiki/Tail_call

Intermediate Approach

- Greatest common size match with repeats
  - Room for computer-science-y algorithms

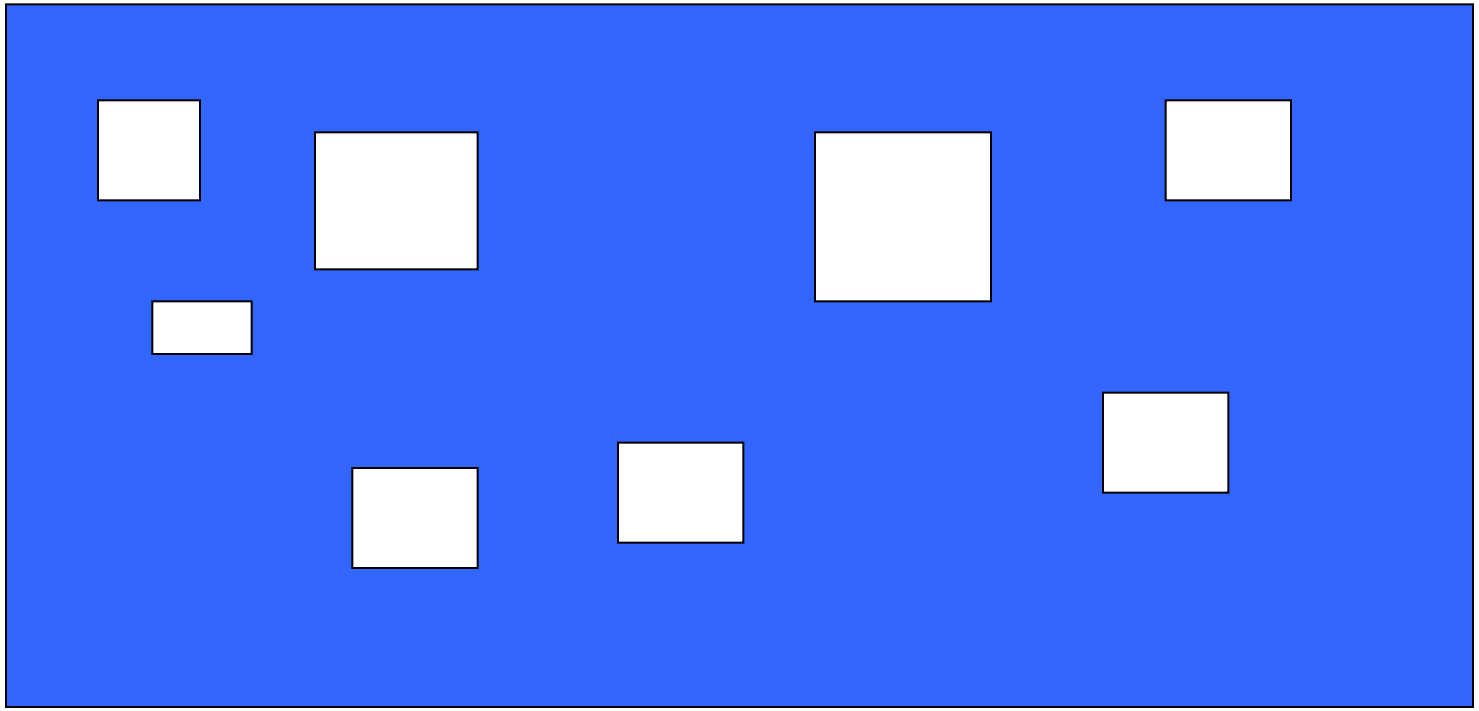
```
Mov r1, 5
Mov r2, r3
Cmp r2, r4
Bne loop
Add r4, 1
Mov [r6], r6
Mul r6, 4, r6
```

```
Mov r1, 5
Mov r2, r3
Cmp r2, r4
Bne loop
Add r4, 1
Sub r5, 1
Call DoSomething
```

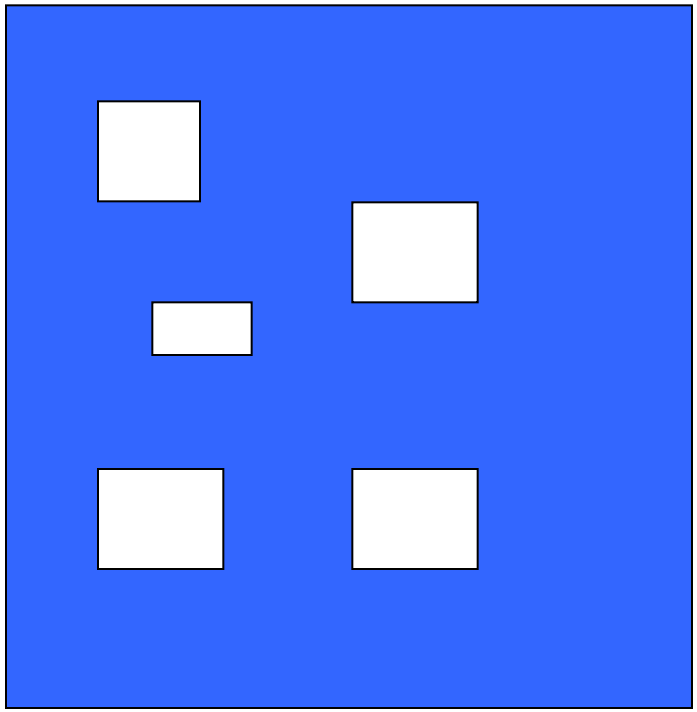
```
Call CommonCode
Mov [r6], r6
Mul r6, 4, r6
```

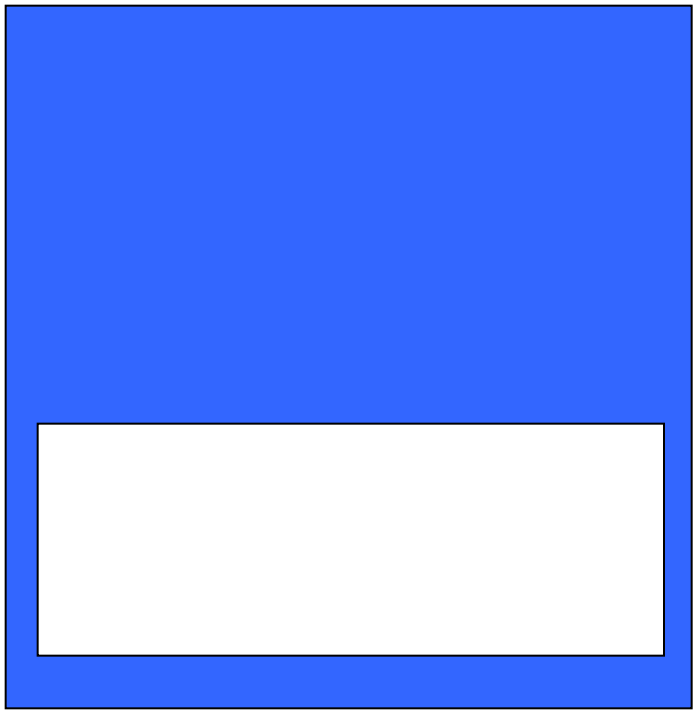
Intermediate Approach

- The compression leaves an image that looks like swiss cheese
  - Can the attacker take advantage of this?



Advanced Approach Refactoring Firmware



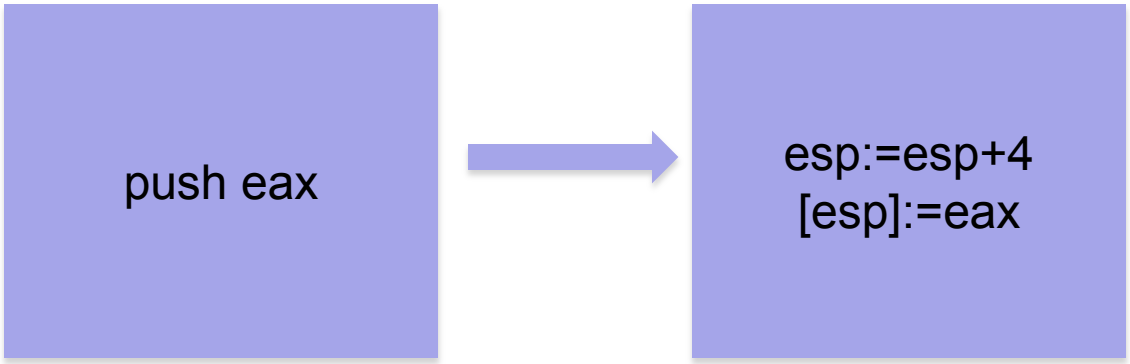


Advanced approach

- Instructions can be decomposed into micro-operations
  - Must describe all the side effects of the assembly
- Micro-op stream can be optimized independent of assembly
- Referred to as binary refactoring
- Binary -> micro-ops -> Binary
  - Guaranteed to preserve functionality

Micro-Ops

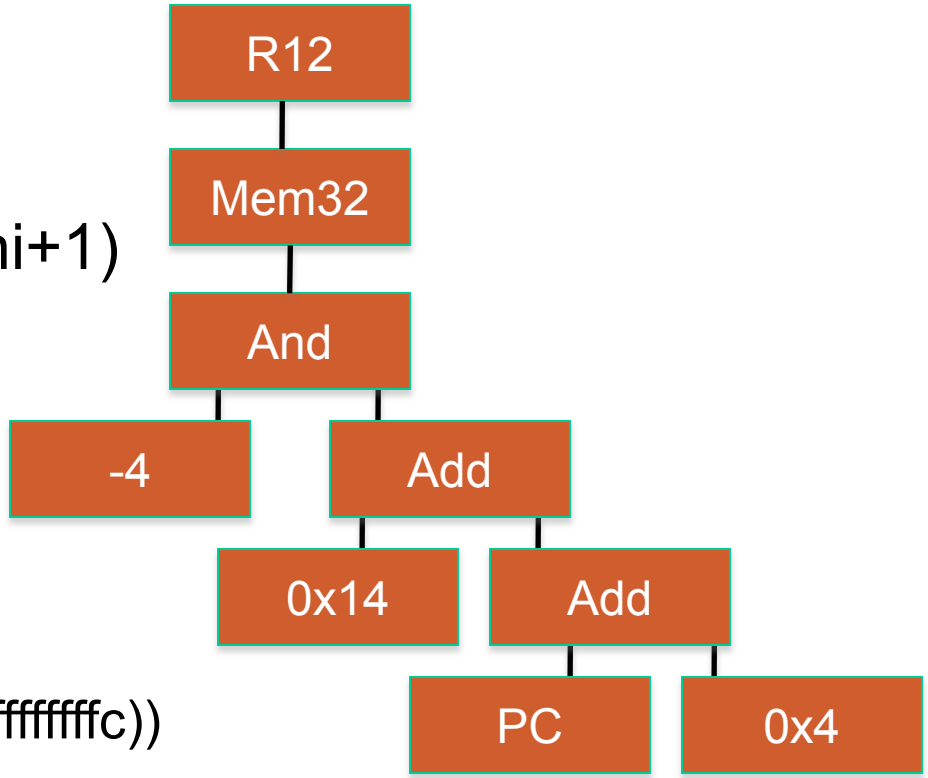
- “push eax” is a complex operation compose of two micro-operations



Syntax Graphs

- Micro-ops can be assembled into a syntax graph

LDR.W R12, =(__libc_csu_fini+1)



R12:=mem32(PC+0x4)+0x14)&0xfffffffffffffffc))

Optimization

- Greatest Repeated Common Repeated Subtree (NP Hard)

Smith-Waterman Approach

- Smith-Waterman is used in genetics to compare to protein chains
  - Certain amino acids produce the same protein
  - SKKRH == TKKRR
  - The relationships with weights is described in a table
  - Gaps and replacements are given weights

Smith-Waterman Approach

- Smith-Waterman is used in genetics to compare to protein chains
  - Certain amino acids produce the same protein
  - SKKRH == TKKRR
  - The relationships with weights is described in a table
  - Gaps and replacements are given weights
- This is exactly what we need to start compressing the firmware
  - Assign each micro-ops a value
  - C++ style name mangling can be used for better resolution
  - Compare best-match n-length segments in a shotgun method

Smith-Waterman

	C	O	D	E	L	I	N
S	0	-10	-20	-30	-40	-50	-60
N	-10	4					
I	-20						
P	-30						

Smith-Waterman

ATGTAGTGTATAGTACATGCA

ATGTAG----------TACATGCA

ATGTAGTGTATAGTACATGCA

ATGTA---G----TA----CATGCA

Smith-Waterman

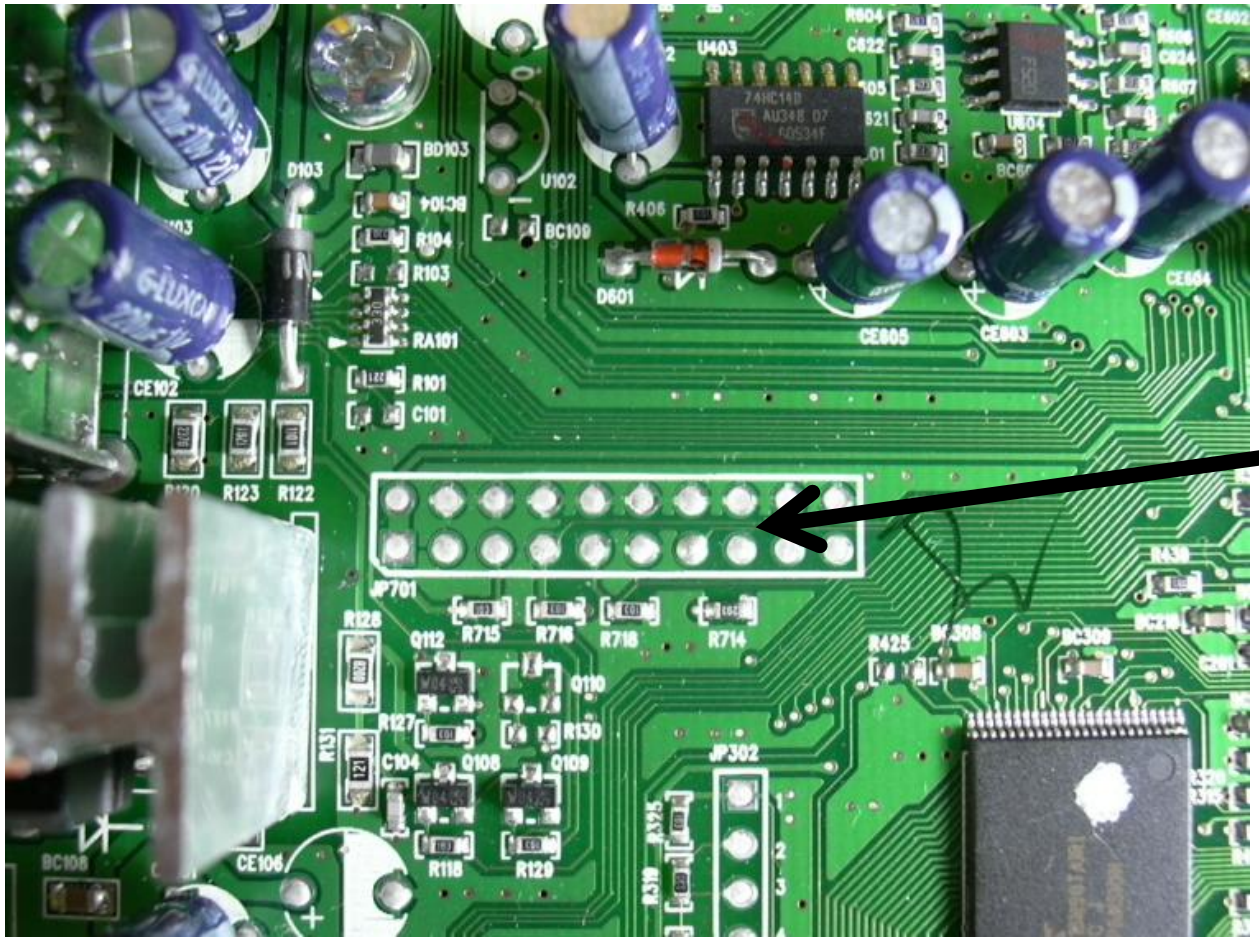
- This approach has produced good results
- It doesn’t deal well with commonly repeated patterns without post-processing
- Compression in the 20-30% is common across multiple architectures

Forensics

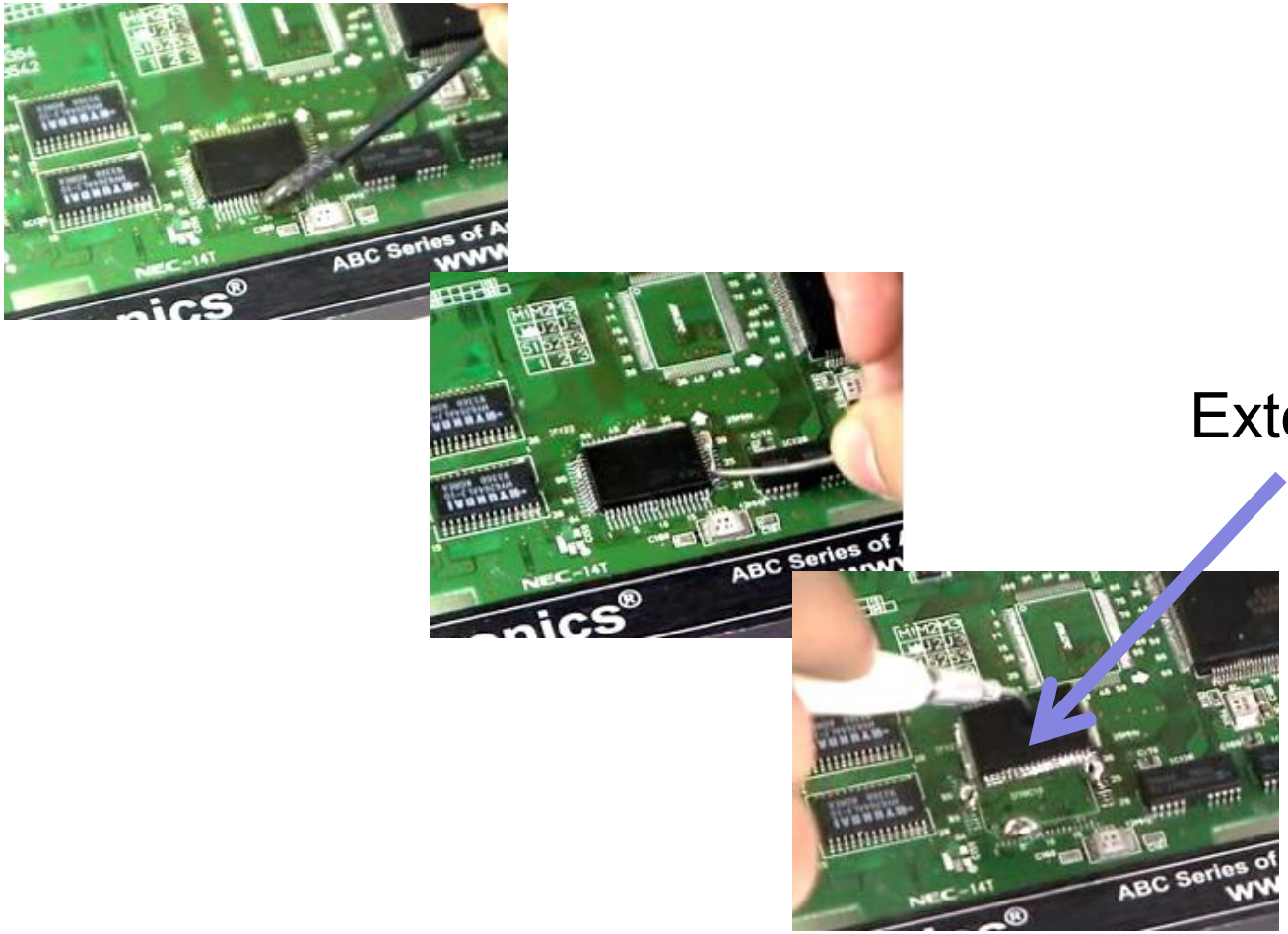
- OK, Problem laid out, now on to forensics
- Someone hands you a device and says “What did the bad people do to me?”

Analyst “Do you have the firmware from 10 years ago?”
While (patience--){
  Vendor “You need to upgrade to v5.1.4.3.2”
  Analyst “I’m doing forensics and need all versions from 2002”
  Vendor “I’ll ask someone”
  Sleep(2 days)
}

Getting the firmware



Getting the firmware



http://www.zeph.com/lowmelt.htm

Binary Normal Form

- Since you don't have a copy of the original firmware for BinDiff how can you go about quickly spotting the rootkit functions?
- Each revision of each compiler performs different optimizations
- The chances of the attacker using the exact same compiler is very small

Binary Normal Form

- XOR ECX, ECX
- ADD R0, 0
- SHL R0, #0
- MOV [ESP], 0x44
- SHR EAX, 2

- ECX:=0
- NOP
- NOP
- PUSH 0x44
- EAX:=EAX*4

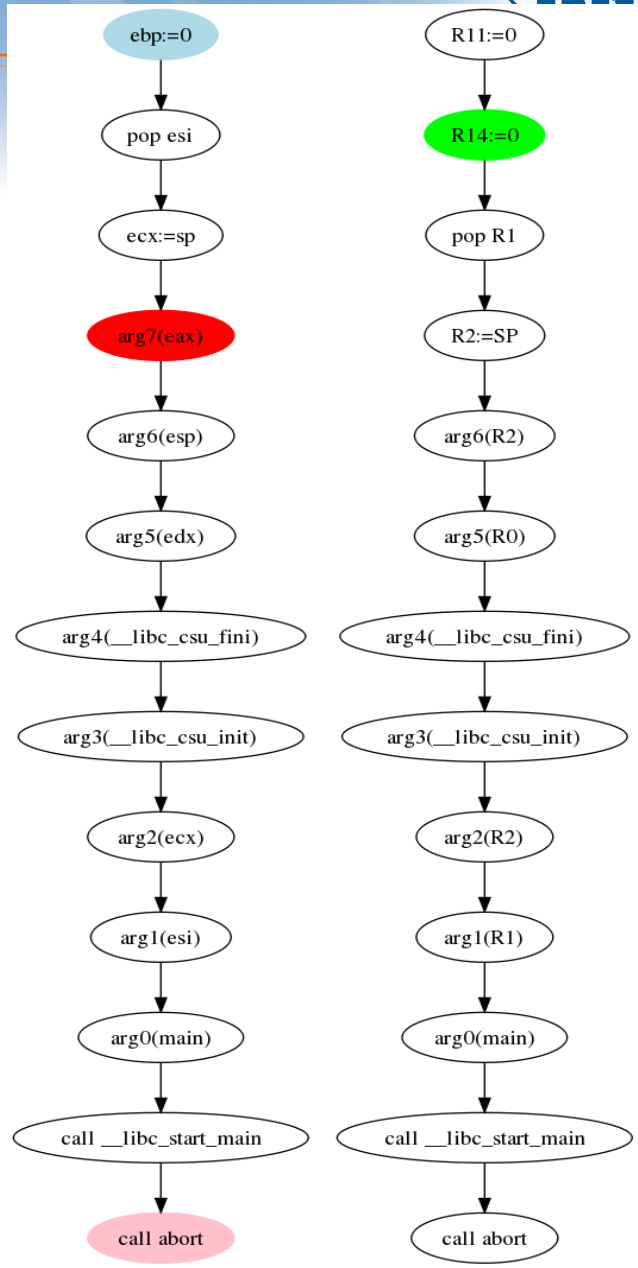
Example Functions

```
cmp     [ebp+arg_0], 3
jle     short loc_1F06

mov     dword ptr [esp+8], 6 ; size_t
lea     eax, (aThree - 1EC5h)[ebx] ; "Three"
mov     [esp+4], eax     ; void *
lea     eax, [ebp+var_40C]
mov     [esp], eax       ; void *
call    _memcpy
jmp     short loc_1F26

loc_1F06
mov
lea
mov
lea
mov
call

mov     [ebp+var_C], eax
xor     eax, eax
cmp     [ebp+arg_0], 3
jle     short loc_1F3F

mov     dword ptr [ebp+var_40C], 65726854h
mov     [ebp+var_408], 65h
jmp     short loc_1F50

loc_1
mov
mov
```

Variance

- Compilers apply the same optimization to all the code
  - Exception emit blocks
- When converting to binary normal form, variances can be noted and assigned colors or tags

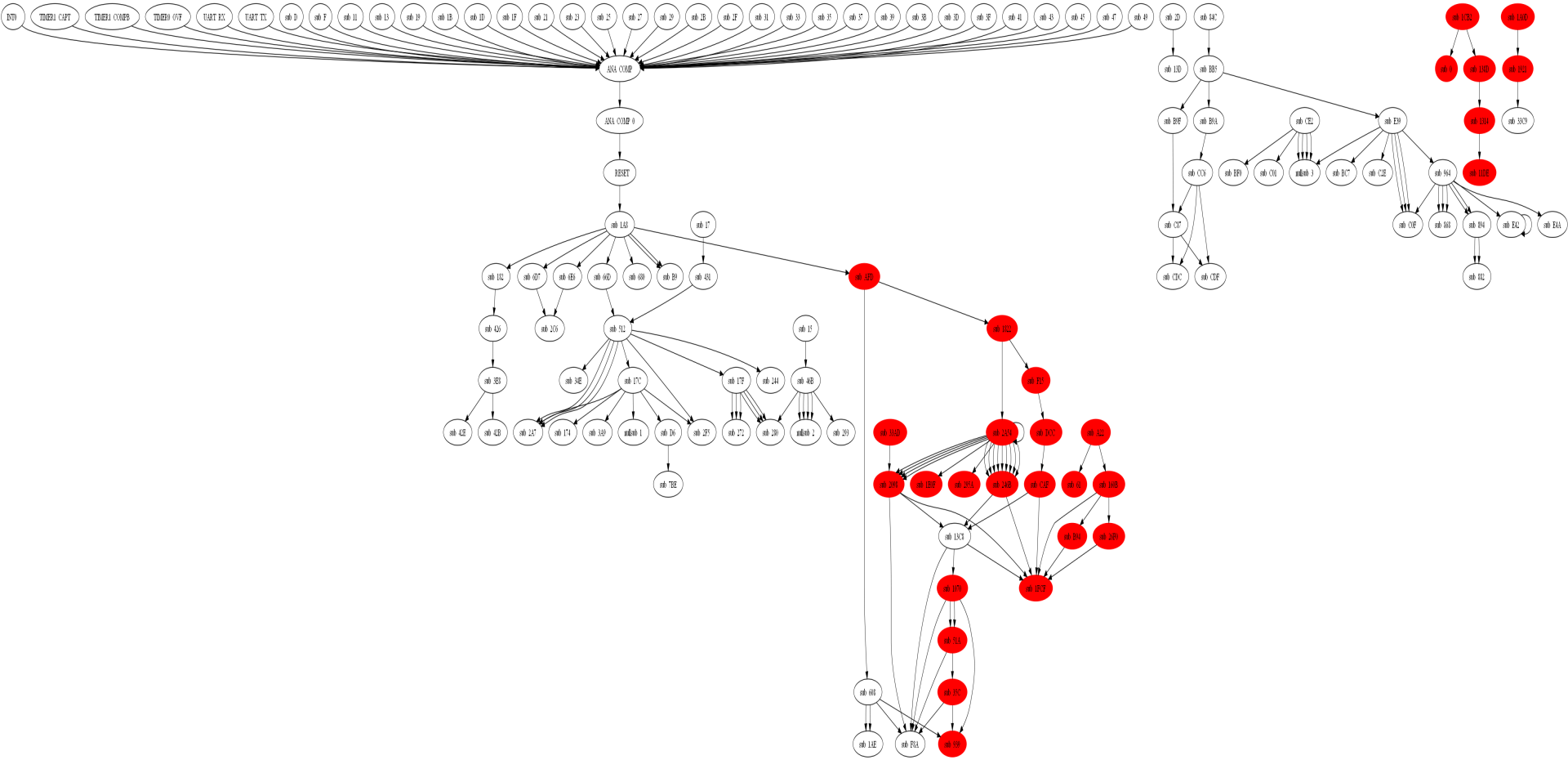
Variance


ebp:=0
pop esi
ecx:=sp
arg7(eax)
arg6(esp)
arg5(edx)
arg4(__libc_csu_fini)
arg3(__libc_csu_init)
arg2(ecx)
arg1(esi)
arg0(main)
call __libc_start_main
call abort
R11:=0
R14:=0
pop R1
R2:=SP
arg6(R2)
arg5(R0)
arg4(__libc_csu_fini)
arg3(__libc_csu_init)
arg2(R2)
arg1(R1)
arg0(main)
call __libc_start_main
call abort

Variance

- Both examples produce the same graph
  - They should since they came from the same source code
  - We can note each variance and assign it a value or color
  - Sorting functions by variance reveals the rootkit

Variance


INT0
TIMER1 CAPT
TIMER1 COMPB
TIMER0 OVF
UART RX
UART TX
ANA COMP
ANA COMP 0
RESET

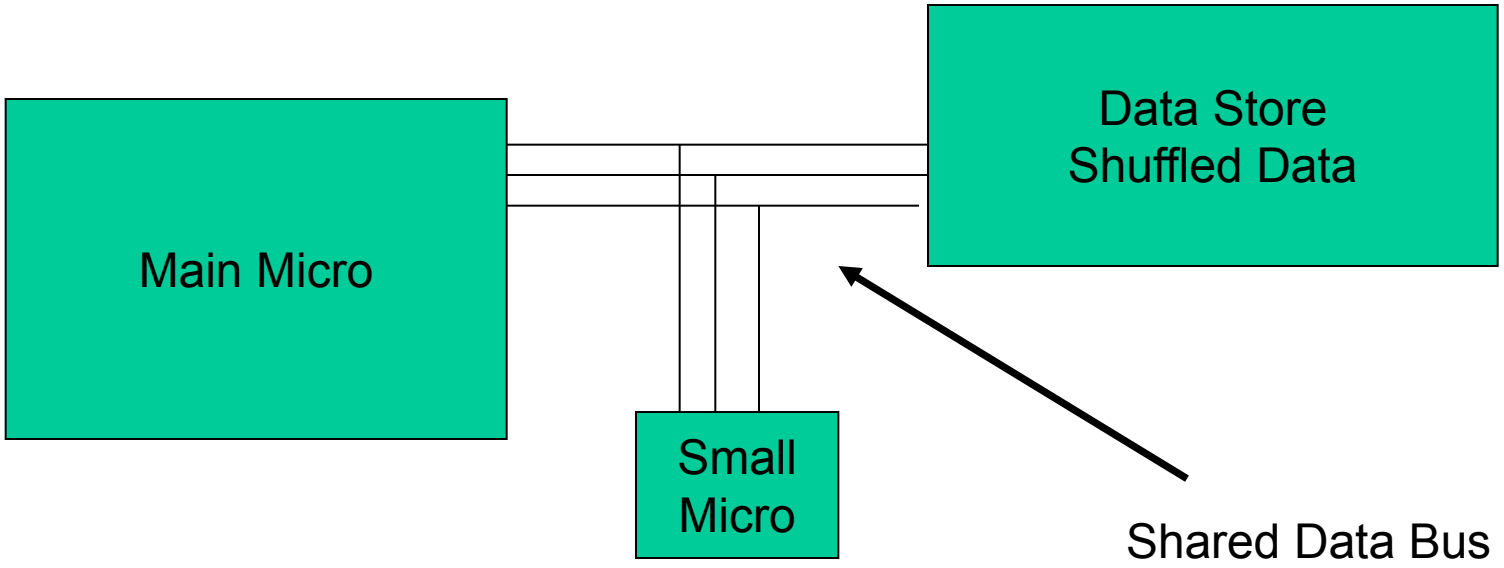
Hiding Off-Chip

- What if the defender is likely to look at the firmware?
  - For example in nuclear where they have good change control
- The first answer is to make the chip lie
  - (The attacker does control the chip communications after all)
- If that's not possible the attacker can choose to live on a smaller chip and then crawl up into the main chip
- What he needs
  - Shared busses or a vulnerability
  - Empty storage somewhere
  - Access to timing information

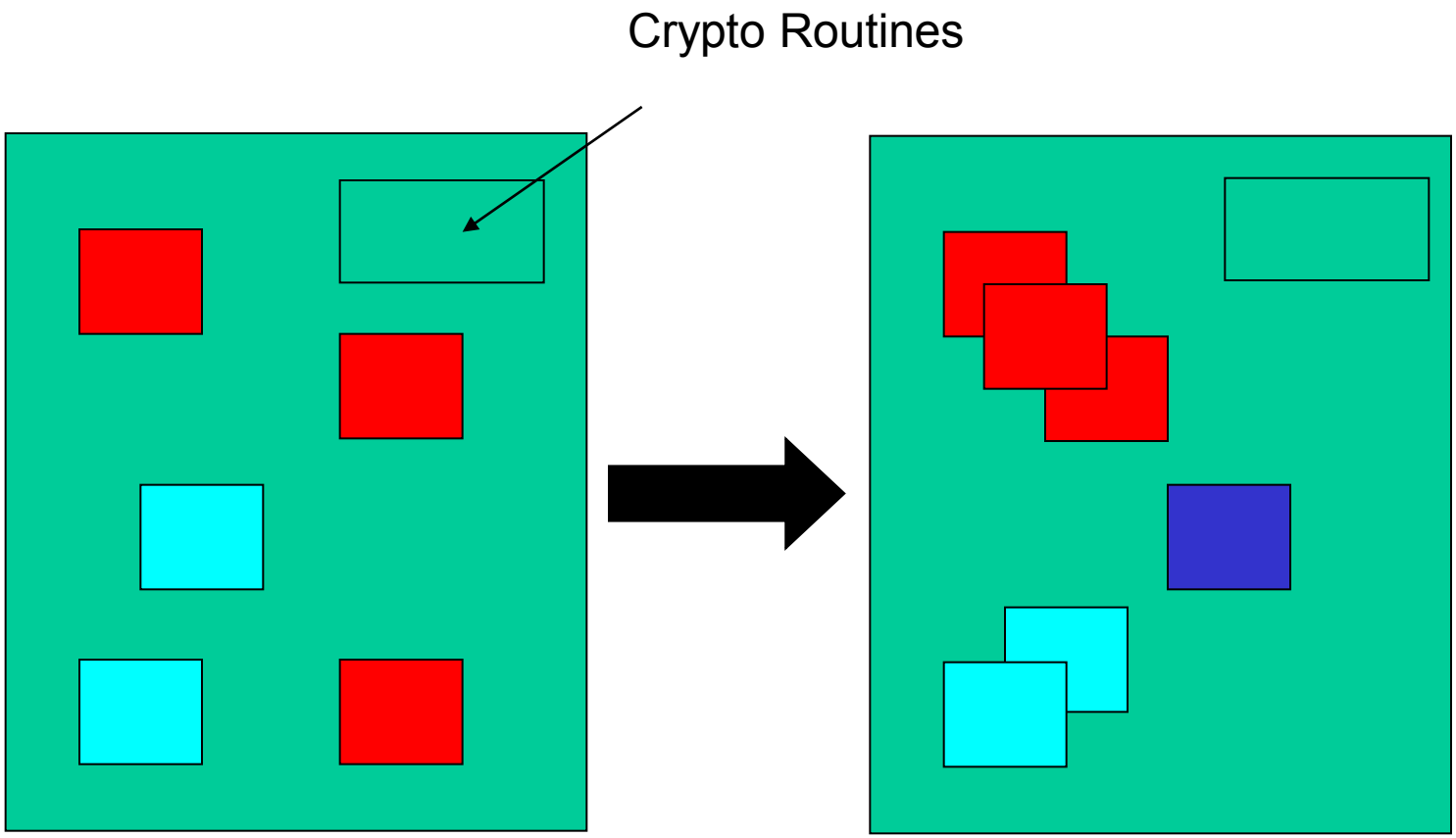
ROPup

- ROP- Return Oriented Programming
  - (The young hackers like to rename all the old techniques)
- ROP gadgets don't have to be small
- Pieces of compression and crypto routines can provide heavyweight building blocks for memory overlays
- A microcontroller can be bootstrapped from one state to another using off-chip storage such as serial memory
  - Preferably encrypted serial memory
  - Slack space in flash works well
  - Bootloaders also can be used

Classic ROPup



Unshuffler in ROP



ROPup

- Shuffled data from the store is combined with existing firmware in memory to produce a more useful image
- A clever attacker can make it so the complete in-memory image is only valid when conditions for the attack are met
- The checksum of the main flash remains unaltered
  - The main CPU is simply a stepping stone to the lower chip during exploitation
  - The main CPU is simply a host for malicious logic during attack
- If necessary no bytes stored anywhere look like executable code
  - No attacker controlled data is in an executable section at rest

Attackers are winning

- At this point, attackers are way ahead of the defenders
- Defenders lack the basic infrastructure to handle attacks against field equipment
  - You have no real way of verifying the various firmwares on your motherboard much less an embedded device

Attackers Are Winning

- We need to build a model of attackers in embedded systems
- We also need to build a toolbox that can find and remove them
- I believe comparison versus a binary normal form is a good starting point

Questions?