Miniaturization

“Why, sometimes I've believed as many
as six impossible things before breakfast.”
— Lewis Carroll, Alice in Wonderland

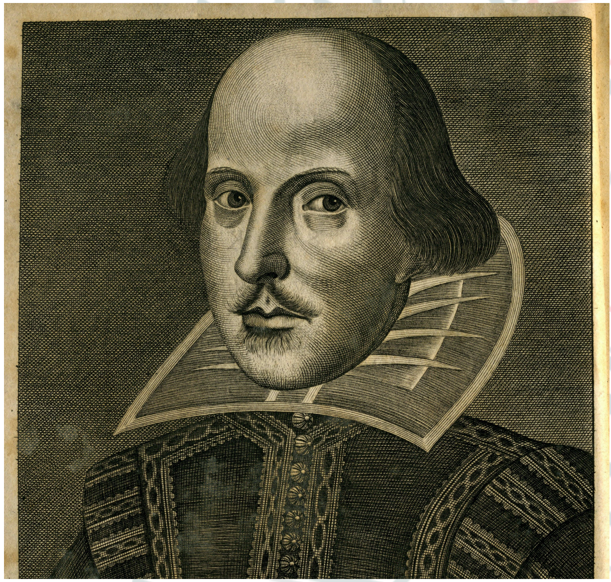
Jason Larsen
Blackhat 2014

Who Am I?

- Jason Larsen
- CyberSecurity Researcher specializing in critical infrastructure

A ~~play~~ presentation in two parts

- I submitted two talks to Black Hat and ….. they said to do both of them at the same time
- Creating a kick@#$$ SCADA attack firmware modification in two acts
  - Act I : Making the attack code really small
  - Act II : Efficiently inserting the rootkit into the firmware
- Popcorn Warning
  Lots of algorithms and assembly code ahead



Could You Hide an Entire Attack in a Pressure Meter?

- Small microcontroller
  - Kilobytes of memory (total)
  - Very little CPU power
  - Kilobytes of flash (total)

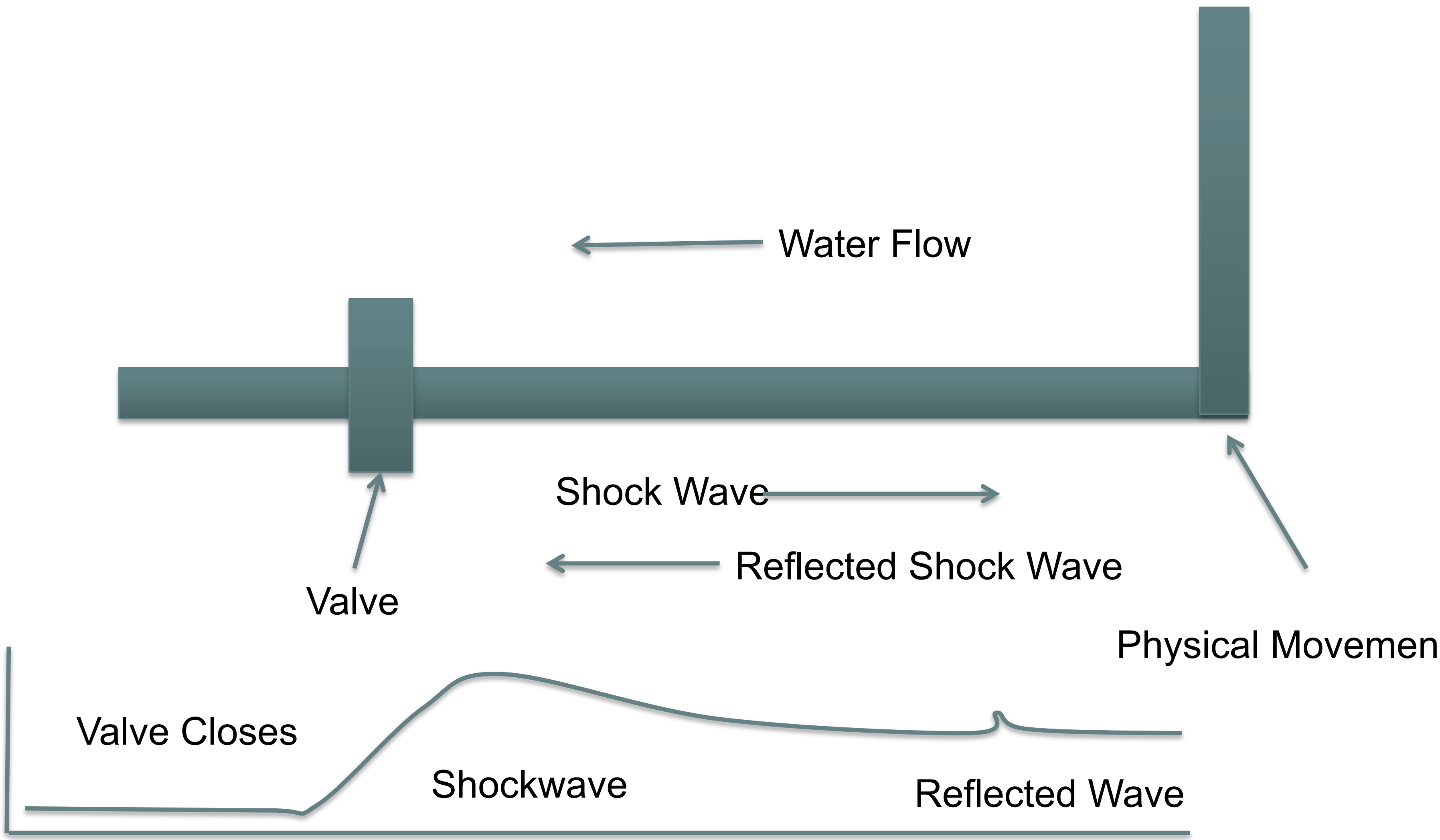
IOActive

Eleven Years Ago
(And yes, it was lame even then)

Record and Playback

- The operator's screens didn't update in this video
- It was created using the trusty record-and-playback method
- What if we want to go small?
- What if we want to go really small?
- What if we want to go down into the sensors?

The Scenario



The Scenario

- The shockwave travels at the speed of sound in water

$$t = L / A$$

- Or, if pipe is elastic

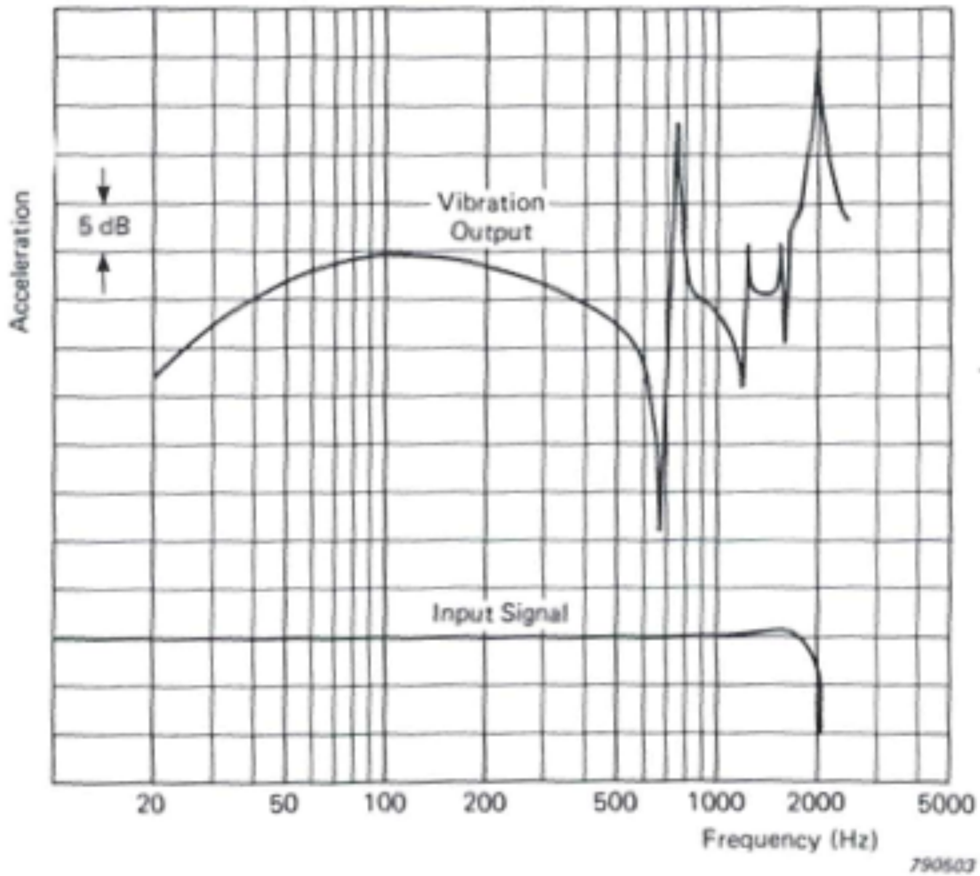
$$E = \frac{Ewater * Tpipe * Epipe}{Tpipe * Epipe + Dpipe * Ewater}$$

- The optimal interval to cycle the valve
  - X is the time between valve closing
  - Y is the time between the pressure wave and the rarefaction wave

$$\frac{2X + Y}{4}$$

*Fluid Dynamics. Professional Publications Inc.

Supersampling



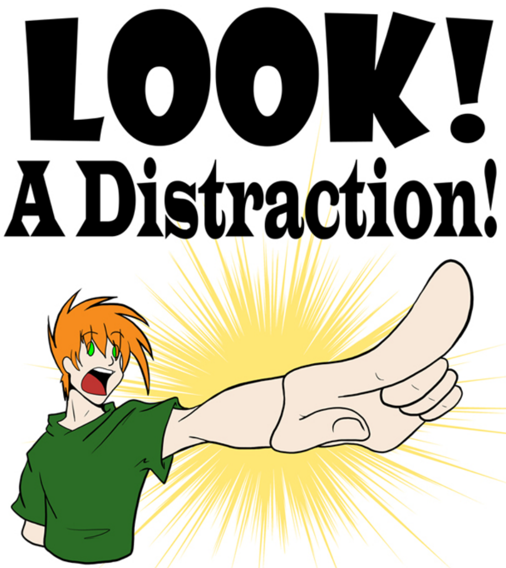
*Mechanical Vibration and Shock Measurements

Act I – Making the Attack Code Really Small

Popcorn Alert! Lots of assembly ahead

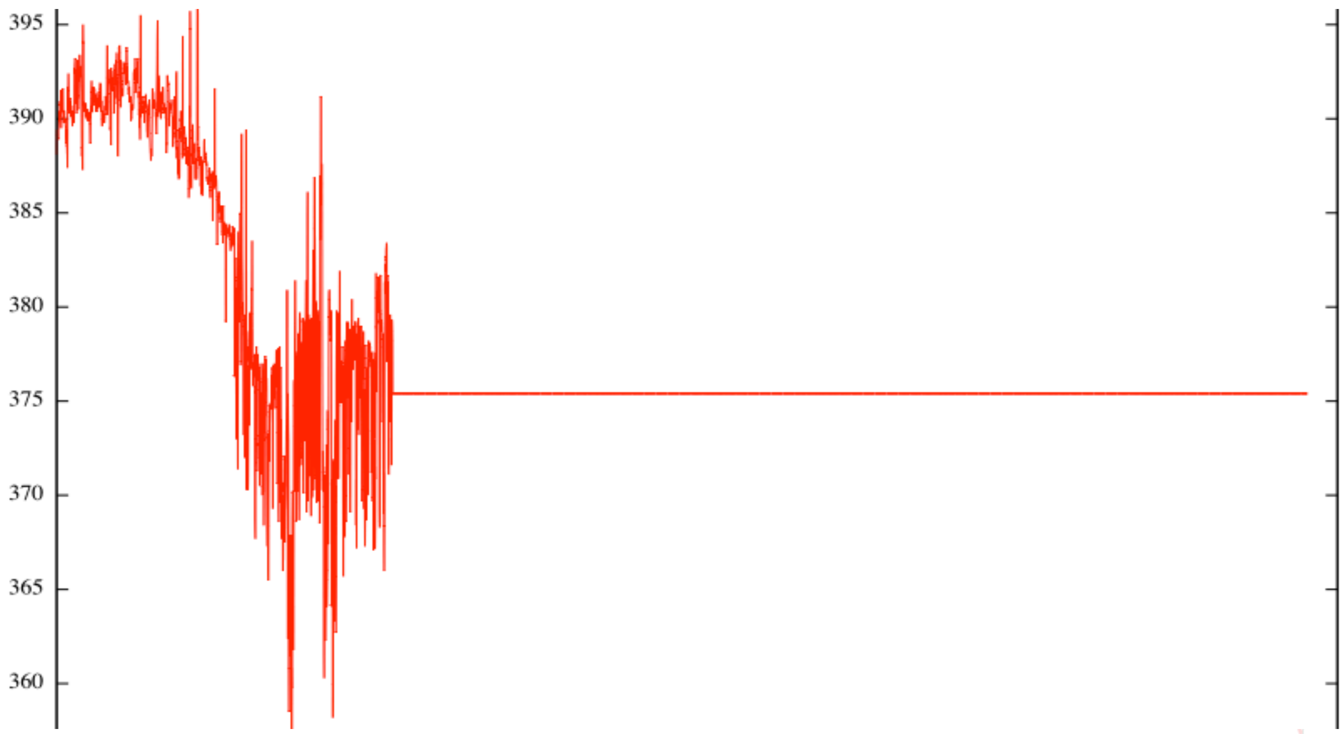
Miniaturizing Firmware Attack Code

- Miniaturizing the Attack Code
  - Spoofing with Runs Analysis
  - Triangles for Filtering Noise
  - Scale-free matching for Watching the Process
- Inserting the Attack Code into the Firmware
  - MicroOps
  - Binary Normal Form
  - Abusing Needleman Wuncsh to Merge Firmware
  - Metasploit for Firmware
- Demos



IOActive

Sensor Noise
(This isn't going to fool anyone)



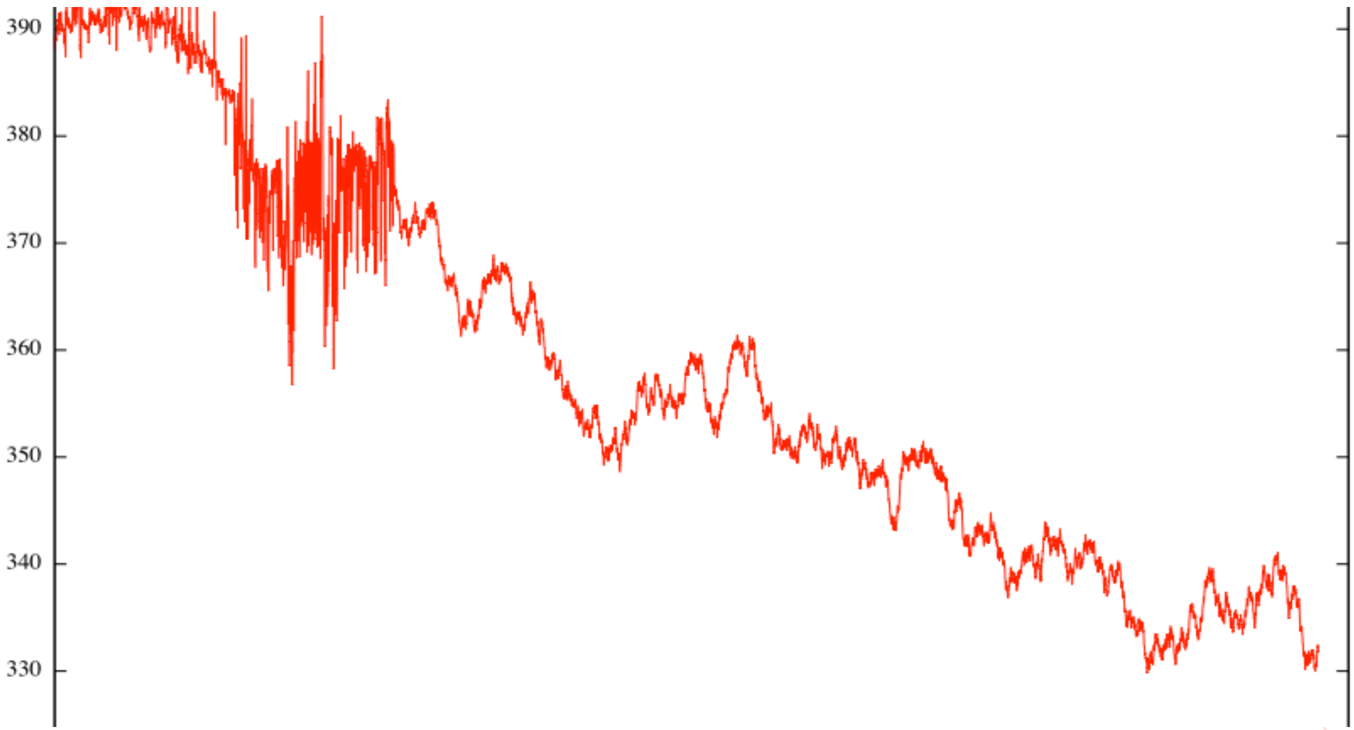
Anyone looking at this will think "dead sensor"
The forensics team will zoom on this immediately

Sensor Noise

- Humans are really good at spotting differences in “randomness”
- Even on graphical displays, operators get used to the “jiggle” in the visualization

Sensor Noise
A Random Walk

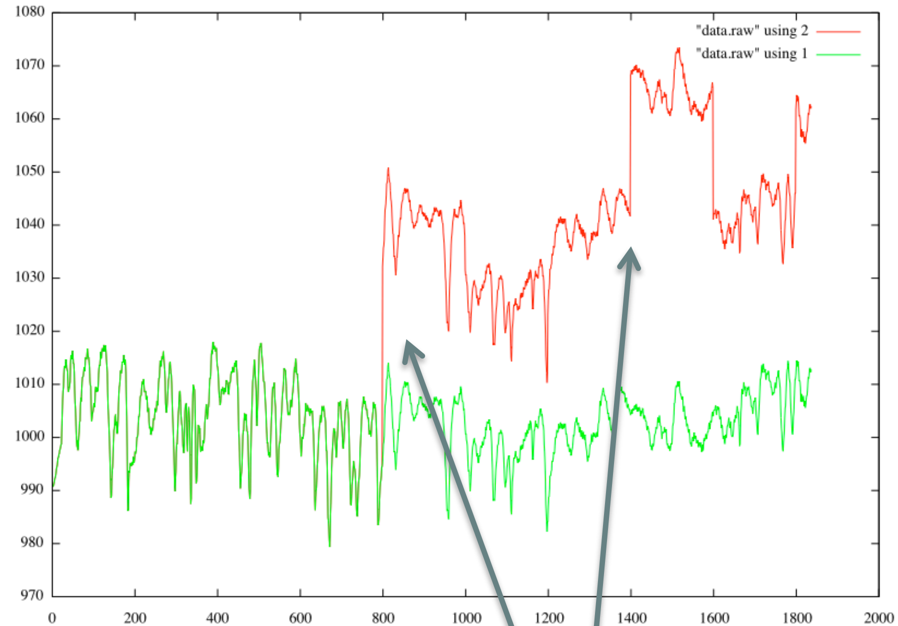
- Just adding randomness
  - It’s easy for a human to spot where the spoof starts
  - This doesn’t preserve the “spikiness”, “width”, and “gaps” of the original



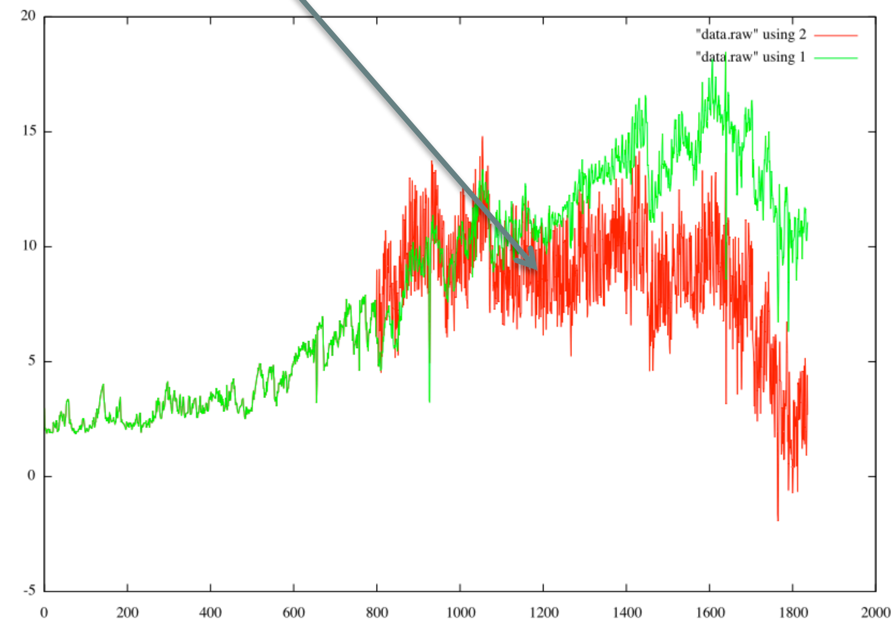
Sensor Noise

- If you’re a math major, you’re probably shouting “Yeah! FFT!”

Scaling and Shifting



Scaling can increase magnitude of the noise



Shifting requires an averaging function to eliminate stair steps during adjustments

These are solvable problems but they grow bigger as you try to get it right

Runs Analysis

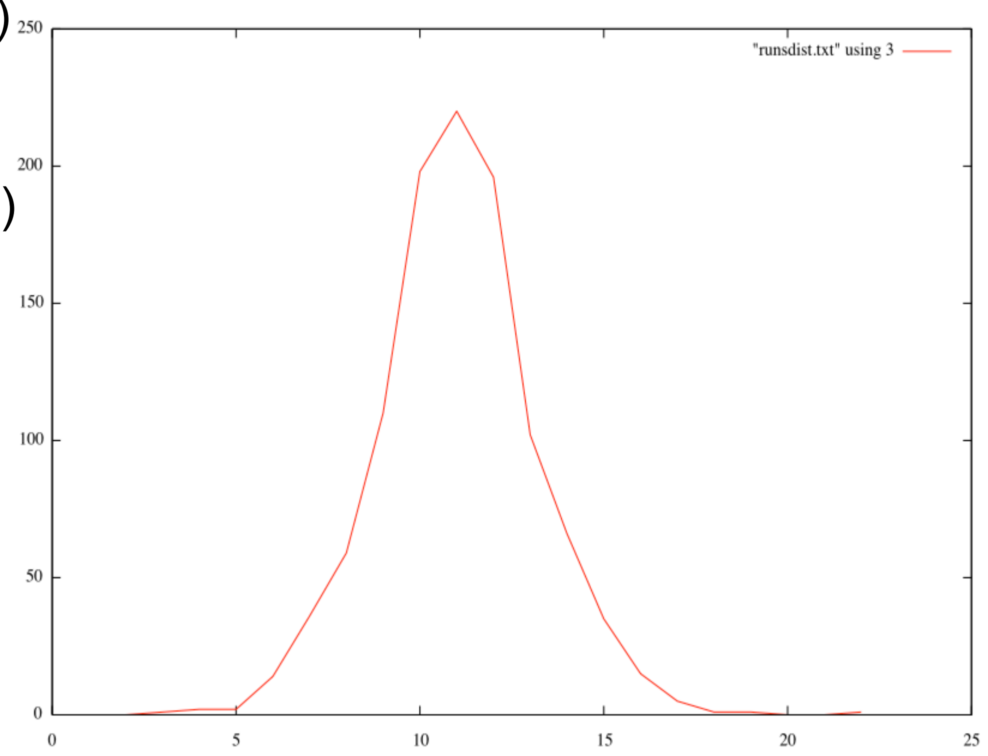
- Most of these techniques require that the attacker have access to previously recorded data to get the algorithm right.
  - What if we don’t get to see the sensor noise before we start?
- Runs analysis can spoof the sensor noise with no preknowledge of the data.
- Sensor noise can be treated as a random walk
- Random walks can be characterized through an analysis of the length and frequency of runs

Runs Analysis

- During a learning phase, count the runs

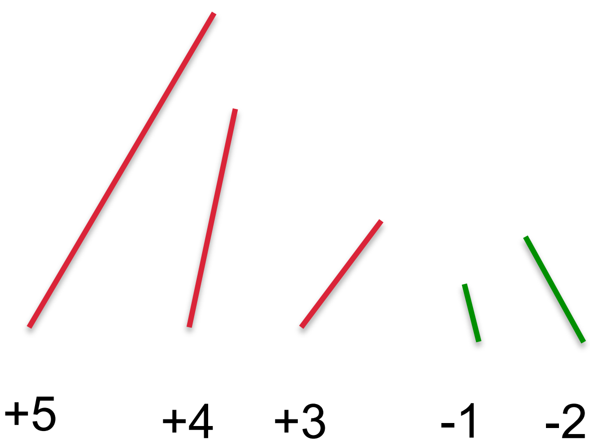
390.3
390.4 +3 increasing (moved 0.3)
390.6 -1 decreasing (moved -0.3)
390.3
390.5 +3 increasing (moved 0.6)
390.9
391.1 -1 decreasing (moved -0.8)
391.2
390.9
390.9
390.8

As expected this gives a nice normal distribution

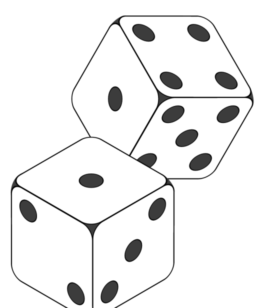


Runs Analysis

- Taking the average movement of a runs bucket turns into a slope and a length

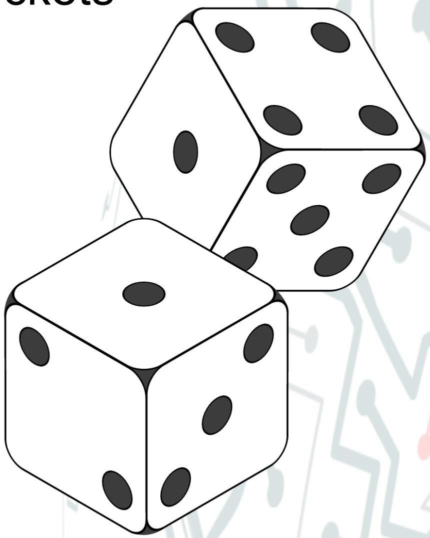


Chaining line segments together reproduces the noise



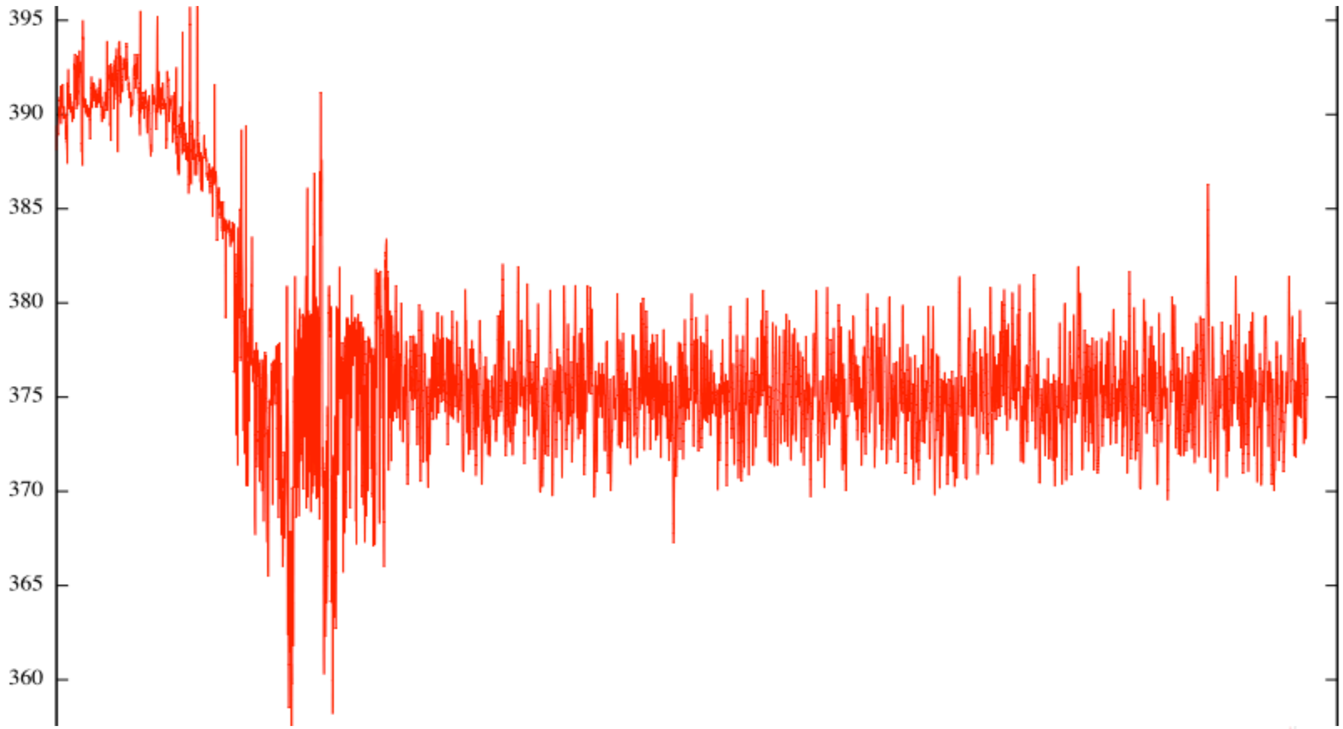
Runs Analysis

- The playback algorithm is really simple
  - Add up all the positive/negative buckets
  - Choose a random number 0<x<sum(buckets)
  - Move by average bucket value for bucket samples
  - If desired is above current, choose from positive buckets otherwise choose from negative buckets



IOActive™

Runs Analysis

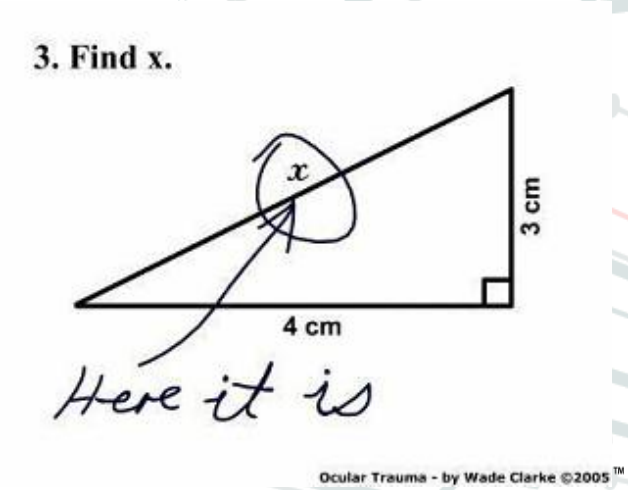


We get nice, believable sensor noise with no prior knowledge of the system

Runs Analysis

- We have to fit this on the microcontroller. How big is the code+data?
  - Just over 400 bytes depending on linker constraints
  - ARM, X86, and PPC are similar in size
- We can definitely fit that inside a pressure sensor

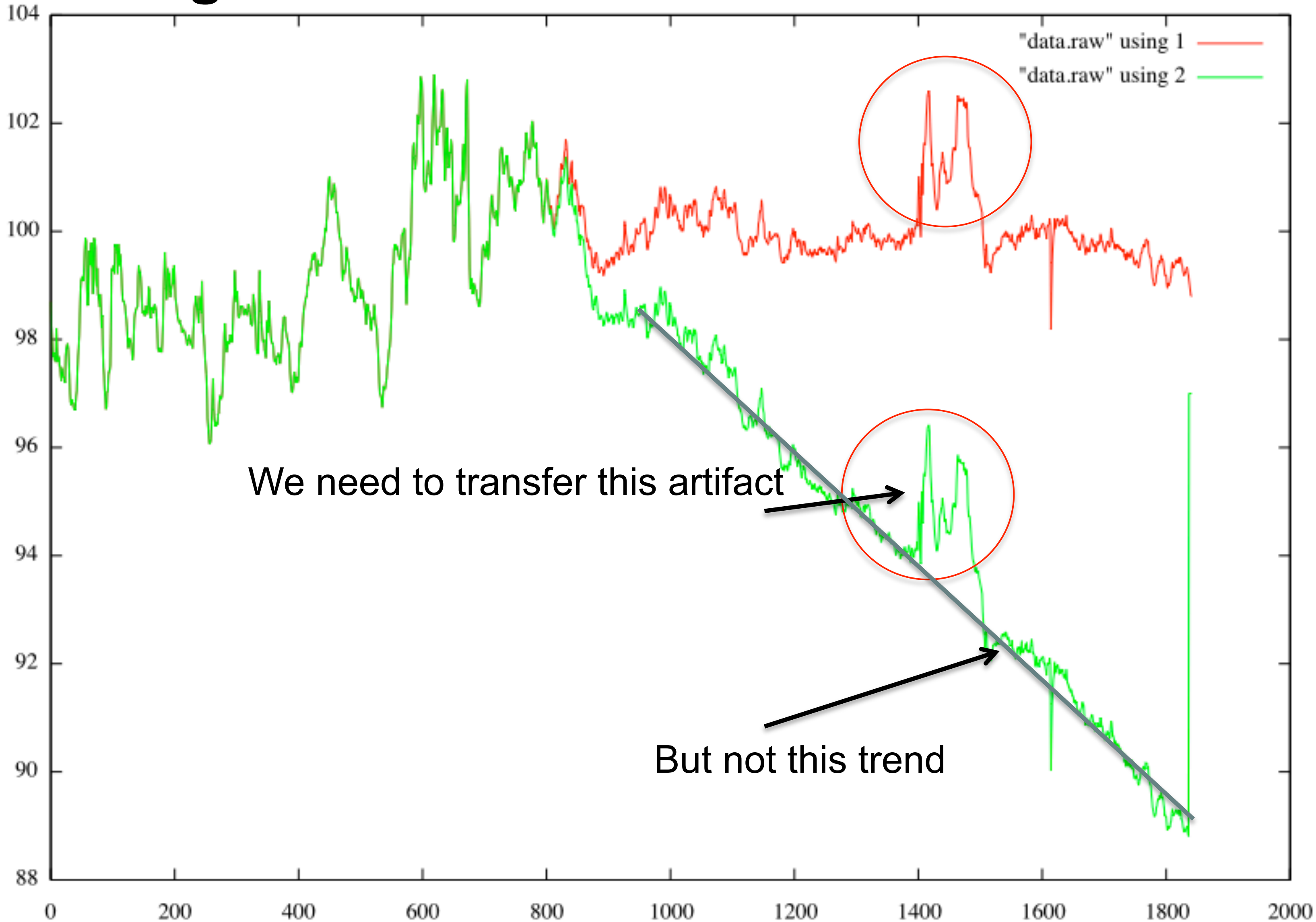
- Miniaturizing the Attack Code
  - Spoofing with Runs Analysis
  - Triangles for filtering noise
  - Scale-free Matching for Watching the Process
- Inserting the Attack Code into the Firmware
  - MicroOps
  - Binary Normal Form
  - Abusing Needleman Wuncsh to Merge Firmware
  - Metasploit for Firmware
- Demos



Leveling

- We’re going to be attacking the process and making changes
- We need to preserver the small changes that are expected so the forensics guys can match them up later
- We need to remove the big changes so the logs don’t show what we’ve been doing

Leveling



"data.raw" using 1
"data.raw" using 2
We need to transfer this artifact
But not this trend
104
102
100
98
96
94
92
90
88
0
200
400
600
800
1000
1200
1400
1600
1800
2000

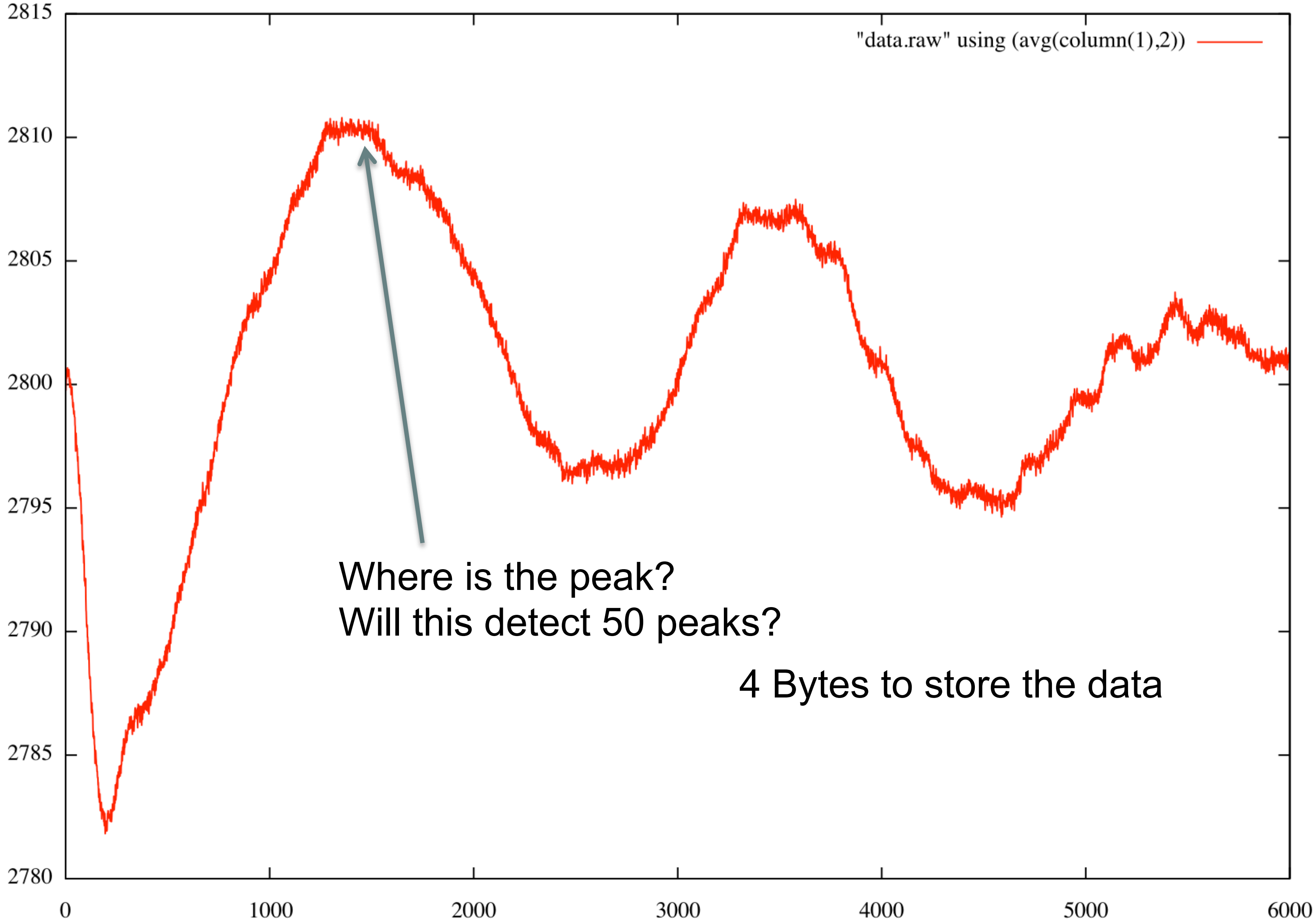
Leveling

- How big is an artifact?
- How big is a disturbance?
- Do I need a different algorithm for every type of signal?
- What if I don’t get to see the signal beforehand to choose my algorithm?

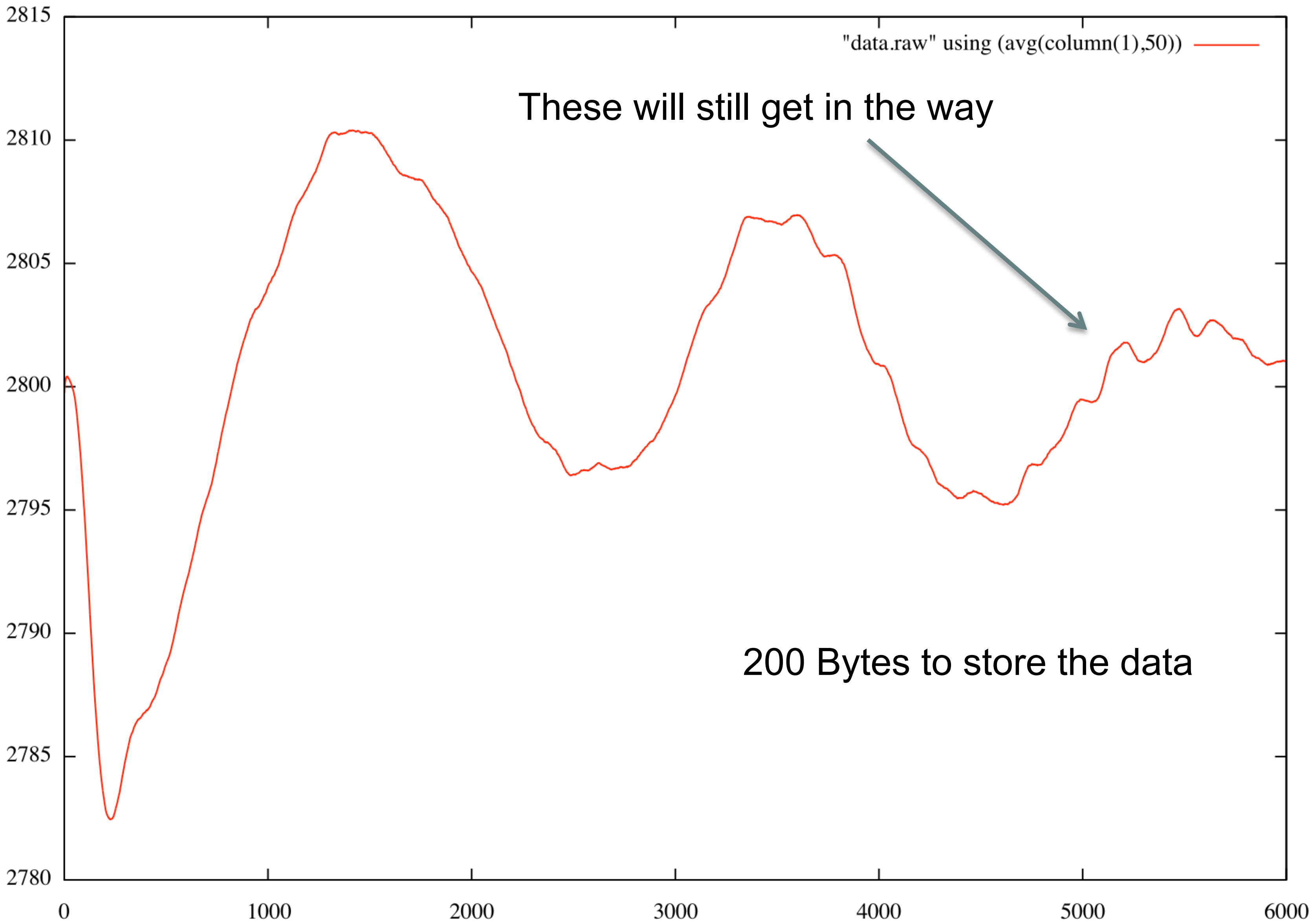
Leveling
Moving Averages

- Everyone starts with a moving average to filter out the data from the noise
  - This might not be the best approach
- Even simple algorithms can be large when the size of the data is taken into account

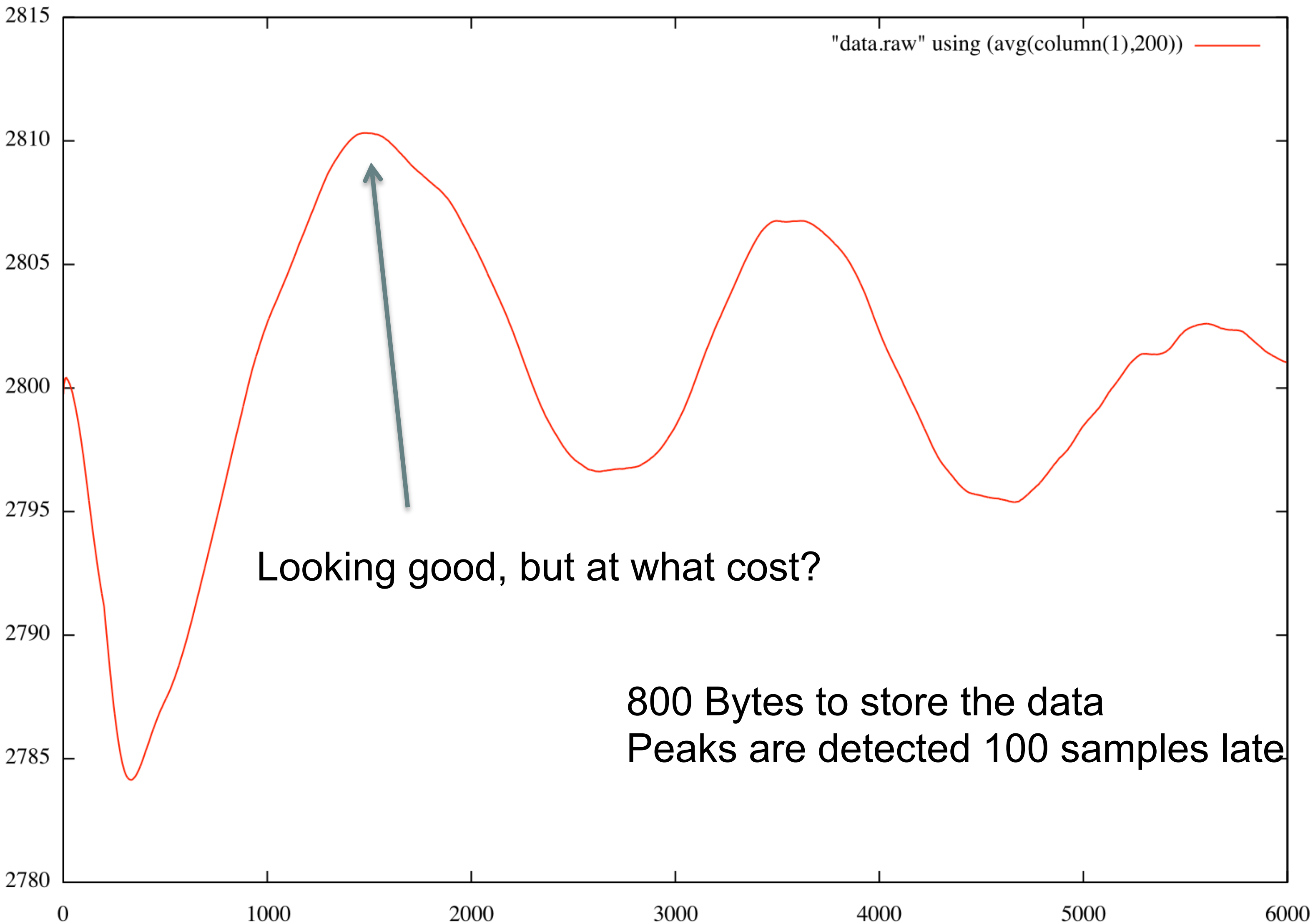
Moving Average 2 point



Moving Average 50 point



Moving Average 200 point



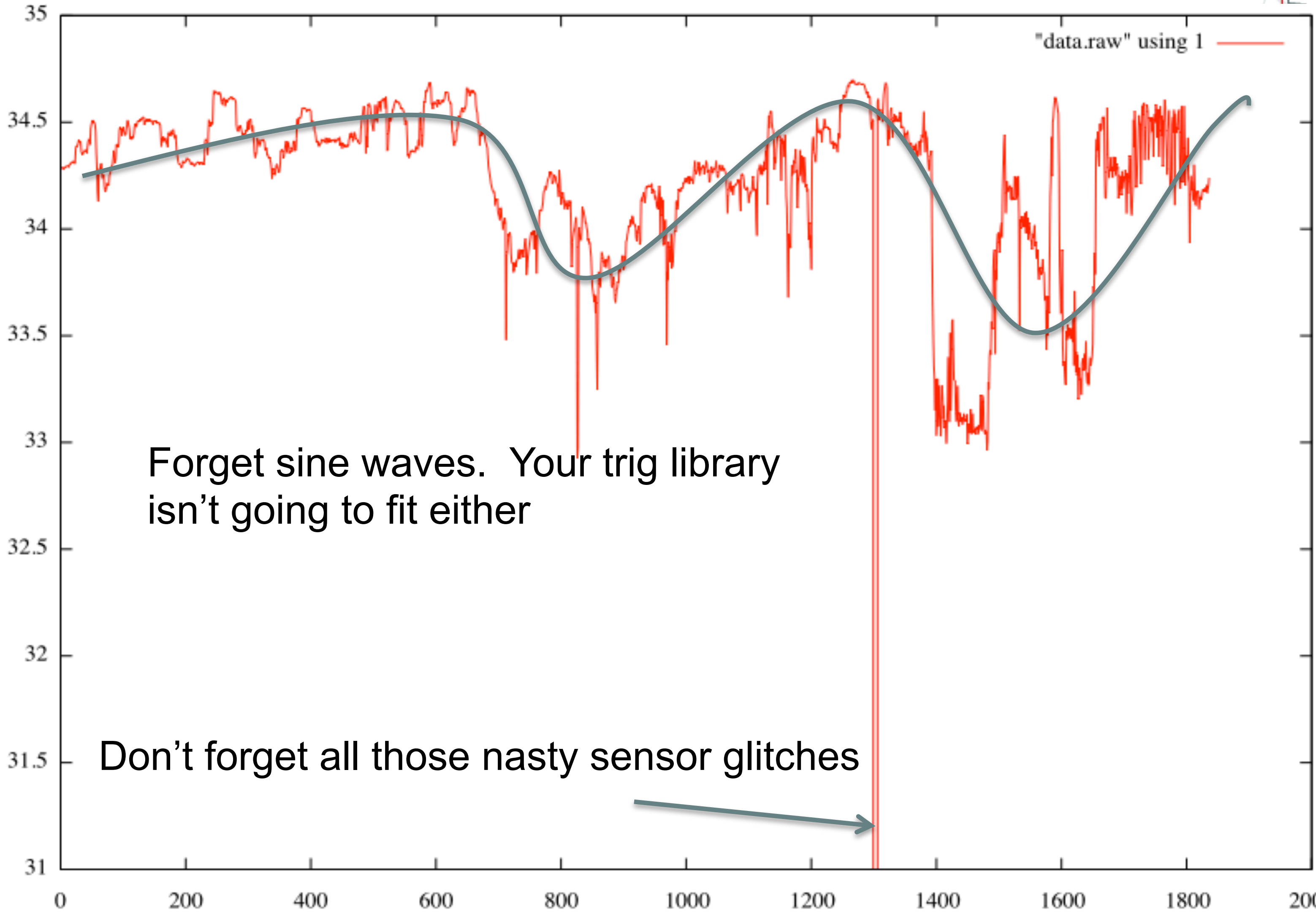
Beyond Moving Averages
Fitting Curves to the Data

- A moving average is an example of a scale-dependent algorithm
- How many points should be applied to smooth out the curve?
  - It’s impossible to know without an example of the data
- LOTS of code is needed to deal with scaling factors
  - Mm/Hg, cm/h20, Pascals?
  - More than all the rest of the attack code combined

IOActive™

Beyond Moving Averages

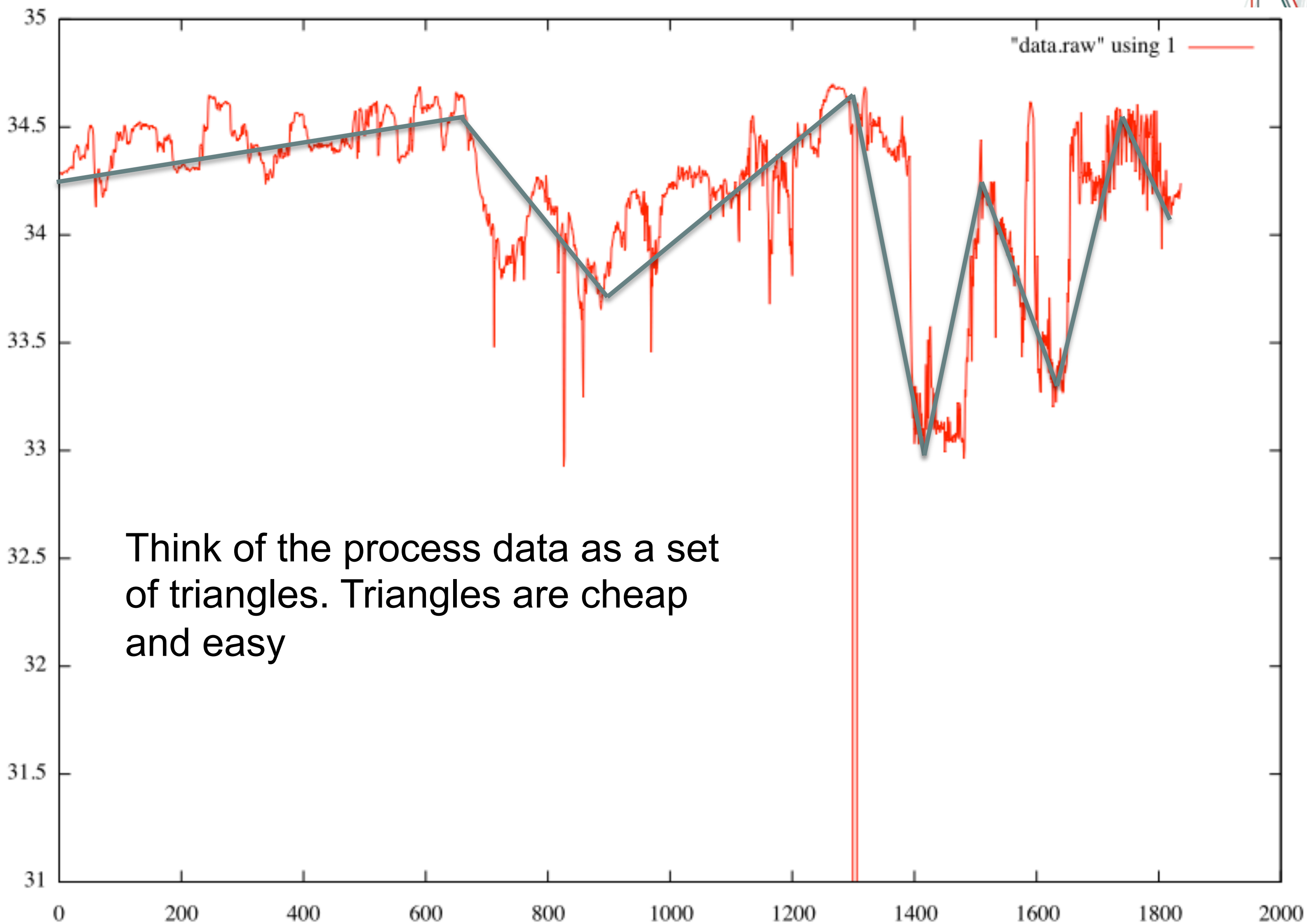
Scaling and Leveling Algorithms



Triangles

- Triangles are a good-enough approximation of the process data
- We just need a very small algorithm to fit triangles onto the process data
- How big is the optimal triangle?
  - The largest features are the ones you care about
  - We need an algorithm that will produce triangles that is scale independent
  - The triangles should all cover a similar area

Triangles



Think of the process data as a set of triangles. Triangles are cheap and easy

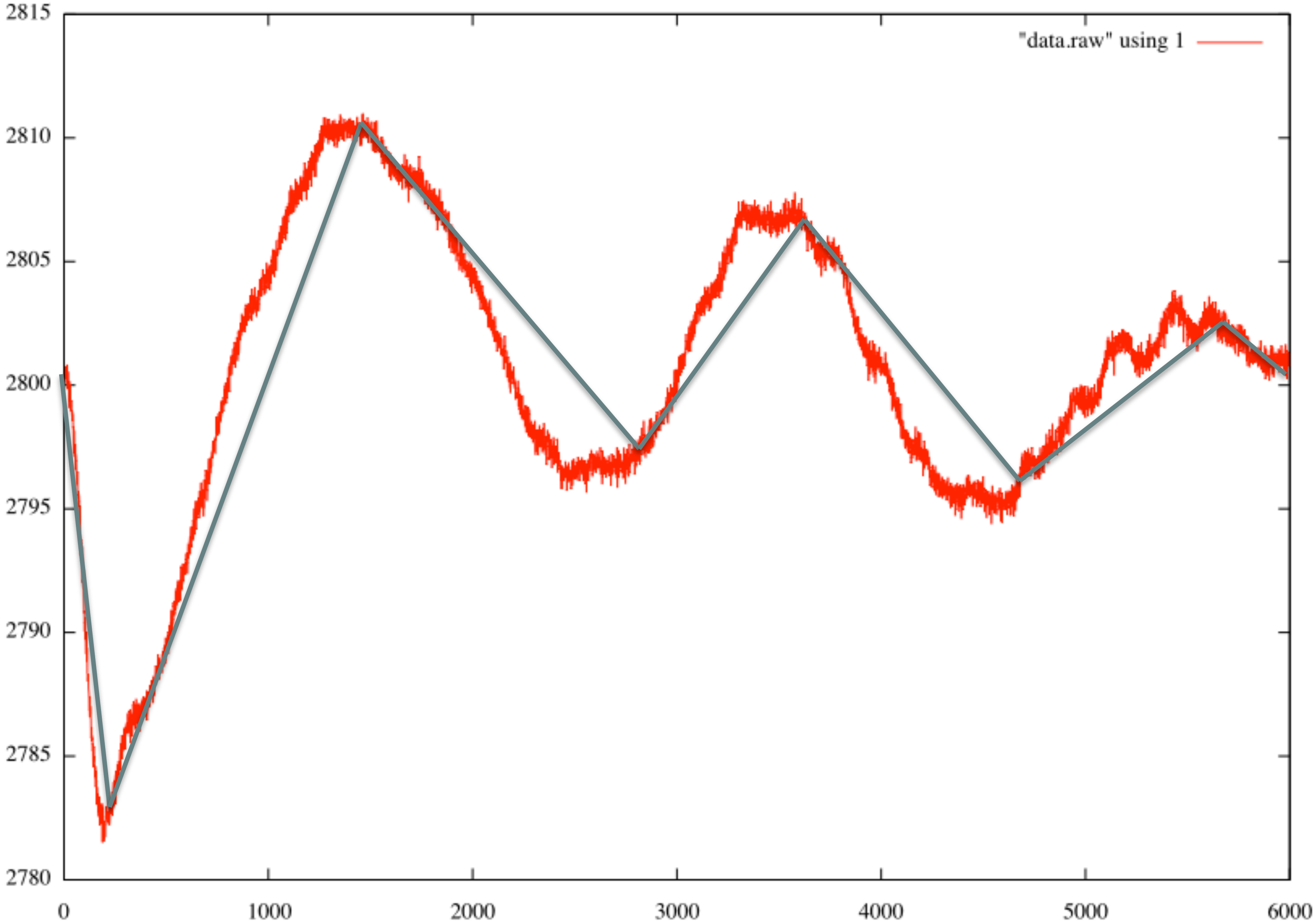
Triangles

- We can make some assumptions about the data
  - The process is not running out of control therefore, it will oscillate as the feedback mechanisms control the process
  - Artifacts smaller than the noise are too small to affect the process
  - There isn't significant hysteresis in the system

Triangles
(Still tweaking this one)

1. A simple algorithm
2. Declare a vertex at the first value
3. Choose an arbitrary starting window n. Calculate or estimate a smoothing factor s=log(n).
4. Note the minimum and the maximum values in the window.
5. Draw a triangle from the origin through the minimum and maximum values and ending in a vertical line at n.
6. Declare a vertex at the midpoint of the vertical line at n.
7. Start drawing a second triangle from the vertex using the slopes of the previous triangle.
8. Count y,z samples that are above/below the triangle.
9. When y or z > s, declare a vertex at the midpoint of the vertical line through the current sample
10. If y<z, increase the slope of the top and decrease the slope of the bottom line otherwise do the opposite
11. If the number of samples between the current sample and the last vertex < 4n. then increase n
12. If at any time there has been no vertex in 4n samples, declare a vertex at the midpoint of the line through the current sample and decrease n.
13. Go to step 6

Triangles

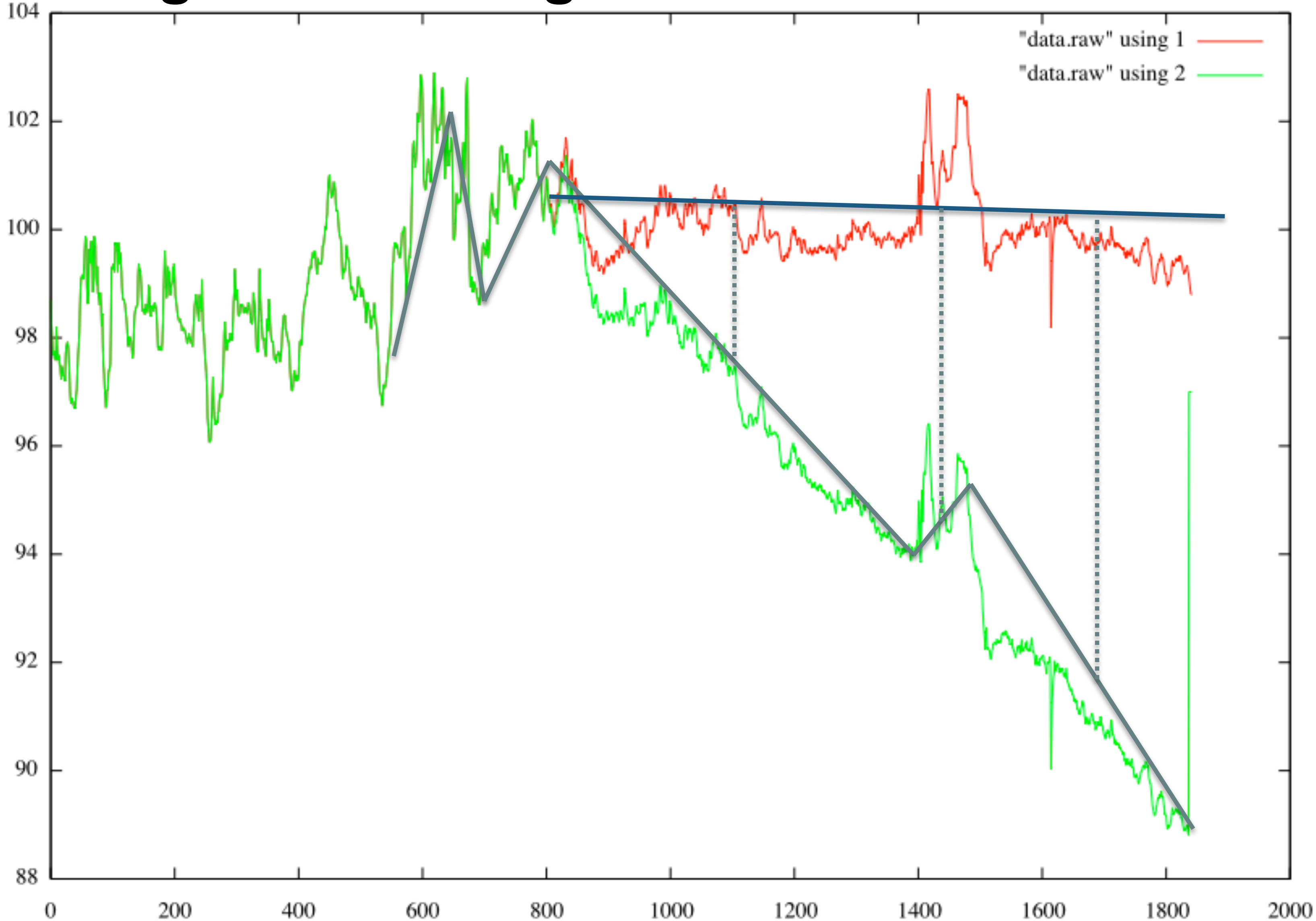


"data.raw" using 1
2815
2810
2805
2800
2795
2790
2785
2780
0
1000
2000
3000
4000
5000
6000

Transferring Artifacts

- Now that the triangles are complete
  - Declare that the midpoint of each line segment should be scaled to the spoof value
  - The difference from the line segment to the observed data is averaged into the spoof data

Scaling and Leveling

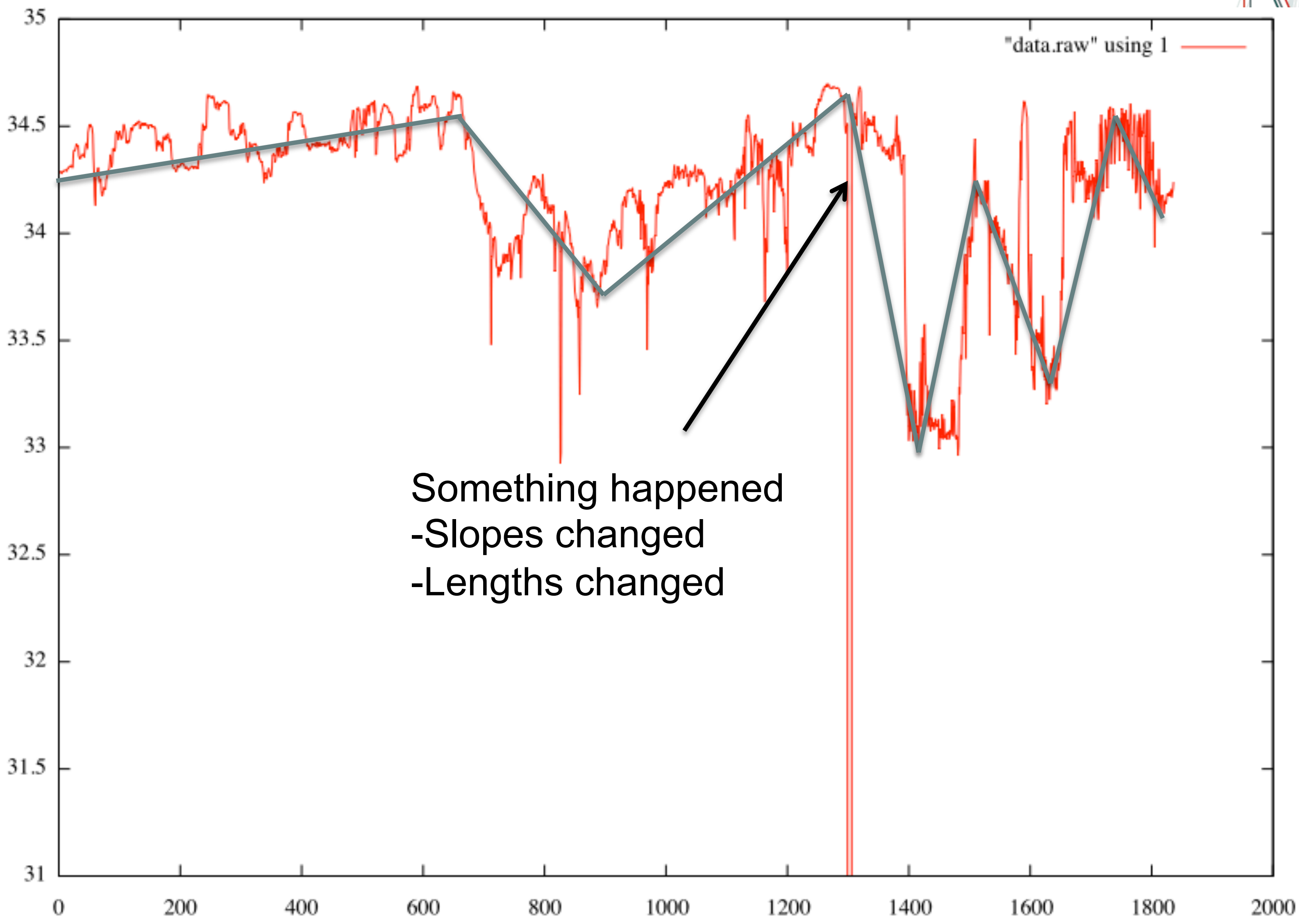


- Miniaturizing the Attack Code
  - Spoofing with Runs Analysis
  - Triangles for Filtering Noise
  - Scale-free Matching for Watching the Process
- Inserting the Attack Code into the Firmware
  - MicroOps
  - Binary Normal Form
  - Abusing Needleman Wuncsh to Merge Firmware
  - Metasploit for Firmware
- Demos

Artifact Extraction

- We need to spot the pressure wave and the reflected wave
- We can extract the state of the process using the triangles
- This saves CPU time because we’re only running this logic when we declare a new vertex

Artifact Extraction



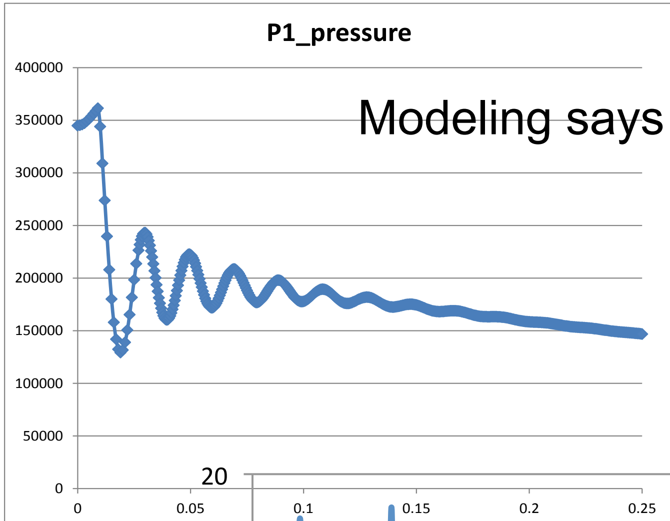
Artifact Extraction

- For our attack model we only need two artifacts
  - When did the pressure wave hit?
  - When did the reflected wave hit?

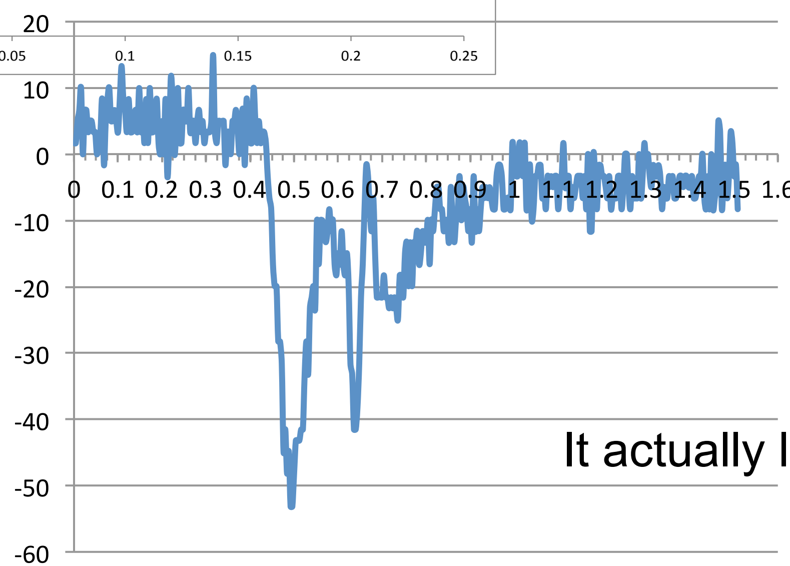
$$\frac{2X + Y}{4}$$

IOActive™

Scale Free Description



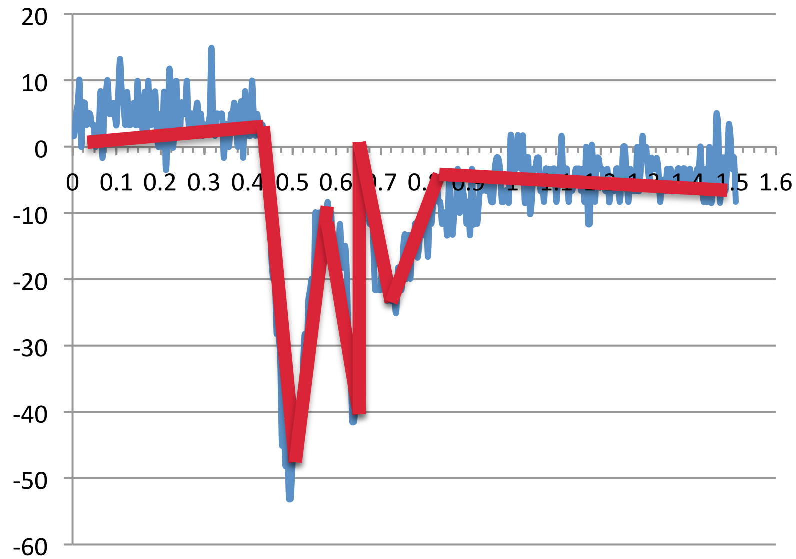
Modeling says it should look like this

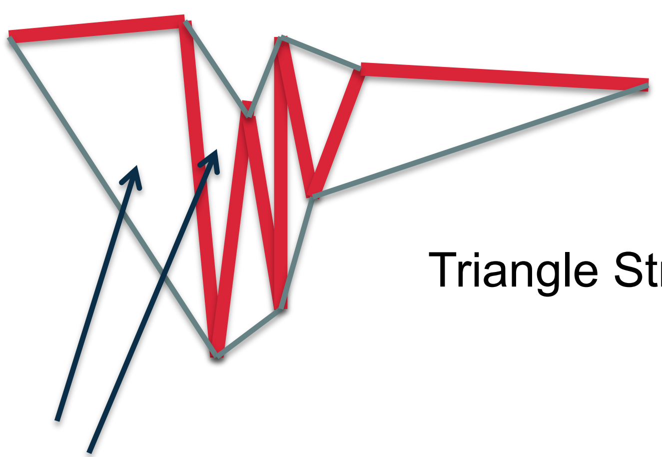


It actually looks like this

We need to cycle the valve at:
X=(0.55-0.4)
Y=(0.65-0.55)
2X+Y=0.4 seconds

Line Segments





Triangle Strips

Ratio of areas between adjacent triangles
(I could have also used ratios of the angles)

Scale Free Description-
Ratio of areas of adjacent triangles

.31-.33
.29-.33
.21-.29

Triangles

- How big is the triangle algorithm? We have to fit it into a pressure sensor, after all.
  - Approx 700 bytes (Ouch!)

Total Size

- Sensor Noise ~ 400 bytes
- Triangles ~ 700 bytes
- DNP CRC – 272 bytes (ouch!)
- Protocol and Glue Logic ~ 600 bytes

- Total Payload – 2174 bytes
  - That's about 0.7% of the total flash

IOActive™

Act II – Inserting the Code into the Firmware

Popcorn Alert! Lots of assembly ahead

Inserting the Rootkit into the Firmware

- I still need to make my payload smaller
  - To make it smaller, I need to reuse the existing code.
- Debugging
  - If I'm reusing existing code, how do I debug it?
  - What if the existing code has side effects?
- Portability
  - I don't want to recode my rootkit for every single sensor I want to invade.

Parallel Construction

- I'm going to write and debug my attack code on my MacBook (X86), debug it, and then deploy it on an pressure sensor (MSP430).
- I need to be able to translate between those two different architectures.

Example Code

```
int CalcSomething(int x){
        int total = 0;
        int i;
        for (i=0;i<x;i++){
                total=total+i;
        }
        return total;
}
```


IOActive™

MSP430 Assembly

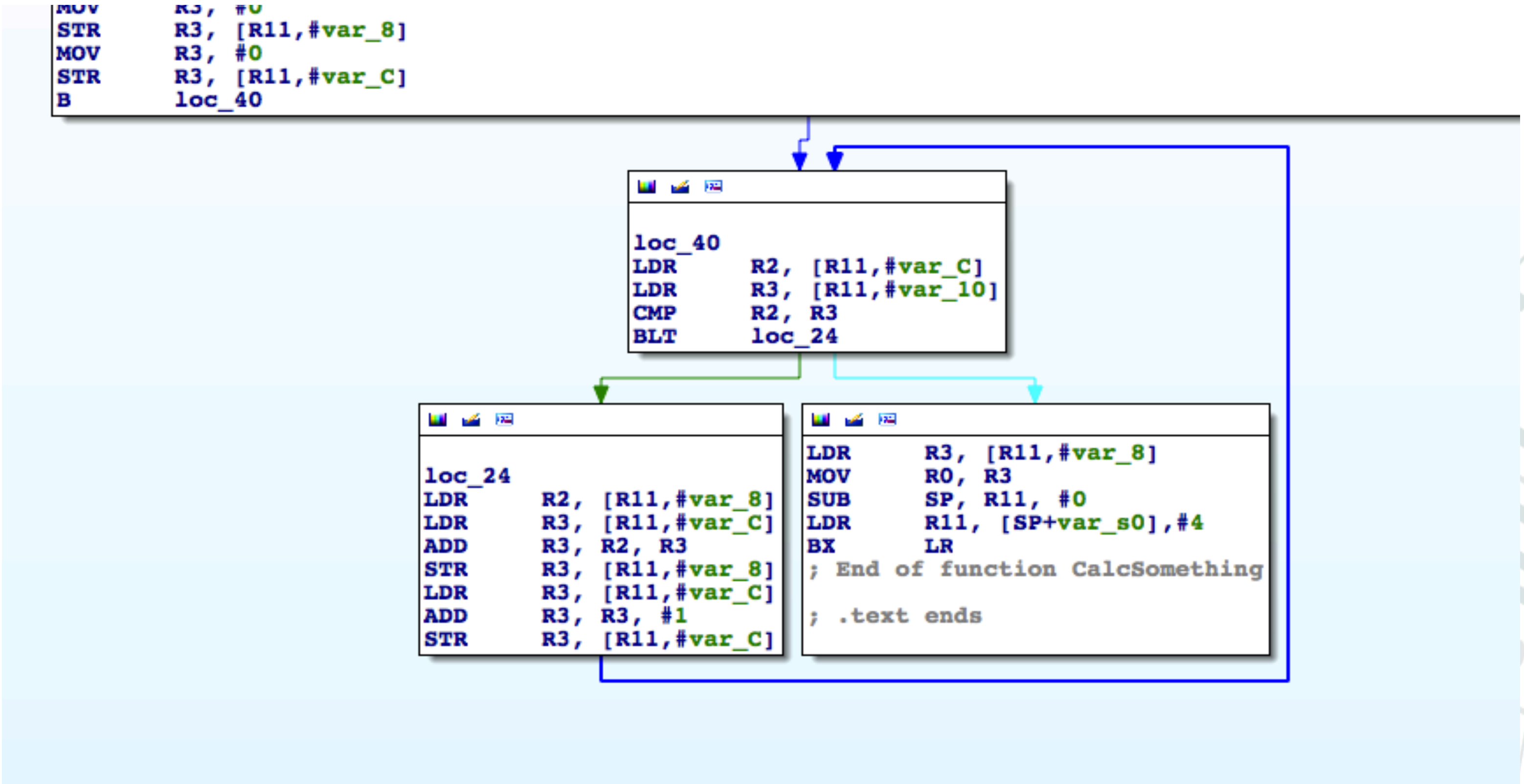
```
.def CalcSomething
CalcSomething:
push.w  R4
mov.w   SP, R4
incd.w  R4
add.w   #0FFFAh, SP
mov.w   R15, 0FFFCh(R4)
clr.w   0FFF8h(R4)
clr.w   0FFFAh(R4)
jmp     loc_22
```

```
loc_22:
cmp.w   0FFFCh(R4), 0FFFAh(R4)
jl      loc_18
```

```
loc_18:
add.w   0FFFAh(R4), 0FFF8h(R4)
inc.w   0FFFAh(R4)
```

```
mov.w   0FFF8h(R4), R15
add.w   #6, SP
pop     R4
ret
; End of function CalcSomething
```

ARM Assembly



Are they different?

- We can't directly compare the two assemblies

VS

IOActive™

```
STR     R11, [SP,#-4+var_s0]!
ADD     R11, SP, #0
SUB     SP, SP, #0x14
STR     R0, [R11,#var_10]
MOV     R3, #0
STR     R3, [R11,#var_8]
MOV     R3, #0
STR     R3, [R11,#var_C]
B       loc_40
LDR     R2, [R11,#var_8]
LDR     R3, [R11,#var_C]
ADD     R3, R2, R3
STR     R3, [R11,#var_8]
LDR     R3, [R11,#var_C]
ADD     R3, R3, #1
STR     R3, [R11,#var_C]
LDR     R2, [R11,#var_C]
LDR     R3, [R11,#var_10]
CMP     R2, R3
LDR     R3, [R11,#var_8]
MOV     R0, R3
SUB     SP, R11, #0
LDR     R11, [SP+var_s0],#4
BX      LR
```

Preamble

Stack Allocation

Argument Linking

Local Variable Initialization

Actual Logic

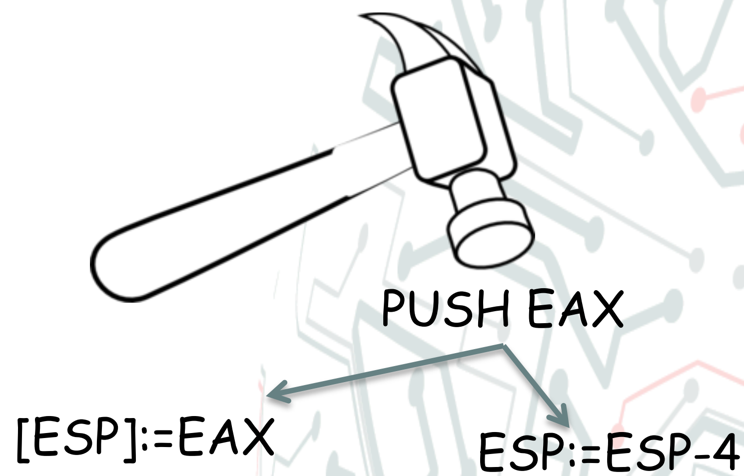
Argument Linking

Postamble

```
push.w  R4
mov.w   SP, R4
incd.w  R4
add.w   #0FFFAh, SP
mov.w   R15, 0FFFCh(R4)
clr.w   0FFF8h(R4)
clr.w   0FFFAh(R4)
jmp     loc_22
add.w   0FFFAh(R4), 0FFF8h(R4)
inc.w   0FFFAh(R4)
cmp.w   0FFFCh(R4), 0FFFAh(R4)
jl      loc_18
mov.w   0FFF8h(R4), R15
add.w   #6, SP
pop     R4
ret
```


IOActive

- Miniaturizing the Attack Code
  - Spoofing with Runs Analysis
  - Triangles for Filtering Noise
  - Scale-free Matching for Watching the Process
- Inserting the Attack Code into the Firmware
  - MicroOps
  - Binary Normal Form
  - Abusing Needleman Wuncsh to Merge Firmware
  - Metasploit for Firmware
- Demos



MicroOps

- Assembly language operations are actually complex
  - They can be described using several smaller operations
- Push EAX is actually complex instruction with two operations
  - Subtract 4 from the stack pointer
  - Move EAX into the memory pointed to by the stack pointer

PUSH EAX → ESP:=ESP-4
[ESP]:=EAX

```
MOV     R3, #0
STR     R3, [R11,#var_8]
MOV     R3, #0
STR     R3, [R11,#var_C]
B       loc_40
LDR     R2, [R11,#var_8]
LDR     R3, [R11,#var_C]
ADD     R3, R2, R3
STR     R3, [R11,#var_8]
LDR     R3, [R11,#var_C]
ADD     R3, R3, #1
STR     R3, [R11,#var_C]
LDR     R2, [R11,#var_C]
LDR     R3, [R11,#var_10]
CMP     R2, R3
BLT     loc_24
```

```
R3:=0
[R11+8]:=R3
R3:=0
[R11+C]:=R3
PC:=loc_40
R3:=[R11+8]
R3:=[R11+C]
R3:=R2+R3
[R11+8:]:=R3
R3:=R11+C]
R3:=R3+1
[R11+C]:=R3
R2:=[R11+C]
R3:=[R11+10]
IF   R2< R3 THEN PC:=loc_24
```

```
clr.w   0FFF8h(R4)
clr.w   0FFFAh(R4)
jmp     loc_22
add.w   0FFFAh(R4), 0FFF8h(R4)
inc.w   0FFFAh(R4)
cmp.w   0FFFCh(R4), 0FFFAh(R4)
jl      loc_18
```

```
[R4+8]:=0
[R4+10]:=0
PC:=loc_22
[R4+8]:=[R4+10]+[R4+8]
[R4+10]:=[R4+10]+1
IF [R4+10]<[R4+8] THEN
PC:=loc_18
```

Let's break these two down into MicroOps

Apples->Pears
Oranges->Pears

Now They are the same language!

But….Not exactly the same yet

- Miniaturizing the Attack Code
  - Spoofing with Runs Analysis
  - Triangles for Filtering Noise
  - Scale-free Matching for Watching the Process
- Inserting the Attack Code into the Firmware
  - MicroOps
  - Binary Normal Form
  - Abusing Needleman Wuncsh to Merge Firmware
  - Metasploit for Firmware
- Demos

Binary Normal Form (BNF)

- What we need is a set of rules. Tame the chaos.
- I call this set of rules Binary Normal Form
- We apply all the rules, we have a good chance of converting the structure of the two MicroOp trees into the same tree.

Binary Normal Form

1. All loads and stores are via a register.
2. All branches are positive form “Jump if Equal” instead of “Jump if not Equal”.
3. The true branch always comes first (Jump to false).
4. ……

Binary Normal Form

```
R3:=0
[R11+8]:=R3
[R11+C]:=R3
PC:=loc_40
R3:=[R11+8]
R2:=[R11+C]
R3:=R2+R3
[R11+8:]:=R3
R3:=[R11+C]
R3:=R3+1
[R11+C]:=R3
R2:=[R11+C]
R3:=[R11+10]
IF   R2< R3 THEN PC:=loc_24
```

```
TMP1:=0
[R4+8]:=TMP1
[R4+10]:=TMP1
PC:=loc_22
TMP1:=[R4+8]
TMP2:=[R4+10]
TMP3:=TMP1+TMP2
[R4+8]:=TMP3
TMP1:=[R4+10]
TMP1:=TMP1+1
[R4+10]:=TMP1
TMP1:=[R4+8]
TMP2:=[R4+10]
IF TMP1<TMP2 THEN PC:=loc_18
```

It's not an exact match. They use different registers and different stack offsets. Compilers may have ordered things differently.

Excellent! They kinda match!!

Infinite Register File

- What can we do to normalize the registers and stack variables?
- It would be a shame we couldn't compare two chunks of code simply because the compiler chose a different register.
- If there were an infinite number of registers, a compiler would never need to reuse them.
  - There would also be no need for stack variables.

Infinite Register File

```
STR     R11, [SP,#-4+var_s0]!
ADD     R11, SP, #0
SUB     SP, SP, #0x14
STR     R0, [R11,#var_10]
MOV     R3, #0
STR     R3, [R11,#var_8]
MOV     R3, #0
STR     R3, [R11,#var_C]
B       loc_40
LDR     R2, [R11,#var_8]
LDR     R3, [R11,#var_C]
ADD     R3, R2, R3
```

Allocate 1 Register for the new base pointer

Becomes move stack into base

Allocate 5 Registers

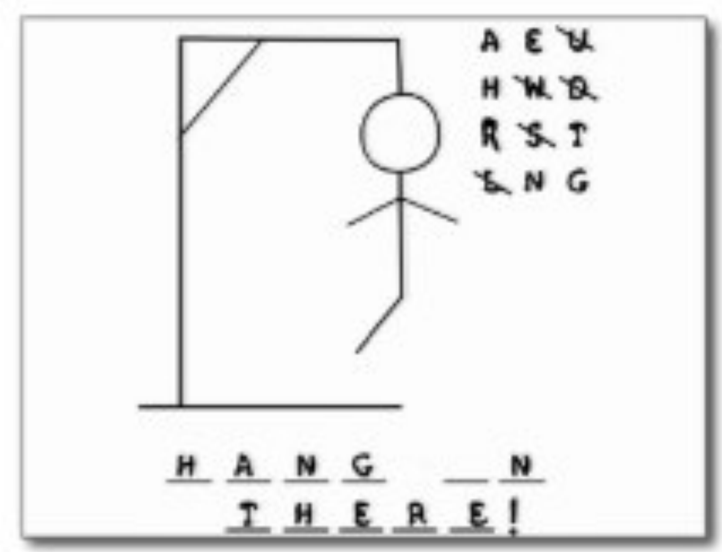
Becomes move zero into a register

Infinite Register File

S1:=0	S1:=0
S2:=0	S2:=0
PC:=PC+2	PC:=PC+2
S1:=S1+S2	S1:=S1+S2
S2:=S2+1	S2:=S2+1
IF S2< ARG1 THEN PC:=PC-2	IF S2<ARG1 THEN PC:=PC-2

- Nasty stack operations are eliminated
- The two code segments match!
- We can say that they are the same logic (minus the register width).

- Miniaturizing the Attack Code
  - Spoofing with Runs Analysis
  - Triangles for Filtering Noise
  - Scale-free Matching for Watching the Process
- Inserting the Attack Code into the Firmware
  - MicroOps
  - Binary Normal Form
  - Abusing Needleman Wuncsh to Merge Firm
  - Metasploit for Firmware
- Demos



Modified Code

```
int CalcSomething(int x){
        int total = 0;
        int i;

        for (i=0;i<x;i++){
                total=total+i;
        }
        return total;
}
```

```
int EvilSomething(int x){
        int total = 0;
        int i;

        for (i=0;i<x;i++){
                total=total+i+4;
        }
        return total;
}
```

What if I made some changes?

Edit Distance

```
S1:=0
S2:=0
PC:=PC+2
S1:=S1+S2
S2:=S2+1
IF    S2< ARG1 THEN PC:=PC-2
```

```
S1:=0
S2:=0
PC:=PC+3
S3:=S2+4
S1:=S1+S3
S2:=S2+1
IF S2<ARG1 THEN PC:=PC-3
```

- How close are these two functions?
- One way to measure that is the edit distance
  - How many IF statements would it take to make them the same?

Edit Distance

```
S1:=0
S2:=0
PC:=loc_40
S1:=S1+S2
S2:=S2+1
IF    S2< ARG1 THEN PC:=loc_24
```

```
S1:=0
S2:=0
PC:=PC+3
S3:=S2+4
S1:=S1+S3
S2:=S2+1
IF S2<ARG1 THEN PC:=PC-3
```

```
S1:=0
S2:=0
PC:=PC+6
IF ARG2 THEN
    S1:=S1+S2
ELSE
    S3:=S2+4
    S1:=S1+S3
S2:=S2+1
IF    S2< ARG1 THEN PC:=PC-6
```

These two functions differ with an edit distance of 1

Edit Distance

- That was a trivial example
- How can we find the edit distance between two pieces of code in a more generic way?
- We can steal from the biologists and use protein matching algorithms
  - Needleman-Wunsch can be used to find the edit distance between two strings
- We can adapt that for our uses

Needleman Wunsch

```
Inserting Code is Fun
Inserting Rootkits is Fun
```

```
Inserting _Co___de is Fun
Inserting Rootkits is Fun
```

Needleman Wunsch

```
Inserting Code is Fun
Inserting Rootkits is Fun
```

```
Inserting _Co___de is Fun
Inserting Rootkits is Fun
```

The strings have an edit distance of 2

18 Characters the same - 10 characters different

Edit Distances Between Functions

```
S1:=0
S2:=0
PC:=PC+2
S1:=S1+S2
S2:=S2+1
IF    S2< ARG1 THEN PC:=PC-2
```

```
S1:=0
S2:=0
PC:=PC+3
S3:=S2+4
S1:=S1+S3
S2:=S2+1
IF S2<ARG1 THEN PC:=PC-3
```

- What if we turned these MicroOps into letters?
- We could calculate the edit distance between any two functions
- It would even tell us where to put the IF statements

Edit Distances Between Functions

S1:=0	MS0
S2:=0	MS0
PC:=PC+2	MR+
S1:=S1+S2	ASS
S2:=S2+1	AS1
IF S2< ARG1 THEN PC:=PC-2	ICLTSAMR-

S1:=0	MS0
S2:=0	MS0
PC:=PC+3	MR+
S3:=S2+4	AS4
S1:=S1+S3	ASS
S2:=S2+1	AS1
IF S2< ARG1 THEN PC:=PC-3	ICLTSAMR-

```
MS0MS0MR+ASSAS1ICLTSARM-
MS0MS0MR+AS4ASSAS1ICLTSARM-
```

```
MS0MS0MR+___ASSAS1ICLTSARM-
MS0MS0MR+AS4ASSAS1ICLTSARM-
```

Edit Distances Between Functions

S1:=0	MS0
S2:=0	MS0
PC:=loc_40	MR+2
S1:=S1+S2	ASS
S2:=S2+1	AS1
IF S2< ARG1 THEN PC:=loc_24	ICLTSAMR-2

S1:=0	MS0
S2:=0	MS0
PC:=loc_40	MR+2
S3:=S2+4	AS4
S1:=S1+S3	ASS
S2:=S2+1	AS1
IF S2< ARG1 THEN PC:=loc_24	ICLTSAMR-2

MS0MS0MR+2___ASSAS1ICLTSARM-2

MS0MS0MR+2AS4ASSAS1ICLTSARM-2

The string shows where to add the IF statements to make the functions that same.

Edit Distances

MS0MS0MR+2___ASSAS1ICLTSARM-2
MS0MS0MR+2AS4ASSAS1ICLTSARM-2

```
S1:=0
S2:=0
PC:=loc_40
IF ARG2 THEN
    S1:=S1+S2
ELSE
    S3:=S2+4
    S1:=S1+S3
S2:=S2+1
IF    S2< ARG1 THEN
PC:=loc_24
```

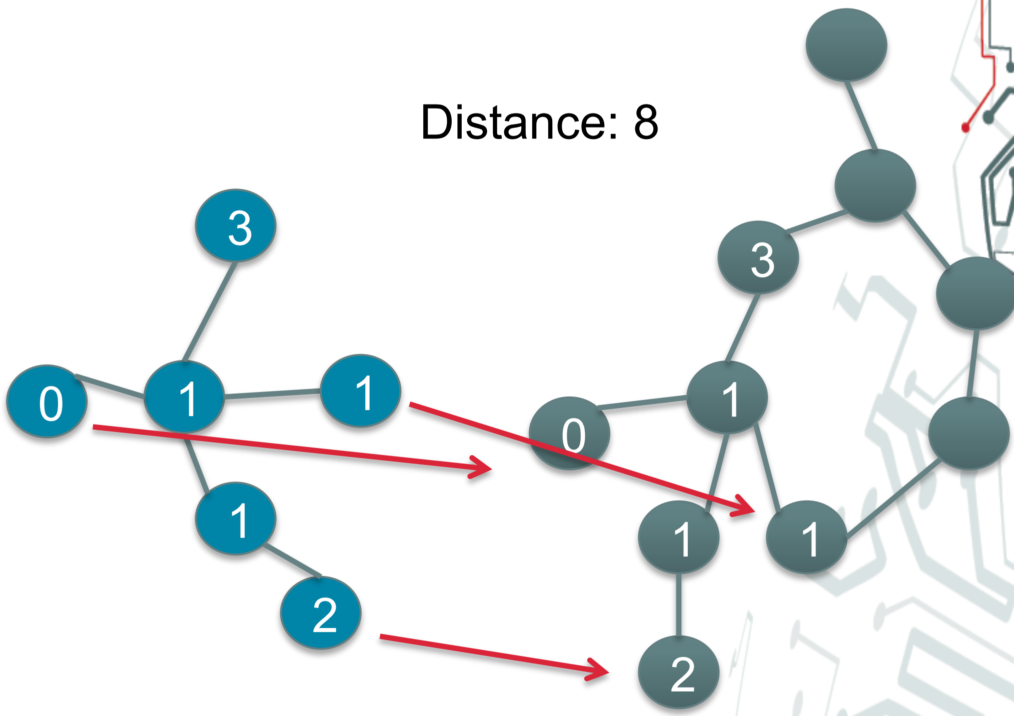
If we know how costly an IF statement is in the target architecture, we can figure out if merging these two function will save space in the final firmware.

Matching Call Trees

- Now that we can match two functions, why not something bigger?
- We can take each of our leaf functions and see if the parents of that leaf function also match.

Matching Call Trees

After matching, it is now possible to calculate the edit distance of an entire subsystem



Finally!
Inserting the Rootkit into the Firmware!

- For each function in the rootkit, I have found a best match function in the target firmware
  - If mine has a CRC-16 and the target has a CRC-16, they will have a small edit distance and get merged together
  - Any orphans that simply don't match will need to be added to the end
- I can even merge two functions in the target together to gain even more space
- Now simply reverse the process from BNF back to the target assembly
- Instant firmware rootkit!

```
uint16_t crc16_update(uint16_t crc, uint8_t a){
      int i;
      crc ^= a;
      for (i = 0; i < 8; ++i){
         if (crc & 1)
            crc = (crc >> 1) ^ 0xA001;
         else
            crc = (crc >> 1);
      }
      return crc;
   }
```


IOActive™

- Miniaturizing the Attack Code
  - Spoofing with Runs Analysis
  - Triangles for Filtering Noise
  - Scale-Free Matching for Watching the Process
- Inserting the Attack Code into the Firmware
  - MicroOps
  - Binary Normal Form
  - Abusing Needleman Wuncsh to Merge Firmware
  - Metasploit for Firmware
- Demos

Random Code from the Internet

- Nobody would ever just copy code from the Internet would they?
- Since we can compare code, we can search to see if this code is in that firmware

www.experts-exchange.com/Programming/Languages/Visual_Basic

Experts Exchange

```
0xf5e2, 0xc3bc, 0x995e, 0xaf00, 0x2c9a, 0x1ac4, 0x4026, 0x7678,
0x0a6b, 0x3c35, 0x66d7, 0x5089, 0xd313, 0xe54d, 0xbfaf, 0x89f1,
0x4789, 0x71d7, 0x2b35, 0x1d6b, 0x9ef1, 0xa8af, 0xf24d, 0xc413,
0xb800, 0x8e5e, 0xd4bc, 0xe2e2, 0x6178, 0x5726, 0x0dc4, 0x3b9a,
0xdc4d, 0xea13, 0xb0f1, 0x86af, 0x0535, 0x336b, 0x6989, 0x5fd7,
0x23c4, 0x159a, 0x4f78, 0x7926, 0xfabc, 0xcce2, 0x9600, 0xa05e,
0x6e26, 0x5878, 0x029a, 0x34c4, 0xb75e, 0x8100, 0xdbe2, 0xedbc,
0x91af, 0xa7f1, 0xfd13, 0xcb4d, 0x48d7, 0x7e89, 0x246b, 0x1235
};


static unsigned short B013_dnpcrc(unsigned char *p, unsigned int count)
{
  unsigned crc = 0;
  while(count--) crc = (crc >> 8) ^ crctable[(crc ^ *p++) & 0x00ff];
  return ~crc;
}
```

IOActive

Future: Metasploit for Firmware

- There are common pieces of software used throughout industrial control systems.
  - i.e. SquareD DNP stack
- As long as our rootkit only needs functionality from the common piece of code, the merge will be self-contained.
  - It can be inserted automatically without a human
  - No need to understand the CPU
  - No need to deal with the version differences

IOActive™

Demos

Questions

- Jason Larsen
- IOActive, Inc.