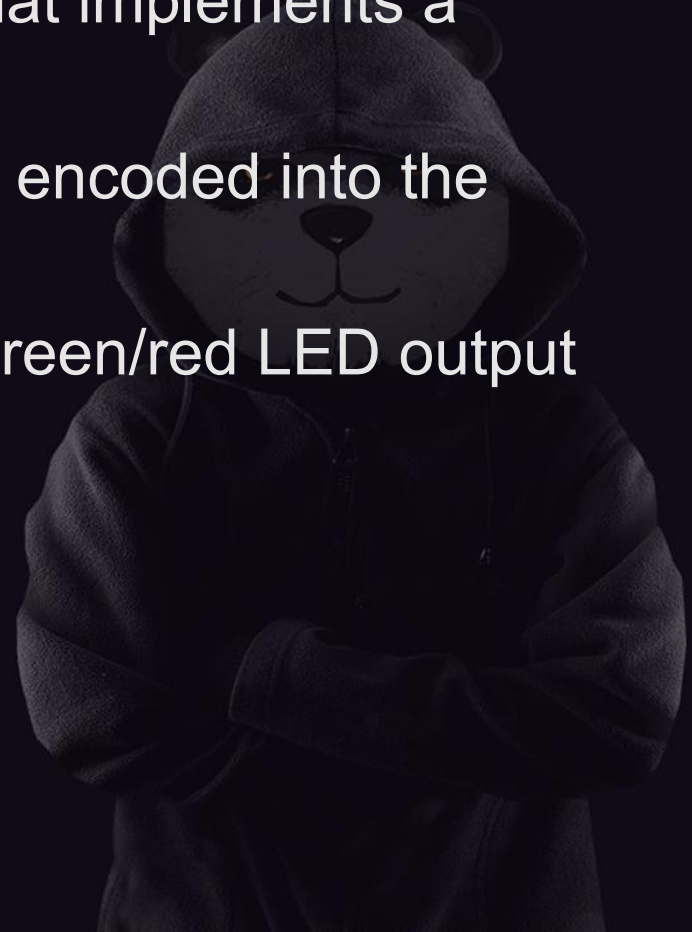
TPM2137

(author: q3k, presented by: implr)

TPM2137 (q3k)

This challenge is about FPGAs and their bitstreams.

- Teams are given a bitstream for an Lattice iCE40 device that implements a simple **password checker**
- This is an offline reverse engineering challenge (the flag is encoded into the bitstream)
- The device has UART input for the password and simple green/red LED output
- **If the green LED lights up, flag is correct**



TPM2137 (q3k)

RadomSemi™

Engineering Your Budget

TPM2137

Secure Passkey Verification

OVERVIEW

Offering the best balance of cyber and price, the TPM2137 offers a simple yet secure solution for password and secret checking in your application.

RadomSemi™ offers full customizability on the password that the device verifies, as long as the password is exactly 8 characters.

FEATURES

- Industry standard UART idle-high receive-only interface at 115200 baud.
- Single 12MHz clock source.
- Simple ‘ok’/’wrong’ LED output pins, active low.
- Based on Truly Unhackable™ FPGA Technology.

Refer to the *TPM2137 Secure Passkey*

TPM2137 (q3k)

A lot has happened around reverse engineering FPGA bitstreams in recent years.

- **iCE40 bitstreams** have been reverse engineered by Project IceStorm http://www.clifford.at/icestorm/
- IceStorm includes iceunpack/icebox_vlog tools that can transform bitstream files back to **structural** Verilog
  - (basically muxes, AND/OR/NOT gates and flip-flops)
- Open-source formal verification tools such as SymbiYosys have appeared

TPM2137 (q3k)

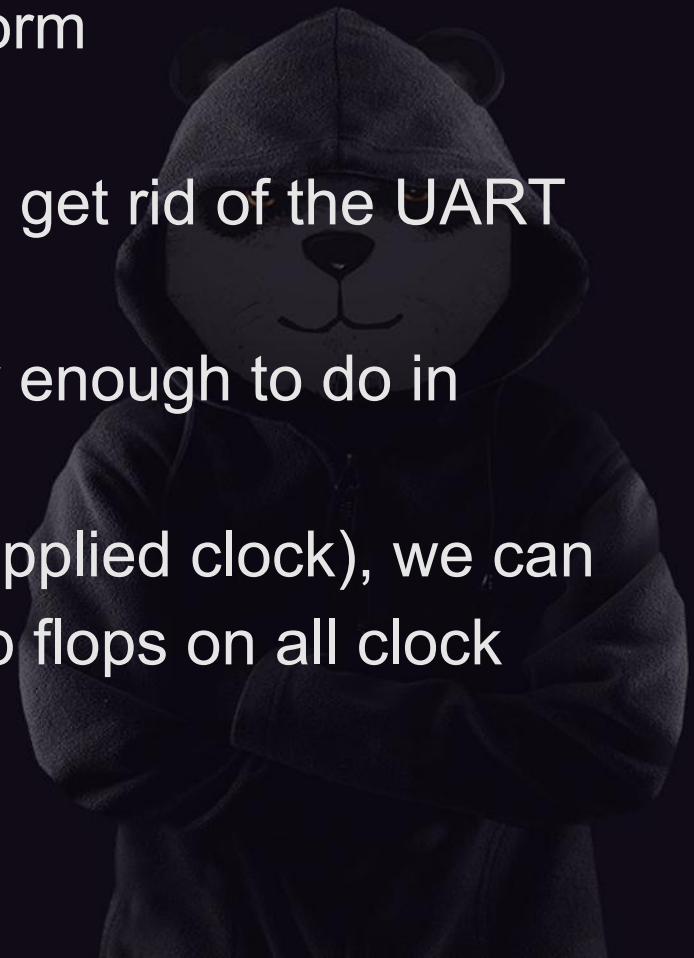
```
wire n390, n391, n392, n393, n394, n395, n396, n397, n398, n399;
wire n400, n401, n402, n403, n404, n405, n406, n407, n408, n409;
wire n410, n411, n412, n413, n414, n415, n416, n417, n418, n419;
wire n420, n421, n422, n423, n424, n425, n426, n427, n428, n429;
wire n430, n431, n432, n433, n434, n435, n436, n437, n438, n439;
wire n440, n441, n442, n443, n444, n445;
reg n5 = 0, n6 = 0, n8 = 0, n9 = 0, n10 = 0, n11 = 0, n12 = 0, n14 = 0, n17 = 0, n18 = 0;
reg n19 = 0, n21 = 0, n22 = 0, n23 = 0, n24 = 0, n26 = 0, n28 = 0, n29 = 0, n31 = 0;
reg n39 = 0, n40 = 0, n27 = 0;
reg n32 = 0, n34 = 0, n35 = 0, n36 = 0, n37 = 0,  n41 = 0, n42 = 0, n43 = 0;
reg n45 = 0, n47 = 0, n48 = 0, n49 = 0, n52 = 0, n53 = 0, n55 = 0, n56 = 0, n57 = 0, n58 = 0;
reg n59 = 0, n60 = 0, n73 = 0,  n86 = 0, n88 = 0, n90 = 0, n92 = 0, n93 = 0;
reg n62 = 0, n63 = 0, n64 = 0, n65 = 0, n67 = 0, n68 = 0, n69 = 0, n70 = 0;
```

```
assign n182 = /* CARRY 13 14  3 */ (n133 & n72) | ((n133 | n72) & n181);
assign n184 = /* CARRY 13 14  5 */ (n177 & n72) | ((n177 | n72) & n183);
assign n221 = /* CARRY 16 14  0 */ (n215 & 1'b0) | ((n215 | 1'b0) & n354);
assign n219 = /* CARRY 16 14  2 */ (n72 & n206) | ((n72 | n206) & n218);
assign n193 = /* CARRY 13 19  1 */ (n83 & n72) | ((n83 | n72) & n220);
assign n222 = /* CARRY 13 14  0 */ (n131 & 1'b0) | ((n131 | 1'b0) & n393);
assign n181 = /* CARRY 13 14  2 */ (n178 & n72) | ((n178 | n72) & n180);
assign n183 = /* CARRY 13 14  4 */ (n72 & n175) | ((n72 | n175) & n182);
/* FF 14 14  5 */ always @(posedge clk) if (n74) n178 <= n3 ? 1'b0 : n223;
/* FF 12 12  5 */ always @(posedge clk) if (n1) n69 <= n3 ? 1'b0 : n224;
/* FF 13 21  3 */ always @(posedge clk) if (n4) n156 <= n3 ? 1'b0 : n225;
/* FF 15 14  0 */ always @(posedge clk) if (n137) n205 <= n3 ? 1'b0 : n226;
/* FF 15 11  2 */ always @(posedge clk) if (n1) n196 <= n3 ? 1'b0 : n227;
/* FF 13 10  4 */ always @(posedge clk) if (n1) n109 <= n3 ? 1'b0 : n228;
/* FF 13 16  6 */ always @(posedge clk) if (n135) n136 <= 1'b0 ? 1'b0 : n229;
/* FF 14 12  4 */ assign n163 = n230;
/* FF 12 12  1 */ assign n66 = n231;
/* FF 13 21  7 */ always @(posedge clk) if (n4) n159 <= n3 ? 1'b0 : n232;
```

TPM2137 (q3k)

How to solve the task (dynamically but manually):

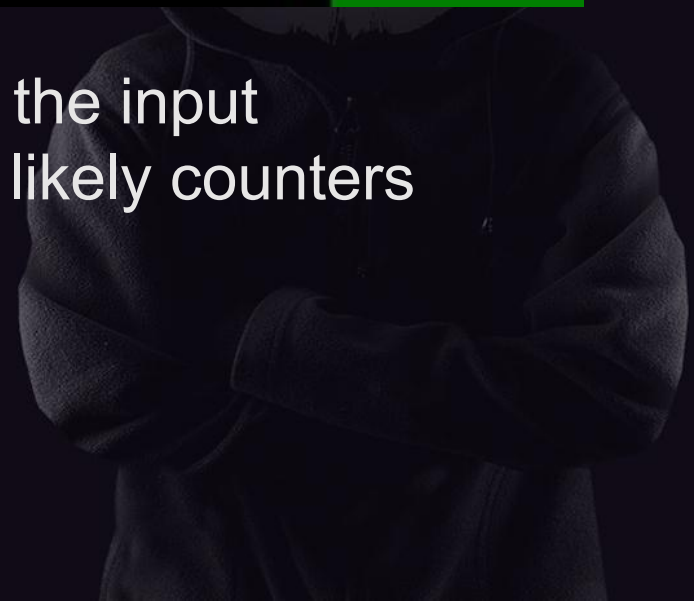
- First, **convert the bitstream to Verilog** with project IceStorm
- Now, we have to **analyze** the resulting circuit
- The input is in the form of UART waveforms — we need to get rid of the UART circuit and focus on the actual checker circuit
- We need a testbench that emits UART waveforms — easy enough to do in Verilator
- Since the whole circuit is synchronous (using externally supplied clock), we can visualize the state of the circuit by showing values of all flip flops on all clock cycles



TPM2137 (q3k)

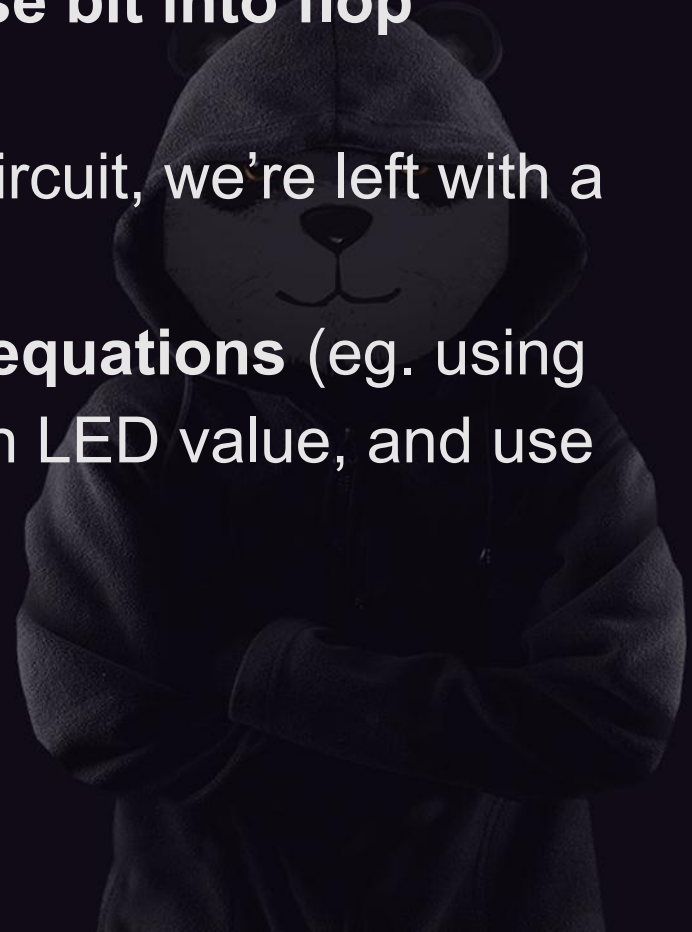
It looks something like this:

- By varying UART input bytes, we can find the flops storing the input
- The blue lines don’t depend on UART byte values and are likely counters



TPM2137 (q3k)

- Once the input flops are identified, we can replace them with external inputs to interface with the password checker directly
  - We now know the **mapping of an inputted passphrase bit into flop name**.
- After this substitution and removing the now-dead UART circuit, we're left with a combinatorial circuit
- It is now enough to **convert the circuit to a set of SMT2 equations** (eg. using yosys-smtbmc or SymbiYosys), add an assert on the green LED value, and use SMT2 solver to find the correct input (which is the flag)
  - If you know angr, this technique might be familiar



TPM2137 (q3k)

How to solve the task (statically but automatically):

Use Yosys to simplify and convert circuit to JSON:

```
yosys -p 'read_verilog challenge.v; synth -noabc; write_json challenge.json'
```

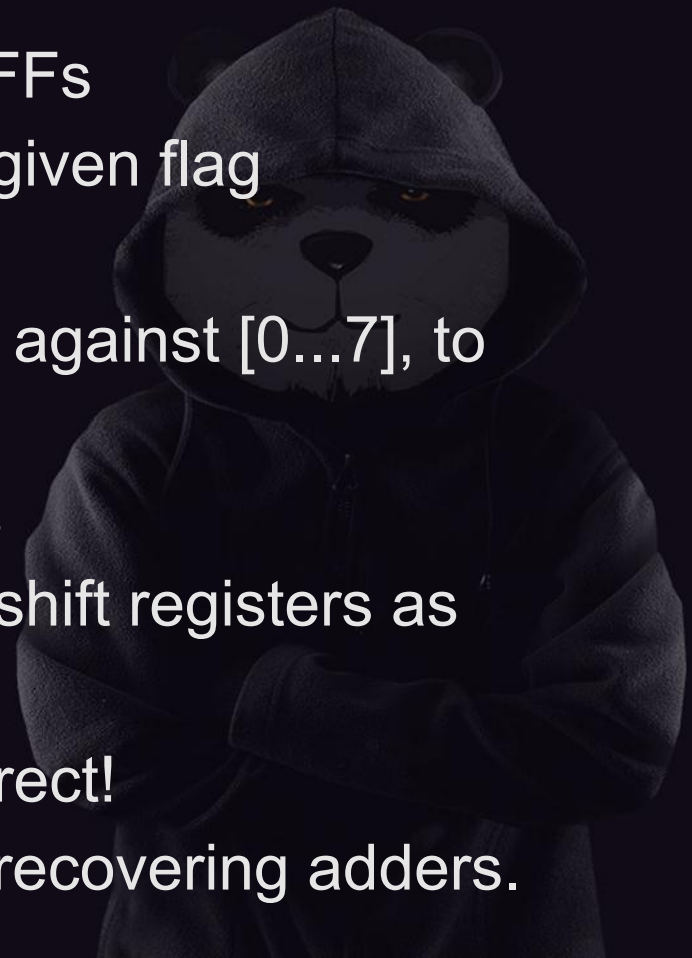
Now, we can write some Python to analyze the JSON for us!

This is of course unnecessary, but it's always good to explore RE automation, even during CTFs!

TPM2137 (q3k)

Write Python automation to:

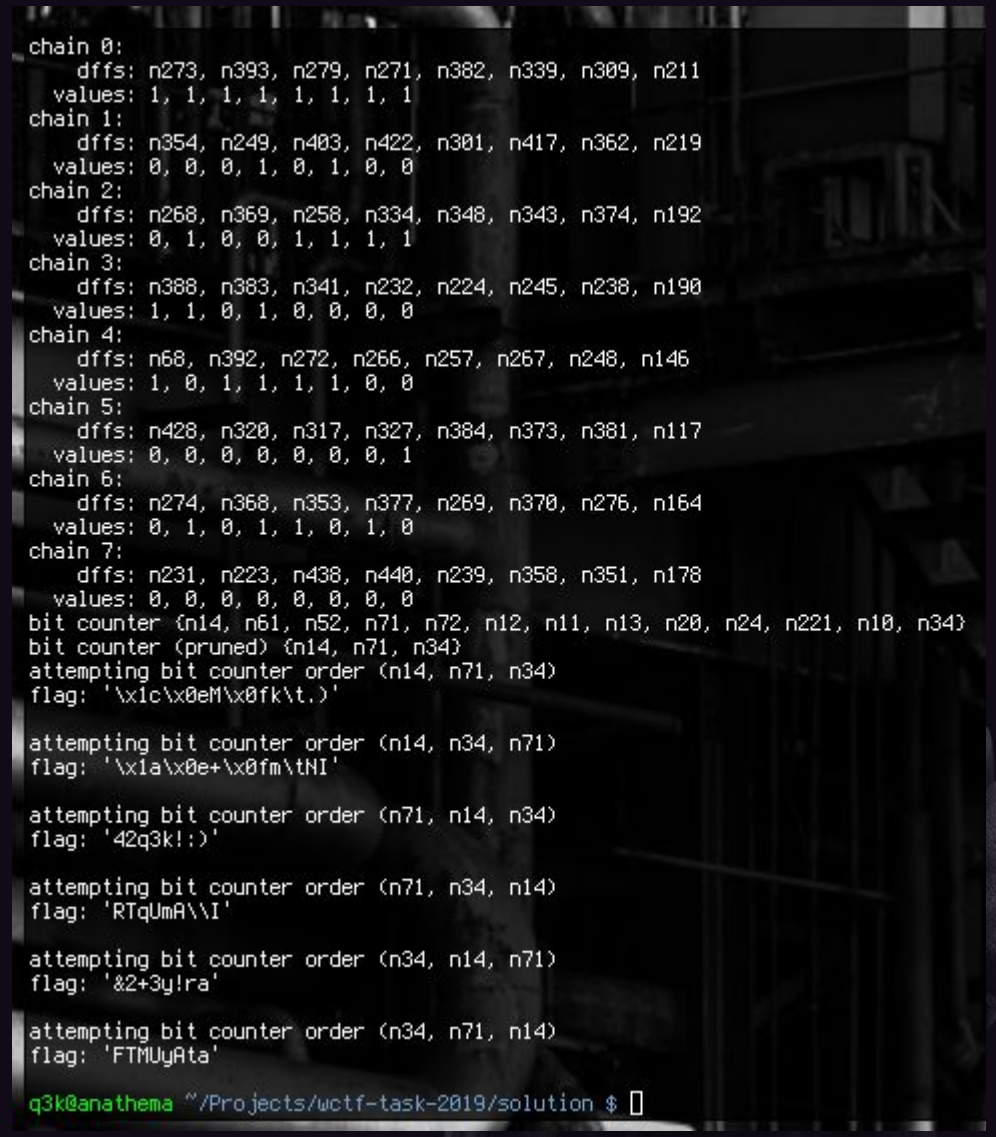
- Recover flag FFs by traversing combinatorial logic from green LED
- Solve flag FFs by using Z3 (64 bits of flag data)
- Recover bit **shift registers** by finding edges that connect FFs
  - There are 8 8-bit shift registers, one for each bit of the given flag
- Find **selectors** for shifting in UART into shift registers
  - Selectors are combinatorial logic to check a bit counter against [0...7], to direct UART data into the corresponding shift register
- Find **counter bits** (3 FFs) that drive shift register selectors
- Try all possible orders of counter bits (3! == 6) to interpret shift registers as particular bit numbers
  - This gives us 6 possible flag values, one of them is correct!
  - We could determine the order of bits, but that requires recovering adders.



TPM2137 (q3k)


q3k@anathema ~/Projects/wctf-task-2019/solution $ make
~/CTF/venv-pypy/bin/python solve.py "challenge.json"
net_ok pin_11
solved flag dff bits:
n279: 1
n339: 1
n271: 1
n382: 1
n309: 1
n422: 1
n417: 1
n211: 1
n393: 1
n192: 1
n273: 1
n374: 1
n348: 1
n232: 1
n343: 1
n369: 1
n266: 1
n68: 1
n117: 1
n272: 1
n388: 1
n269: 1
n276: 1
n383: 1
n267: 1
n368: 1
n377: 1
n257: 1
n239: 0
n178: 0
n351: 0
n358: 0
n440: 0
n223: 0
n231: 0
n438: 0
n354: 0
n249: 0
n258: 0
n334: 0

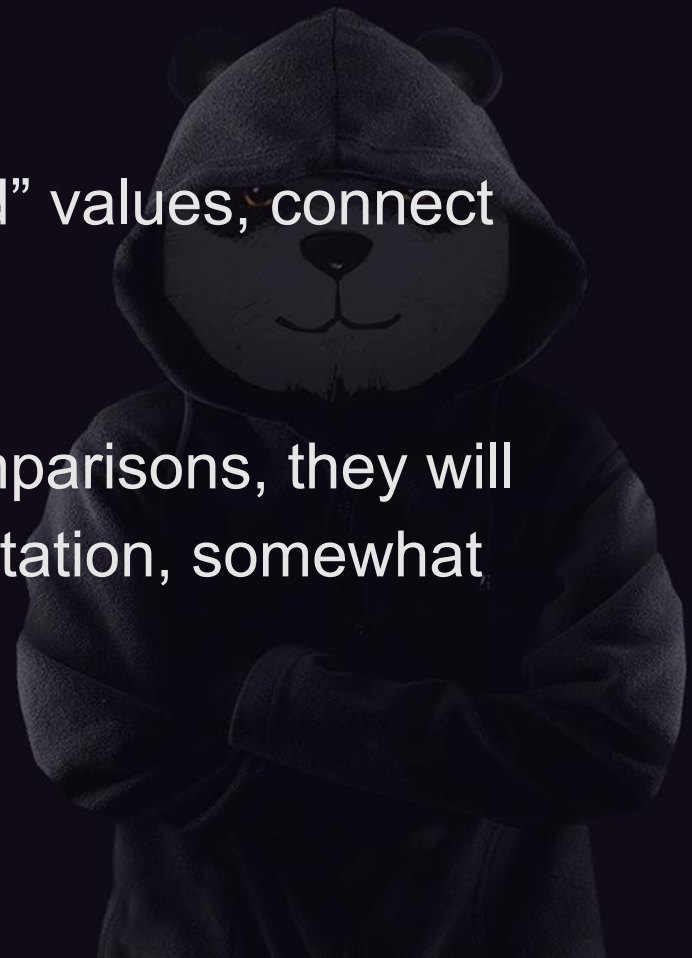

n392: 0
n373: 0
n381: 0
connectivity n393 -> n279
connectivity n382 -> n339
connectivity n279 -> n271
connectivity n271 -> n382
connectivity n339 -> n309
connectivity n403 -> n422
connectivity n301 -> n417
connectivity n309 -> n211
connectivity n273 -> n393
connectivity n374 -> n192
connectivity n420 -> n273
connectivity n343 -> n374
connectivity n334 -> n348
connectivity n341 -> n232
connectivity n348 -> n343
connectivity n268 -> n369
connectivity n272 -> n266
connectivity n79 -> n68
connectivity n381 -> n117
connectivity n392 -> n272
connectivity n379 -> n388
connectivity n377 -> n269
connectivity n370 -> n276
connectivity n388 -> n383
connectivity n257 -> n267
connectivity n274 -> n368
connectivity n353 -> n377
connectivity n266 -> n257
connectivity n440 -> n239
connectivity n351 -> n178
connectivity n358 -> n351
connectivity n239 -> n358
connectivity n438 -> n440
connectivity n231 -> n223
connectivity n81 -> n231
connectivity n223 -> n438
connectivity n253 -> n354
connectivity n354 -> n249
connectivity n369 -> n258
connectivity n258 -> n334
connectivity n333 -> n268
connectivity n245 -> n238
connectivity n224 -> n245


chain 0:
dffs: n273, n393, n279, n271, n382, n339, n309, n211
values: 1, 1, 1, 1, 1, 1, 1, 1
chain 1:
dffs: n354, n249, n403, n422, n301, n417, n362, n219
values: 0, 0, 0, 1, 0, 1, 0, 0
chain 2:
dffs: n268, n369, n258, n334, n348, n343, n374, n192
values: 0, 1, 0, 0, 1, 1, 1, 1
chain 3:
dffs: n388, n383, n341, n232, n224, n245, n238, n190
values: 1, 1, 0, 1, 0, 0, 0, 0
chain 4:
dffs: n68, n392, n272, n266, n257, n267, n248, n146
values: 1, 0, 1, 1, 1, 1, 0, 0
chain 5:
dffs: n428, n320, n317, n327, n384, n373, n381, n117
values: 0, 0, 0, 0, 0, 0, 0, 1
chain 6:
dffs: n274, n368, n353, n377, n269, n370, n276, n164
values: 0, 1, 0, 1, 1, 0, 1, 0
chain 7:
dffs: n231, n223, n438, n440, n239, n358, n351, n178
values: 0, 0, 0, 0, 0, 0, 0, 0
bit counter {n14, n61, n52, n71, n72, n12, n11, n13, n20, n24, n221, n10, n34}
bit counter (pruned) {n14, n71, n34}
attempting bit counter order (n14, n71, n34)
flag: '\x1c\x0eM\x0fk\t.)'
attempting bit counter order (n14, n34, n71)
flag: '\x1a\x0e+\x0fm\tNI'
attempting bit counter order (n71, n14, n34)
flag: '42q3k!:)'
attempting bit counter order (n71, n34, n14)
flag: 'RTqUmA\\I'
attempting bit counter order (n34, n14, n71)
flag: '&2+3y!ra'
attempting bit counter order (n34, n71, n14)
flag: 'FTMUyAta'
q3k@anathema ~/Projects/wctf-task-2019/solution $

TPM2137 (q3k)

The design was dead simple (we didn't want the task to be too complex):

- Make a simple standard UART receiver circuit
- Fetch the UART input to registers
- Compare the registers for equality with hardcoded "wanted" values, connect output to LED
- Synthesize & implement with yosys & nextpnr
- While the original check was a simple AND of equality comparisons, they will get converted to LUTs (look-up tables) by FPGA implementation, somewhat complicating the task



TPM2137 (q3k)

```
reg [7:0] want_0 = 8'b01111110;
reg [7:0] want_1 = 8'b00110001;
reg [7:0] want_2 = 8'b00001100;
reg [7:0] want_3 = 8'b10011000;
reg [7:0] want_4 = 8'b00000011;
reg [7:0] want_5 = 8'b11111111;
reg [7:0] want_6 = 8'b11111100;
reg [7:0] want_7 = 8'b00000000;

wire open =   (want_0 == given_0) &&
              (want_1 == given_1) &&
              (want_2 == given_2) &&
              (want_3 == given_3) &&
              (want_4 == given_4) &&
              (want_5 == given_5) &&
              (want_6 == given_6) &&
              (want_7 == given_7);

assign led_green = !open;
assign led_red = open;
```

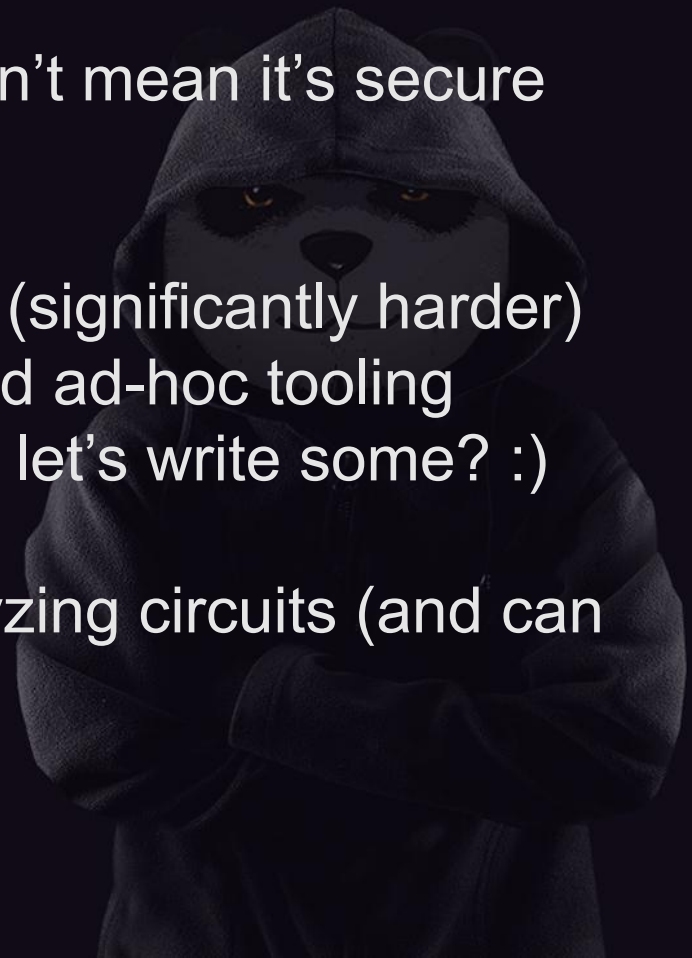
TPM2137 (q3k)

We made this challenge to showcase hardware reverse engineering techniques and open source EDA tools.

- Just because something is implemented in hardware doesn't mean it's secure
  - **Cisco CVE-2019-1649**

- FPGA bitstreams can be dumped, silicon can be analyzed (significantly harder)
  - Bitstream analysis possible using open source tools and ad-hoc tooling
    - … but there's no fully automated tooling yet, maybe let's write some? :)

- Powerful **formal verification tools** exist to help with analyzing circuits (and can be used to find inputs satisfying requested conditions)



TPM2137 (q3k)

Challenge sources and automated solver:

https://github.com/q3k/TPM2137

Thanks!