SecSi Product Development:

Techniques for ensuring Secure Silicon applied to open-source Verilog projects

Joe FitzPatrick (SecuringHardware.com)
joefitz@securinghardware.com

Note

This document is a preliminary revision. For the latest version, please see http://securinghardware.com/secure-opencores

Abstract

Secure development processes for software have formed, developed, and matured in the past decade to the point where there are well defined categories of security bugs and proven methods to find them. Secure hardware development, on the other hand, is essentially undefined at this point. Most developers of integrated circuits do no hardware security validation, or are secretive about their methods and findings.

Three separate Open-Source CPU designs were chosen and their verilog source code was reviewed for common security vulnerabilities. This paper will first introduce each of these designs and then identify security vulnerabilities discovered. This paper will discuss the process that that lead to discovery, potential means for exploiting the flaw, and finally methods of fixing it. This paper will conclude by generalizing these bugs into a checklist of common hardware vulnerabilities

Introduction

Software stacks continue to grow, enabling new and different ways of accessing and utilizing the increasing computing capacity available today. New software layers are gradually being designed with a pro-active approach to security while the supporting layers continue to be refined and secured.

All of this software depends on a solid hardware foundation. From purely a reliability perspective, hardware does exactly that and continues to improve. However, due to a number of factors, integrated circuit devices do not get nearly the same scrutiny in terms of security.

Attacking hardware is inherently slower, more difficult, and more expensive that software. In most cases the attacker must be physically present with the need to carefully observe and manipulate electronics, always with the risk of permanently destroying the target device. This plays a strong part of the 'it's in hardware so we can trust it' mindset that even paranoid software security professionals get caught in.

Background

There is no doubt that many people are doing security validation of hardware designs; however, outside of academia, there is not much public dialog about methods and practices. Newcomers to the field have no reference manuals or seminal works, and no idea of where to start. Established hardware security validation teams are not benefiting from any techniques that are not home grown.

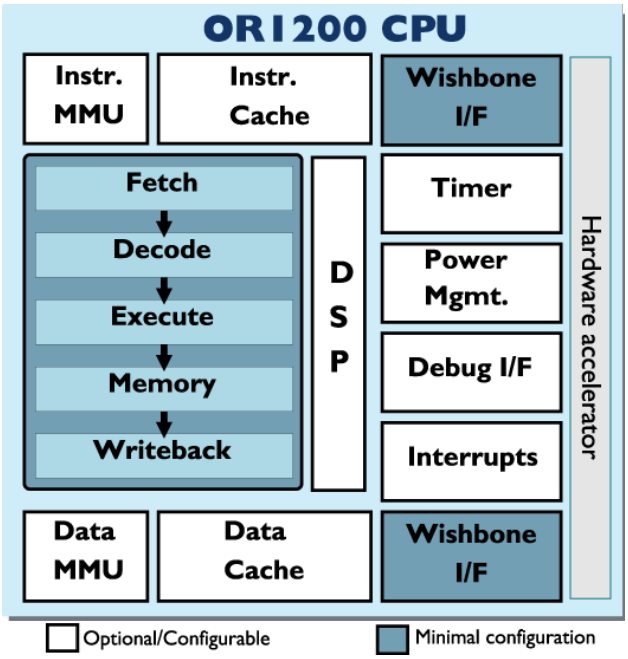
Likewise, while formal verification and other methods promise future solutions to hardware security, the current state of the industry reflects the state of software security 15 years ago where a few minutes of manual inspection can reveal multiple bugs on what is considered a mature and stable system.

In the interest of starting a conversation about hardware security validation methods, This paper will document some of them as applied to actual real-world, publicly-available Verilog code. Three different stable 'Opencores Certified' projects from opencores.org were used as examples in lieu of contrived code blocks or heavily redacted snippets of proprietary code.

OpenRisc

OpenRISC is a completely open-sourced RISC CPU architecture developed by the OpenCores Community. The current implementation is the OpenRISC 1200 written in Verilog. It is considered stable, has mainline Linux kernel support, is used

heavily by academia, and has also been implemented in a number of commercial products.



http://www.rte.se/blog/blogg-modesty-corex/openrisc-1200-soft-processor

Openrisc is combined with several other open components into OpenRISC Reference Platform System-on-Chip or ORPSoC. We conducted a security-minded code review of the latest ORPSoCv2 Verilog source available from Opencores.org. Since this review was of generic Verilog code and not tied to a specific implementation, it's difficult to determine 'exploitability' of any noted issues. Despite that, the issues identified are either likely to be exploitable, or representative of issues that could be exploitable.

One-Hot and JTAG

A common construct in hardware designs is a State Machine. The contents of a register define how the machine behaves, and the machine's behaviour combined with the current state will determine the next state to be loaded.

One-hot encoding is often used for storing the current state of a machine. This means there is a single bit for each state, and only one bit is set at a given time. While this requires extra bits of storage, it eliminates the need for any decoding logic and makes it easier to detect invalid states.

ORPSoCv2 implements JTAG as a one-hot state machine. It defines a single register for each jtag state:

```
// Registers
reg     test_logic_reset;
reg     run_test_idle;
reg     select_dr_scan;
reg     capture_dr;
reg     shift_dr;
reg     exit1_dr;
reg     pause_dr;
reg     exit2_dr;
reg     update_dr;
reg     select_ir_scan;
reg     capture_ir;
reg     shift_ir, shift_ir_neg;
reg     exit1_ir;
reg     pause_ir;
reg     exit2_ir;
reg     update_ir;
```

The implementation of this one-hot state machine is clearly derived directly from the JTAG spec and is in fact functionally compliant. Each state has an always @ block which is triggered on tap reset or tclk, and each of the 16 blocks are structured similarly:

```
// test_logic_reset state
always @ (posedge tck_pad_i or posedge
trst_pad_i)
begin

  if(trst_pad_i)
    test_logic_reset<= 1'b1;
  else if (tms_reset)
    test_logic_reset<= 1'b1;
  else
    begin
      if(tms_pad_i & (test_logic_reset |
select_ir_scan))
        test_logic_reset<= 1'b1;
      else
        test_logic_reset<= 1'b0;
    end
end

// run_test_idle state
always @ (posedge tck_pad_i or posedge
trst_pad_i)
begin
  if(trst_pad_i)
    run_test_idle<= 1'b0;
  else if (tms_reset)
    run_test_idle<= 1'b0;
  else
  if(~tms_pad_i & (test_logic_reset |
run_test_idle | update_dr | update_ir))
    run_test_idle<= 1'b1;
  else
```

```
    run_test_idle<= 1'b0;
end
```

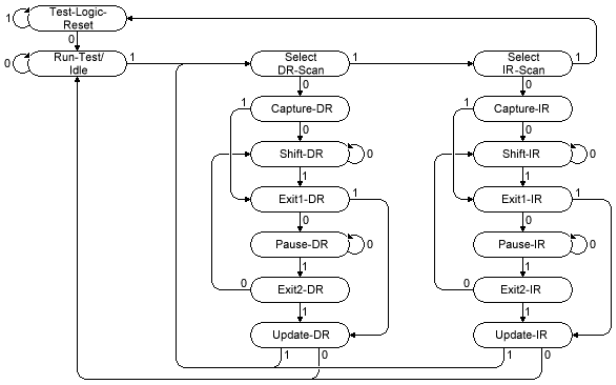
The first assignment sets the state in the event TRST is asserted. The second sets the state if tap reset is triggered via TMS. The third and fourth assignments set the state based on the combination of previous state and the value of TMS.
Functionally, this is correct. Assuming only one JTAG state is asserted, only one next state will be valid. Just like with software, exploitation usually comes down to undermining assumptions. If we can find a way to set multiple state registers to '1' at the same time, we will begin operating this state machine with two active states since there is currently no logic to differentiate between one-hotness or one hot mess.
There is no obvious logical path to set an extra bit in the state. While a few of the state registers are visible as outputs, none are settable as inputs:

```
// TAP states
output  shift_dr_o;
output  pause_dr_o;
output  update_dr_o;
output  capture_dr_o;
...
assign shift_dr_o = shift_dr;
assign pause_dr_o = pause_dr;
assign update_dr_o = update_dr;
assign capture_dr_o = capture_dr;
```

Due to the very similar parallel structure of the state generating blocks, a timing attack is not initially obvious, however there is one outlier:

```
  if(~tms_pad_i & (test_logic_reset |
run_test_idle | update_dr | update_ir))
    run_test_idle<= 1'b1;
```

There are 4 different paths to the run-test-idle state, making this assignment dependent on the combination of 5 separate inputs. Although this depends heavily on syntesis optimization and FPGA architecture, If the FPGA's LUT are only 4-input, it means that run-test-idle's assignment will be the only one with cascading layers of logic. Combined with the fact that TCK is typically an externally-controlled pin, It is conceivable to 'spawn' an extra state bit into run-test-idle by manipulating TMS.
The next step is to determine what exactly to do with this 'ghost' state. This particular flaw is difficult to exploit because of the iterative paths in jtag:



The most obvious way to interfere with JTAG would bet to traverse both the IR and DR paths at the same time. By spawning an extra state during Select-DR-Scan, Exit2-DR, or Exit2-IR, Two adjacent states could exist that would eventually traverse down IR and DR paths simultaneously. The impact would be that TDI data would shift into both the IR and DR at the same time. Examining the code indicates that IR takes precedence over DR on TDO.

```
/
*****************************************
**********************************
*
*
*   Multiplexing TDO data
*
*
*
*****************************************
**********************************/
always @ (shift_ir_neg or exit1_ir or
instruction_tdo or latched_jtag_ir_neg or
idcode_tdo or
          debug_tdi_i or bs_chain_tdi_i or
mbist_tdi_i or
          bypassed_tdo)
begin
  if(shift_ir_neg)
    tdo_pad_o = instruction_tdo;
  else
```

In the end, there's no clear juicy exploit hiding here - yet. JTAG tends to interconnect to nearly everything in a SOC somehow or another:

- What if DR had precedence over IR on TDO?
- What other functions of the chip respond to the current JTAG state?

- Oftentimes, chips enter interesting operating states between Capture-xR and Update-xR states.
- What if these operating states collide?

Beyond JTAG-specific considerations, these observations should apply to all one-hot encodings. If there is enough reason to implement a state as one-hot, there should be enough reason to validate that the state is in fact one-hot, and have a defined failure-recovery reset state. It is important to remember, however, that assertions are not helpful in this regard since they are not synthesized.

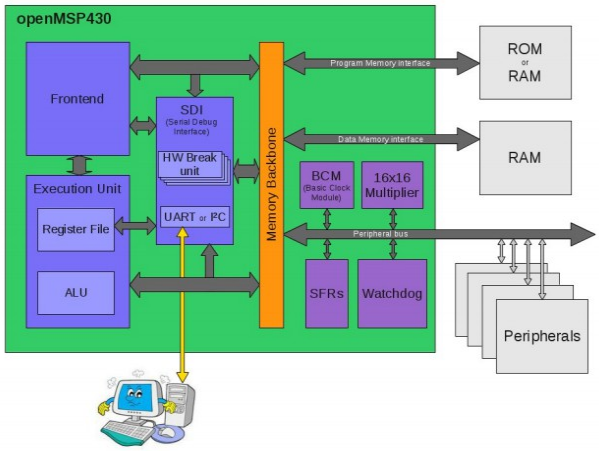
As an added bonus, here's an extra JTAG bug:

```
/
**********************************************
**********************************
*
*
*   idcode logic
*
*
*
**********************************************
**********************************/
reg [31:0] idcode_reg;
reg        idcode_tdo;

always @ (posedge tck_pad_i)
begin
  if(idcode_select & shift_dr)
    idcode_reg <=  {tdi_pad_i,
idcode_reg[31:1]};
  else
    idcode_reg <=  `IDCODE_VALUE;
end

always @ (negedge tck_pad_i)
begin
    idcode_tdo <=  idcode_reg[0]; // JB 100911
end
```

It appears to be fine. If we're in shift_dr, and idcode is the selected IR, shift TDI through the idcode register. Otherwise, reset it. However, this breaks the Pause-DR state's functionality, which is supposed to be sort of an idle state that preserves the contents of DR, while this implementation would obliterate it.

OpenMSP430

openMSP430 is a open-source implementation of TI's 16-bit MSP430 architecture, capable of accurately executing code from any MSP430 toolchain. This represents a much smaller scale processor with a simpler microarchitecture than OpenRISC:



OpenMSP430 may be simpler than OpenRISC, but it still has separate frontend and execution stages. Again, we examined the Verilog code in the Opencores.org SVN repository, looking for potentially exploitable security bugs.

Frontend State Machine Clocking

On first appearance it is clear that openMSP430 has a very different coding style. The state machine for the frontend is contained in a single always@ block as shown:

```
// States Transitions
always @(i_state    or inst_sz  or inst_sz_nxt
or pc_sw_wr or exec_done or
         irq_detect or cpuoff   or
cpu_halt_cmd or e_state)
    case(i_state)
      I_IDLE     : i_state_nxt = (irq_detect &
~cpu_halt_cmd) ? I_IRQ_FETCH :
                                 (~cpuoff    &
~cpu_halt_cmd) ? I_DEC       : I_IDLE;
      I_IRQ_FETCH: i_state_nxt =  I_IRQ_DONE;
      I_IRQ_DONE : i_state_nxt =  I_DEC;
      I_DEC      : i_state_nxt =  irq_detect
? I_IRQ_FETCH :
                           (cpuoff |
cpu_halt_cmd) & exec_done ? I_IDLE      :
                             cpu_halt_cmd &
(e_state==E_IDLE)  ? I_IDLE      :
                              pc_sw_wr
? I_DEC       :
```

```
                              ~exec_done &
~(e_state==E_IDLE)  ? I_DEC       :

// Wait in decode state
                                  (inst_sz_nxt!
=2'b00)        ? I_EXT1      : I_DEC;

// until execution is completed
      I_EXT1     : i_state_nxt =  pc_sw_wr
? I_DEC       :
                                  (inst_sz!
=2'b01)            ? I_EXT2      : I_DEC;
      I_EXT2     : i_state_nxt =  I_DEC;
    // pragma coverage off
      default    : i_state_nxt =  I_IRQ_FETCH;
    // pragma coverage on
    endcase
```

While the actual encoding is abstracted from this code block (it is fully encoded, not one-hot), the existence of the 'default' case clearly defines what is supposed to happen when the state is not valid. If it were one-hot encoded, this line would immediately jump to I_IRQ_FETCH state if it ended up on a too hot two-hot situation.

It is also quickly apparent that this always@ block is not directly clocked. The sensitivity list contains 9 different signals which we might assume are synchronously clocked, and this block would be triggered whenever one or more of the signals changes. This is significant because if any one signal can be asynchronously toggled, the entire state machine will increment forward one step - perhaps out of sync with the backend and rest of the architecture.

Following back the cpu_halt_cmd, we find:

```
// CPU on/off through the debug
interface or cpu_en port
wire   cpu_halt_cmd = dbg_halt_cmd |
~cpu_en_s;
```

This is a pretty common practice. Several signals might be combined with logic into a more descriptive wire name. The disadvantage of this is that it can hide from simple inspection external factors that might allow control of a signal. For example, if there were a secure lock/unlock signal that was combined with a user-controllable input, a coder might not use the correct signal and accidentally give a user-controllable signal influence over what should be a secured resource.

As we continue to step backwards we find:

```
always @(posedge dbg_clk or posedge dbg_rst)
  if (dbg_rst)            halt_flag <= 1'b0;
  else if (halt_flag_clr) halt_flag <= 1'b0;
  else if (halt_flag_set) halt_flag <= 1'b1;

wire dbg_halt_cmd = (halt_flag |
halt_flag_set) & ~inc_step[1];
```

dbg_halt_cmd depends on halt_flag, which normally is synchronous to dbg_clk, but the preceding always@ block is also sensitive to dbg_rst. We have found an external signal that can be used to trigger the frontend state machine logic asynchronously.

Looking back at the state machine, we can pick out the state that might be vulnerable to this, particularly IRQ_FETCH, IRQ_DONE, and DEC. When timed carefully, we should be able to skip any one of these states. This means we could theoretically bypass the execution of a single instruction since we failed to fetch it, and we would likely instead execute stale data, most likely re-executing the previous instruction.

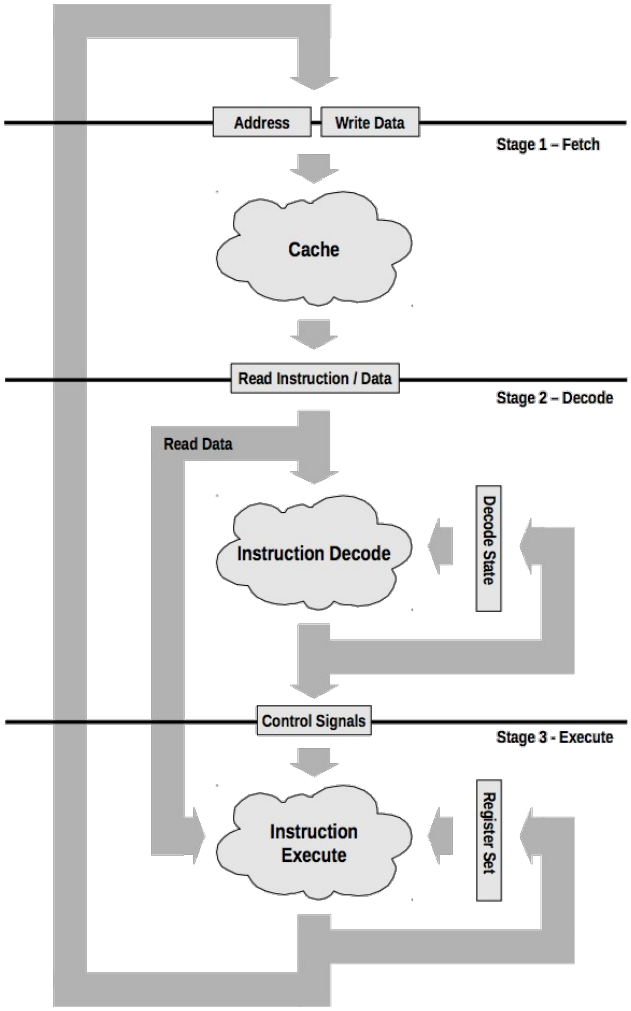
The takeaway here is to be mindful and deliberate about state transitions:

- Is your state machine strictly clocked?
- Are your sensitivity list parameters all synchronous?
- Are your critical signals 'pure' or are the combined logically with other, perhaps asynchronous or user-controlled signals?

Amber

Amber is an open-source ARMv2 compatible core. By implementing an older version of the ARM instruction set, the project avoids patent issues with later instruction sets, however ARMv2 is not supported by recent Linux kernels.

Amber sits in between openMSP430 and OpenRISC in complexity and capacity. While it's a much more complicated and more capable architecture than MSP430, it is architected with all memory access routed over the Wishbone interface versus a native memory interfaces in ORPSoC implementations.



Memory, Addressing, and Caching

Our analysis of Amber focused on memory, addressing, and caching - an area that often seems simple at first glance, but is filled with nuances, conditions, exceptions, and corner cases that can be difficult to hit but have a big impact when they are. After our analysis, we actually didn't encounter any security issues that seemed potentially vulnerable. As a result, let's highlight some of the things that were done right and how they might have gone wrong.

First, let's examine the memory map, as defined in system/memory_configuration.v:

```
// e.g. 24 for 32MBytes, 26 for 128MBytes
localparam MAIN_MSB             = 26;

// e.g. 13 for 4k words
localparam BOOT_MSB             = 13;

localparam MAIN_BASE            =
32'h0000_0000; /*  Main Memory            */
localparam BOOT_BASE            =
32'h0000_0000; /*  Cachable Boot Memory   */
localparam AMBER_TM_BASE        = 16'h1300;
/*  Timers Module          */
localparam AMBER_IC_BASE        = 16'h1400;
/*  Interrupt Controller   */
localparam AMBER_UART0_BASE     = 16'h1600;
/*  UART 0                 */
localparam AMBER_UART1_BASE     = 16'h1700;
/*  UART 1                 */
localparam ETHMAC_BASE          = 16'h2000;
/*  Ethernet MAC           */
localparam HIBOOT_BASE          =
32'h2800_0000; /*  Uncachable Boot Memory */
localparam TEST_BASE            = 16'hf000;
/*  Test Module            */
…

// UART 0 address space
function in_uart0;
    input [31:0] address;
begin
    in_uart0 = address [31:16] ==
AMBER_UART0_BASE;
end
endfunction
```

Here, we see the wishbone addresses where each peripheral as well as memory are mapped, followed by an example of one of the "in_xxx" functions that returns true when a given address maps to that target peripheral. There are two common pitfalls that often happen right here.

First, overlapping memory spaces can introduce a number of problems. This is remediated by giving each device a full 64k of address space. As the code is formatted, it's plain to see the order in memory and the high 16 bits of each address range. Also thanks to the well-formated Verilog, it is quick to confirm that all of the "in_xxx" functions properly compare the same high 16 bits against the tested address.

Second, introducing configurable memory mapping adds orders of magnitude of complexity right here. Considering this is a soft core design intended for customization on an FPGA, there are nuances to customizing this memory map that are not immediately apparent. Placing I/O devices as low as 0x13000000 limits the system to just over 128MB of memory. As memory capacity has increased, most FPGA development boards come equipped with more memory. It is not clear from this code that adjusting MAIN_MSB beyond it's current value might be catastrophic.

Do determine if, in fact, there are any memory aliasing issues, we have to examine the logic that actually resolves addresses using these functions and defines. For that we examine system/wishbone_arbiter.v:

```
// Arbitrate between slaves
assign current_slave = in_ethmac
( master_adr ) ? 4'd0  :  // Ethmac
                          in_boot_mem
( master_adr ) ? 4'd1  :  // Boot memory
                          in_main_mem
( master_adr ) ? 4'd2  :  // Main memory
                          in_uart0
( master_adr ) ? 4'd3  :  // UART 0
                          in_uart1
( master_adr ) ? 4'd4  :  // UART 1
                          in_test
( master_adr ) ? 4'd5  :  // Test Module
                          in_tm
( master_adr ) ? 4'd6  :  // Timer Module
                          in_ic
( master_adr ) ? 4'd7  :  // Interrupt
Controller

4'd2  ;  // default to main memory
```

It is immediately clear that address resolution priority is well defined. In the event there were overlapping memory regions, the way this assignment is structured ensures that the same device would get priority over an address every time. Just as in software - the fewer different implementations of the same logic the better.

In the event the memory size were increased to 512MB, main memory would overtake the UARTs, Test module, Timer module, and interrupt controller. However, writes to 0x20000000 to 0x2000FFFF would still be directed to the Ethernet Controller creating a hole in main memory.

We can also see that any address not otherwise defined is mapped to main memory. This begs the question: What would happen if we accessed address 0x18000000? No I/O device is mapped in this region, so the access would be directed to main memory. in system/main_mem.v:

```
    //
-----------------------------------------------
--------
    // Write for 32-bit wishbone
    //
-----------------------------------------------
--------
    always @( posedge i_clk )
        begin
        wr_en          <= start_write;
        wr_mask        <= ~ i_wb_sel;
        wr_data        <= i_wb_dat;

                          // Wrap the address
at 32 MB, or full width
        addr_d1        <= i_mem_ctrl ? {5'd0,
i_wb_adr[24:2]} : i_wb_adr[29:2];

        if ( wr_en )
            ram [addr_d1[27:2]]  <=
masked_wdata;
        end
 …
    //
-----------------------------------------------
--------
    // Read for 32-bit wishbone
    //
-----------------------------------------------
--------
    assign rd_data = ram [addr_d1[27:2]];
 …
```

Note that this is not fully a fair assessment. main_mem.v is non synthesizable and for simulation only. However, it is a good indication of what is likely to be the behaviour of a reals system. High order bits of the address are ignored - memory simply wraps around past 128MB. This means that a write to 0x10000001 is identical to a write to 0x01. In a more complex system, it also means that a access to 0x10000001 might be able to circumvent any checks that would apply to an access to 0x01.

Since this is a very simple core with no MMU, there's no paging or memory virtualization to bypass here. Also, ARMv2's privilege levels don't confer any different access to memory, so there's no additional exploit here that wouldn't have been possible via a more direct approach.

Despite the fact that no further vulnerability was discovered, the pathways left open on this system are indicative of what could be found in more complicated systems, and could be used to bypass security measures. When evaluating memory, addressing, and caching, consider:

- What devices live in the memory map?
- How are they allocated? Ordered? Prioritized?
- If there are multiple implementations of address decoding, do they decode overlaps with the same priority?
- Do memory aliases exist? Can they be used to bypass protections?

Summary

In the course of reviewing multiple different open-source Verilog projects, a number of potential vulnerabilities were identified. The vulnerabilities highlighted are representative of security related issues commonly seen in hardware designs. To recap:

- Different state encoding schemes have different benefits. If possible, validate current states, and define the behavior for when an invalid state occurs. Depending on design, this could be done with separate combinational logic or could be as easy as more carefully defining default cases of case statements.
- Examine interesting pathways through state machines. The most direct route makes the most sense, but if alternate routes exist they should be validated.
- Every signal in a sensitivity list is granted a small degree of influence over a block of logic. Be sure that the correct signals are in the list, and if possible isolate separate blocks of logic to separate sensitivity lists with only the relevant signals.
- Combining signals may help code readability but also opens the door for unexpected inputs into a block of logic. Trace critical signals backwards to make sure their cone of logic does not include attacker-controllable signals.
- It is critical to iron out the details of a system's memory map. Mirroring, overlapping, and decode priority can all cause trouble, especially in systems that depend on decoding for access control.

Conclusion

Exploitable hardware security bugs do exist in production systems. There is very little black art involved in finding these bugs when the source code is available. Basic hardware security validation is well within the capabilities of most silicon developers and validators, so long as they take the time to understand the importance of a product's security requirements.

References

- Fuzzing the RTL: http://conference.hitb.org/hitbsecconf2010kul/materials/D1T2%20-%20Mary%20Yeoh%20-%20Fuzzing%20the%20RTL.pdf
- HSDL: A Security Development Lifecycle for hardware technologies: http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?arnumber=6224330
- A Survey of Frequently Identified Vulnerabilities in Commercial Computing Semiconductors: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5955008&navigation=1
- Threat analysis for hardware and software products using HazOP: http://dl.acm.org/citation.cfm?id=1569853
- Practical Secure Hardware Design for Embedded Systems: http://www.grandideastudio.com/wp-content/uploads/secure_embed_paper.pdf
- An efficient algorithm for identifying security relevant logic and vulnerabilities in RTL designs: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6581567