A Formal Approach to Security Verification

Proving Secrets Can't Leak into Insecure Channels on an IFC Processor

This outlines a verification approach for information security applied to a hypothetical processor design.

Processor Description

This hypothetical processor provides fine grained information flow control (IFC) by tagging all data elements with security labels. The processor consumes data on high and low input channels, performs computation on this data, and them produces results for high and low output channels for downstream consumption.

Once data enters the processor on the input channels, data words are given labels to designate their security level (high or low) and these labels propagate with the data during computation allowing the processor to track how information flows in a program. When the program outputs data to the output channels, the processor ensures no data labeled high is transmitted on the low output channel. All high data attempted to be written to the low output will send a constant zero instead.

For simplicity, instructions are streamed into the processor; the processor itself does not fetch its own instructions nor does it have an internal instruction or data memory.

The interface of the processor includes input and output channels for both high and low security and an input program in the form of an instruction stream:

```
                          +-------------------+
                          |                   |
High Input Channel ---->|                   |----> High Output Channel
                          |        CPU        |
Low Input  Channel ---->|                   |----> Low Output Channel
                          |                   |
                          +-------------------+
                                   ^
                                   |
                              Instructions
```

For simplicity, the processor has a small register file of 8 registers that includes three general purpose registers, a constant zero register, and four registers mapped to the input and output channels.

The processor's complete register file:

```
- ZERO        : Constant zero register.
- INPUT_HIGH  : High security input.
- INPUT_LOW   : Low security input.
- OUTPUT_HIGH : High security output.
- OUTPUT_LOW  : Low security output.
- REG_A       : General purpose register A.
```

```
  - REG_B       : General purpose register B.
  - REG_C       : General purpose register C.
```

Since the purpose is to model and verify information flow control properties, the actual computation capabilities of the processor is not that important, with the exception of label operations. Therefore the processor only supports one datatype: booleans, so each register is made up of two bits: one for the value and one for the label.

The processor provides the following RISC-like instructions:

```
- COPY <src> <dst>         :  dst = src;
- NOT <src> <dst>          :  dst = !src;
- AND <src1> <src2> <dst> :  dst = src1 && src2;
- OR <src1> <src2> <dst>  :  dst = src1 || src2;
- CLASSIFY <src> <dst>    :  Copy src to dst and label with high.
- LABEL_OF <src> <dst>    :  Get the label of src and provided it as a
value to dst, labeled with low.
```

Noninterference Property

The security property we are attempting to verify is a form of noninterference, namely anything that could happen on the high input channel should be unobservable at the low output channel. Or more formally:

```
Let H1 and H2 be streams of high input channel data.
Let L be a stream of low input channel data.
Let I be a stream of input instructions.
Let CPU_LOW(H, L, I) be a function that maps a high input stream,
  a low input stream, and an instruction stream to a low output stream.

Then

Forall H1, H2, L, I . CPU_LOW(H1, L, I) === CPU_LOW(H2, L, I)
```

A few assumptions to consider: First, side channels are out of scope. Second, only the input and output high channels are considered high security, the instruction stream is not. If parts of a program are secret, this would require labeling individual instructions along with substantial architectural changes that are beyond the scope of this example.

Modeling and Verification

The processor is modeled in SystemVerilog and the noninterference property is specified with a combination of SystemVerilog Assertions (SVA) and a "testbench" to help formulate the problem.

Model checking is an excellent technique for this type of verification. Because of the limited size of the modeled design (size of state space), proofs will terminate quickly with a pass, fail, or an inconclusive result. Furthermore, when we get a failure, the model checker will return a counter example to help with debugging (on a failure) or proof guidance (on an inconclusive).

The model checker we are using is SBY provided by YosysHQ. SBY parses and elaborates SystemVerilog using Verific (the frontend used by many commercial EDA tools), generates proof obligations, and submits these to several backend solvers. It supports a few types of model checking techniques; our example will use k-induction with SMT solvers. This limits us to safety properties (properties with a finite horizon), but SBY can also handle liveness properties via BDD solvers.

Specifying the universally quantified variables H1, H2, L, and I, is done by creating inputs in a top level SystemVerilog module. A clock is also needed to cycle from one state to the next:

```
module highLowCpuVerify (
    // Free variables for verification, i.e. H1, H2, L, and I.
    input logic clk,
    input logic high1_i,  // Different high input sequences.
    input logic high2_i,
    input logic low_i,    // Common low input sequence.
    input instr_t instr   // Common instruction stream.
);
```

The next "testbench" trick is to instantiate two copies of our processor. The inputs L and I are common to both, but H1 and H2 are independent and drive each of the two processor instances. This allows the proof search to find any differences between H1 and H2 that would be observable at the low outputs.

```
highLowCpu cpu1 (
    .clk(clk),
    .reset(reset),
    .high_i(high1_i),
    .low_i(low_i),
    .instr(instr),
    .high_o(),
    .low_o(low1_o)
);

highLowCpu cpu2 (
    .clk(clk),
    .reset(reset),
    .high_i(high2_i),
    .low_i(low_i),
    .instr(instr),
    .high_o(),
    .low_o(low2_o)
);
```

Finally, we create an assertion to capture our noninterference property:

```
// If not in reset, the low outputs from both processors should be the
same.
assert property (@(posedge clk) imply(!reset, low1_o == low2_o));
```

Verification Results

Running the k-induction model checker initially leads to an inconclusive result. Base case checking (looking forward from known initial conditions at time zero) didn't find any problems out to 20 twenty steps. However the inductive step found a way to initialize the registers that caused an error in one cycle. SBY provides counter examples in both a waveform (Verilog VCD) and as a Verilog testbench. Reading the generated testbench quickly gives us an indication of what is going on. Here are the initial register values in both processor instances that led to a failure:

```
UUT.cpu1.\reg_a[label]  = 1'b0;
UUT.cpu1.\reg_a[value]  = 1'b1;
UUT.cpu1.\reg_b[label]  = 1'b0;
UUT.cpu1.\reg_b[value]  = 1'b1;
UUT.cpu1.\reg_c[label]  = 1'b1;
UUT.cpu1.\reg_c[value]  = 1'b1;
UUT.cpu1.\reg_output_high[label]  = 1'b0;
UUT.cpu1.\reg_output_high[value]  = 1'b1;
UUT.cpu1.\reg_output_low[label]  = 1'b0;
UUT.cpu1.\reg_output_low[value]  = 1'b0;
UUT.cpu1.skip = 1'b0;
UUT.cpu2.\reg_a[label]  = 1'b0;
UUT.cpu2.\reg_a[value]  = 1'b1;
UUT.cpu2.\reg_b[label]  = 1'b0;
UUT.cpu2.\reg_b[value]  = 1'b0;
UUT.cpu2.\reg_c[label]  = 1'b0;
UUT.cpu2.\reg_c[value]  = 1'b1;
UUT.cpu2.\reg_output_high[label]  = 1'b0;
UUT.cpu2.\reg_output_high[value]  = 1'b1;
UUT.cpu2.\reg_output_low[label]  = 1'b0;
UUT.cpu2.\reg_output_low[value]  = 1'b0;
UUT.cpu2.skip = 1'b0;
UUT.reset = 1'b0;
```

The key is to compare REG_B in the two instances. Both labels are low, but the value in cpu1 is true while the value in cpu2 is false. The processor program would merely need to copy REG_B to OUTPUT_LOW for the difference to be observable at the output.

```
UUT.cpu1.\reg_b[label]  = 1'b0;
UUT.cpu1.\reg_b[value]  = 1'b1;
UUT.cpu2.\reg_b[label]  = 1'b0;
UUT.cpu2.\reg_b[value]  = 1'b0;
```

Sometimes it is possible to increase the depth of the search (increase K) to overcome inconclusive results. But not here. The program could just sit idle for however long it needed to outlive the search depth of K, not modify REG_B, then send REG_B to the low output channel. In this case the proof needs help. The question to ask is if this state is even reachable: Is it possible for a register to have two different values across the two processor instances while either is labeled low? No, this should not happen and therefore we can write a lemma to prove it; not just for REG_B, but all registers. In SystemVerilog we do this with a combination of a helper function and an assertion:

```
// Compare corresponding registers between the two CPUs.
// If either is labeled low, the values and the labels must be equal.
function automatic logic if_low_then_equal(value_label_t r1, value_label_t
r2);
    return imply(!r1.label || !r2.label, r1 == r2);
endfunction

// Lemma: All mutable registers that are labeled low must be equal between
the two CPUs.
assert property (@(posedge clk)
  imply (!reset,
    if_low_then_equal(cpu1.reg_a, cpu2.reg_a) &&
    if_low_then_equal(cpu1.reg_b, cpu2.reg_b) &&
    if_low_then_equal(cpu1.reg_c, cpu2.reg_c) &&
    if_low_then_equal(cpu1.reg_output_high, cpu2.reg_output_high) &&
    if_low_then_equal(cpu1.reg_output_low, cpu2.reg_output_low)
  ));
```

With this new lemma the proof passes; the model checking run is nearly instantaneous.

Architectural Exploration

Now with a provably secure processor model, it becomes an evaluation platform to try out architectural extensions and new instructions. One extension to explore are ways to manipulate the control flow of a program, namely conditional jumps. What if we add a PIC style skip instruction?

```
- SKIP_NEXT <src>         :  If src is true, skip the next instruction.
```

We implement this by adding a "skip" flag, which when set blocks any register update in the next cycle and then clears itself.

```
// Skip logic for SKIP_NEXT instruction.
logic skip;
```

```
// Update registers.
always_ff @(posedge clk) begin
  if (reset) begin
    reg_output_high <= reg_zero;
    reg_output_low  <= reg_zero;
    reg_a           <= reg_zero;
    reg_b           <= reg_zero;
    reg_c           <= reg_zero;
  end else begin
    if (!skip && instr_writes_to_reg(instr)) begin   // skip can block
register updates.
      case (instr.dst)
        OUTPUT_HIGH: reg_output_high <= result;
        OUTPUT_LOW:  reg_output_low  <= result;
        REG_A:       reg_a           <= result;
        REG_B:       reg_b           <= result;
        REG_C:       reg_c           <= result;
      endcase
    end
  end
end

// Update skip flag.
always_ff @(posedge clk) begin
  if (reset) begin
    skip <= 1'b0;
  end else begin
    if (!skip && instr.opcode == SKIP_NEXT) begin
      skip <= src1.value; // Skip next if src1 is true.
    end else begin
      skip <= 1'b0;
    end
  end
end
```

With SKIP_NEXT implemented, the verification quickly yields a base case failure: a definite bug. The produced counter example shows the following instruction sequence:

```
SKIP_NEXT INPUT_HIGH
NOT ZERO OUTPUT_LOW
COPY ZERO OUTPUT_LOW
```

The program checks the input high channel and if it is set, outputs a true value on the output low channel. This is a classic example of an implicit flow and is equivalent to:

```
if (secret)
  output_public(true);
output_public(false);
```

A clear security bug caught by model checking.

Automation Potential

LLMs have demonstrated the ability to reason surprising well about general programming. What is less known is how well they can handle state machines and hardware designs, for two reasons:

1. There is less available SystemVerilog source code for training.
2. The semantics of SystemVerilog (state machines, assertions) is significantly different than common programming languages.

If these challenges can be overcome, if they in fact exist at all, there is much potential for using LLMs to aid with model checking based verification. First, as very often in these problems, simple bugs are the first to appear in early counter examples of proof failures. A iterative loop between the model checker and an LLM could quickly fix a lot of the tedious problems up front.

Then once both the design and specification start to mature, the next problems typically to emerge are inconclusive results. Often K-induction is not powerful enough to prove high level properties without help. Again, possibly an iteration loop between the model checker and an LLM could be used generate proof lemmas.

Finally, there is the distinct possibility that LLMs in concert with a model checker could generate correct designs purely from specifications.

We explored these last two possibilities with the processor example and another example: an arbiter design.

To test the possibility for lemma generation on the processor example, we removed the low-registers-are-equal lemmas, reran the verification, then gave the design and the counter example to an LLM (Grok). The LLM was immediately successful at producing the necessary lemma in the first iteration.
But on further inspection it was noticed that the 'if-low-then-equal' helper functions and comments were left in the code, giving the LLM enough clues to deduce a solution. When removing the function and surrounding comments the LLM failed to even approach solution with no improvement after 5 iterations.

The next test was to try an LLM at design synthesis using a specification for a resource arbiter. The arbiter included a few semi complicated requirements including granting multiple resources in the same cycle and limiting how long a device should wait for a resource.

Implementing the design correctly by hand was not straightforward. Even with the help from the model checker, coding a correct design took about an hour, with much trial and error. The big surprise was the LLM was able to find a better solution after only two iterations. The whole process took 30 seconds.

The handwritten solution. Short, but hard to follow. The approach was to always grant device A if requested then grant either device B or C based on requests and if device C was blocked in the last cycle Kludgy and asymmetric:

```
logic deniedC;
initial
    deniedC = 1'b0;
always_ff @(posedge clk)
    deniedC <= requestC && !grantC;

assign grantA = requestA;
assign grantB = requestB && !(requestA && deniedC && requestC);
assign grantC = requestC && !(grantA && grantB);
```

The LLM's solution. It's longer, but a cleaner approach and easier to comprehend. It always grants requests unless all three devices request at the same time at which point it decided based on a priority order of which devices where blocked in the previous cycle:

```
logic was_blocked_A;
logic was_blocked_B;
logic was_blocked_C;

// Explicit reset initialization (required for some formal tools and clean
simulation)
initial begin
    was_blocked_A = 1'b0;
    was_blocked_B = 1'b0;
    was_blocked_C = 1'b0;
end

always_ff @(posedge clk) begin
    was_blocked_A <= requestA && !grantA;
    was_blocked_B <= requestB && !grantB;
    was_blocked_C <= requestC && !grantC;
end

always_comb begin
    // Default: grant every active request
    grantA = requestA;
    grantB = requestB;
    grantC = requestC;

    // Three concurrent requests → grant only two
    if (requestA && requestB && requestC) begin
        if (was_blocked_A) begin
            // A was blocked last cycle → grant A+B, block C
            grantA = 1'b1;
            grantB = 1'b1;
            grantC = 1'b0;
        end
        else if (was_blocked_B) begin
            // B was blocked last cycle → grant A+B, block C
```

```
                grantA = 1'b1;
                grantB = 1'b1;
                grantC = 1'b0;
            end
            else if (was_blocked_C) begin
                // C was blocked last cycle → grant A+C, block B
                grantA = 1'b1;
                grantB = 1'b0;
                grantC = 1'b1;
            end
            else begin
                // First time all three are active → fixed priority A > B > C
                grantA = 1'b1;
                grantB = 1'b1;
                grantC = 1'b0;
            end
        end
    end
```

LLMs are full of surprises!

Acknowledgments

A big thanks to YosysHQ for providing SBY licensing and support!