Formal Method on Hardware Security

September 2022

1 Toy Example: Compare the Property in Rosette and in Coq

1.1 Property in Rosette.

Property to verify. Last week, we have a state machine and want to verify a property hold for each state of the machine. Formally,

$$\begin{aligned} \forall \ & \text{input} \ \ in_0, in_1, \ldots, in_n, \\ \text{Let} \ & \ S_0 \xrightarrow{in_0} S_1 \xrightarrow{in_1} \cdots \xrightarrow{in_n} S_{n+1} \\ \text{If} \ & \ \text{No smash in the } in_0, in_1, \ldots, in_n \\ \text{We have} \ & \ PropertyHold(S_0), PropertyHold(S_1), \ldots, PropertyHold(S_n) \end{aligned}$$

Property for induction step. The property I verify for the induction step is:

$$\begin{aligned} \forall \ & \text{state } S_i, \text{ input } in_i, \ in_{i+1}, \\ \text{Let} \ & \ S_i \xrightarrow{in_i} S_{i+1} \xrightarrow{in_{i+1}} S_{i+2} \\ \text{If} \ & \ \text{No smash in the } in_i, in_{i+1} \\ \text{We have} \ & \ PropertyHold(S_{i+1}) \Rightarrow PropertyHold(S_{i+2}) \end{aligned}$$

It is a little weird since why not just use:

$$\begin{aligned} \forall \ & \text{state } S_i, \text{ input } in_i, \\ \text{Let} \ & \ S_i \xrightarrow{in_i} S_{i+1} \\ \text{If} \ & \ \text{No smash in the } in_i \\ \text{We have} \ & \ PropertyHold(S_i) \Rightarrow PropertyHold(S_{i+1}) \end{aligned}$$

The answer is that, yes, this bottom version definitely looks more natural for our toy example. However, the above version used by mistake because I was looking at the property from another perspective. I want to explain this other perspective because firstly, that is how I will define it in coq and in more other scenarios for convenience. And secondly this will help you understand that induction itself is more powerful than you might think.

1.2 Property in Coq.

Property to verify. Let's translate the property in following way:

$$\begin{aligned} \forall \ & \text{input} \ \ in_0, in_1, \ldots, in_n, \\ \text{We have} \ & \ PropertyHold'(in_0, in_1, \ldots, in_n) \end{aligned}$$

Here, stead of considering the property is a function on the state, we consider it as a function on the whole input sequence. And it says that with this input sequence, if there's no smash in it, then we run it on the state machine we defined from initial state, it should not violate property we defined above.

Property for induction step. Now, let's consider how to use the induction on this property. It is actually:

$$\begin{aligned} \forall \ & \text{input} \ \ in_0, in_1, \ldots, in_i, in_{i+1}, \\ \text{We have} \ \ & PropertyHold'(in_0, in_1, \ldots, in_i) \Rightarrow PropertyHold'(in_0, in_1, \ldots, in_i, in_{i+1}) \end{aligned}$$

Note that this is exactly the version that we will prove in coq.

Relate the coq property with Rosette property. However how does this connect to our older induction step in rosette? We can try to expand this property a little bit.

$$\begin{aligned} \forall \ & \text{input} \ \ in_0, in_1, \ldots, in_i, in_{i+1}, \\ \text{Let} \ \ & S_0 \xrightarrow{in_0} S_1 \xrightarrow{in_1} \cdots \xrightarrow{in_n} S_i \xrightarrow{in_{i+1}} S_{i+2} \xrightarrow{in_{i+1}} S_{i+2} \\ \text{We have} \ \ & PropertyHold(S_1) \wedge \cdots \wedge PropertyHold(S_{i+1}) \Rightarrow PropertyHold(S_{i+2}) \end{aligned}$$

To further become identical to the Rosette property we are verifying, we actually conservatively think the S_n here can be an arbitrary state.

2 Verification on Real Example

2.1 Safety Property and Liveness Properties

There are many types of properties in real world. For example, safety property and liveness property are very popular. Safety means bad things not happen, for example, we say our "not reaching broken state" is actually a security property.

However, only security property cannot model our intuitive expectation on real world system very well. Specifically, our traffic light can just have a bug and stucks at green state. Yes, "bad things do not happen", it is not broken, but it is also not doing good thing. So the liveness property is saying good things will happen. Of course, our toy example is too simple that we can basically say it becomes which state at which cycle. However, in real complex example, we are not sure how many cycles or steps a good event will happen, but we want to say expected behaviour will eventually happen.

2.2 CPU Models

OK, I think everything we have been talking about is actually general to any verification. But now, let's bring our focus on hardware security. The first thing is how to model the CPU. Let's start with models we are familiar with, then go to abstract models whose existence is to assist the verification.

RTL. Firstly, RTL is definitely the first example. It defines what is the state of the system and what is the logic to trigger state transition. For example, the RTL of a pipeline CPU will include architecture states like register file, memory and PC, and also micro-architecture states like pipeline buffer, cache. And you write combinational logic to define how to trigger state updates. You can imagine when we want to verify such large state machine in Rosette or Coq. We probably just write down the state and update functions just like our toy example.

ISA Specification. A second CPU model we are familiar is ISA specification. It executes the instructions sequencially, it does not have cycle, and you probably want to use it to verify the functionality of your RTL, which means you simulate the RTL and ISA specification together, and each time the RTL commit an instruction, you check the correctness.

This is our intuition about the specification, however, when talking about formal verification, I generally understand it from another perspective to get the logic clearer. I would say ISA specification represents a set of CPU implementations. They may vary at how long to commit an instruction, but everytime they commit an instruction, they should update the architecture states in a same way. And since ISA can capture a set of CPUs or RTL implementations, and RTL is just one point, we call ISA is an abstraction to the RTL.

2.3 Refinement Property

Actually, only these two models can already allow us to define many interesting properties to verify. You probably are very familiar with these properties I will describe, but I still want us to re-think about them again, especially thinking about the fact that ISA specification is an abstraction and it represents a set of CPUs.

RTL Implementation refines ISA Specification. OK, refinement, it is just saying whether or not, one CPU implementation, or a small set of them, belongs to a large set of CPU implementations. It is just what we call "a CPU implementation is secure" in a fancy way, but the mind set is that ISA specification is a larget set of designs and is an abstraction. Honestly, using CPU implementation and specification to explain refinement is kind of abuse because everyone knows what "a CPU implementation is secure" means, why bother proposing a new word called refinement. However, I just want this example to be a simple ground truth and later in case you get confused about what refinement means, you can always refer to this example.

Any-size fifo refines infinite-size fifo in bluespec. A second example, which is more interesting is to think about refinement in smaller state machines. I just use the example Thomas always uses, a fifo. A fifo can enqueue a data, the data will be stored into a buffer, and later the data can be dequeued. Regarding to the size of this fifo, an insteresting claim is that a one-element fifo refines an two-element fifo.

OK, you must be confused about this statement because I have just define refinement as an implementation point belongs to a set of implementations. And these two fifos are both implementations points. The answer to this is that the definition of refinement does to specify how you define the element and the set. In CPU implementation and specification example, the element is RTL and set is a set of RTL. Here, the element is a valid execution trace of the fifo and set is a set of them. Think about what are the valid execution trace of one-element fifo. It can only be enqueue, dequeue, enqueue, dequeue, right? And think about what are the valid trace of two-element fifo. It can be enqueue, dequeue, but it can also be enqueue, enqueue, dequeue, dequeue, or others.

Let's first summarize a better definition of refinement and then understand what "one-element fifo refines an two-element fifo" means. Refinement is regarding to two objects, from a given perspective, one is smaller set than the other. In our CPU example, the perspective is when to commit instructions. And in fifo example, it is the execution trace. Then the question is how to choose this perspective. I guess it is mainly to assist your proof and more concretely, it is about how your object interact with others.

Benefits of refinement. I have spent some efforts to use word refinement to explain a few facts. Now, let me explain how the refinement aspect can help us. Let me first give two examples.

Firstly, in general, we can verifiy the RTL implementation refine the ISA. Why? Because people are designing software that can work properly on ISA, meaning a set of implementations, and you say our CPU is one of them. This is just a re-think our old idea of abstraction layers.

And secondly, we can use this abstraction layers any places in our verification. Think about our fifo example. We eventually want to verify a CPU works with a 10-element fifo. However, we can first verify it works with a infinite fifo. Then, by using the refinement, we can say it also work with the 10-element fifo.

So actually, these two examples are the same, which is creating abstraction layers. But I guess everyone is familiar with the first example of ISA, but this is actaully helpful to use at any places.

Then, what's the benefits? It can simply make the verification at higher level easier, and adding another refinement property to verify. Think about verify a program is correct on ISA or micro-architecture. Clearly ISA is easy. And the hardware designers take care of the refinement property.

Pensieve. Currently, I am trying to verify an OoO CPU is not vulnerable to spectre. And I am actually verifying an OoO CPU with many abstract module.

2.4 More CPU Models on Different Abstraction Level

We have mentioned the RTL and ISA level of CPU models. After you understand the refinement and abstract, it is clear that there exist a lots of other useful models in the middle of them. For example, an RTL may specify a implementation with a 10-element fifo. However, we can have a little more abstract model that replace it with an infinite fifo.

2.5 More CPU Models with Different Description Language.

2.6 Single Trace Property and Hyperproperty.

Another interesting set of properties I will explain is called hyperproperty. I think these two properties are just too important that we cannot avoid. But actually, another category of peroperties is more related to the working I am doing recently. It is single trace property and hyperproperties.