Marquis Hackett

Micro-Policy Monitors with Tags, Lecture Notes

Can we build it? Yes, we can.

This paper focuses on the problem of creating a monitor for a given safety property that will run in “the hardware.” Running a monitor is a way to make sure some system upholds some security property. However, running it in the software can be computationally expensive. We regain performance by designing a special machine that runs this monitor in hardware. The researchers come up with a “backwards-refinement” strategy for creating a monitor in the hardware. Construct an “abstract machine” that specifies the property. Do a “backwards-refinement” by constructing a “symbolic machine” that uses tags, and also encapsulates the property specified in the abstract machine. They describe the process for several safety properties. (Memory Safety, Control Flow Integrity, Compartmentalization, Sealing).

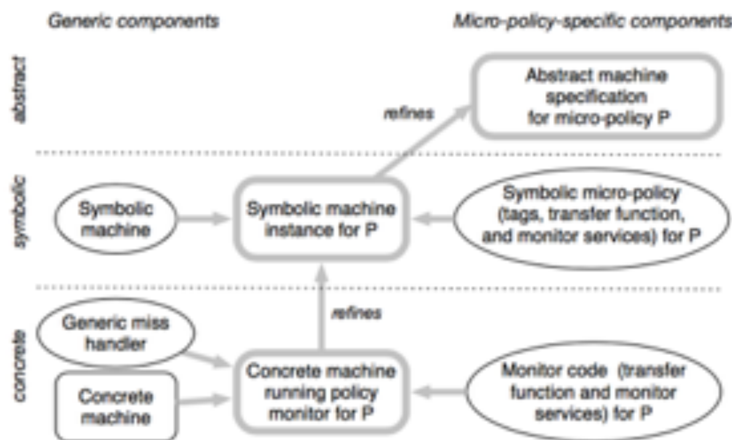

Figure 1. System overview

The symbolic machine can then be translated into a concrete machine monitor that can be executed on real hardware (i.e. hardware that is touchable, not imaginary). They describe a machine with a simplified instruction set based on RISC and architecture based on “PUMP.”

What is a monitor?
So, we execute our program, and want to make sure some security property is upheld. We accomplish this by executing a “monitor” alongside our program. The monitor approximates the execution of the program, and alerts us if the property is violated. In this case, the monitor is realized with a transition function; if the program tries to take a transition that is not defined in the transition function (that is, if the program tries to enter a state from a state when this state transition is not approved), then when halt and catch fire, etc.

The Basic (Abstract) Machine

$$\frac{mem[pc] = i \quad decode\ i = \mathsf{Binop}_{\oplus}\ r_1\ r_2\ r_d \quad reg[r_1]{=}w_1 \quad reg[r_2]{=}w_2 \quad reg'{=}reg[r_d{\leftarrow}w_1 \oplus w_2]}{(mem, reg, pc) \rightarrow (mem, reg', pc{+}1)} \ \text{(BINOP)}$$

$$\frac{reg[r_{arg1}] = \{w\}_k \quad reg[r_{arg2}] = k \quad reg' = reg[r_{ret}{\leftarrow}w] \quad reg[r_a] = pc'}{(mem, reg, \mathsf{unseal_addr}, ks) \rightarrow (mem, reg', pc', ks)} \ \text{(UNSEAL)}$$

This is a rule from the abstract machine. More on this late, but for now, just notice that it has no “tags.” In order to customize an abstract machine for your specific micro-policy, you add another rule like this. For example, look at the step rule for “seal” from the abstract sealing machine. This adds a rule for the things that go on when something is being sealed.

Marquis Hackett

Symbolic Machine

Symbolic states are represented by a tuple containing (mem, reg, pc, extra). These are symbolic atoms represented by w@t, which means “The thing that is w can be found at memory address t”.

$$\frac{\begin{array}{c} mem[w_{pc}] = i@t_i \quad decode\ i = \mathsf{Binop}_{\oplus}\ r_1\ r_2\ r_d \\ reg[r_1]{=}w_1@t_1 \quad reg[r_2]{=}w_2@t_2 \quad reg[r_d]{=}_@t_d \\ transfer(\mathsf{Binop}_{\oplus}, t_{pc}, t_i, t_1, t_2, t_d) = (t'_{pc}, t'_d) \\ reg' = reg[r_d{\leftarrow}(w_1 \oplus w_2)@t'_d] \end{array}}{(mem, reg, w_{pc}@t_{pc}, extra) \rightarrow (mem, reg', (w_{pc}{+}1)@t'_{pc}, extra)} \quad (\textsc{Binop})$$

This (left) is an example of one of the symbolic rules that correspond to an instruction in our reduced instruction set. Loosely speaking, it means “If all of the stuff above the line is true/happening/live, then take the transition/step below the line. All of the rules are defined similarly.

As a useful example of this whole transfer thing, consider taint tracking. Symbolic tags {tainted, untainted}. Also, _ is a “don’t care” character. With this transfer function, that’s how we get our policy violation halt and catch fire. Really, it just gets stuck. If *transfer(…)* doesn’t return anything, that means there was no thing defined for it in the transition function. That is, this wasn’t allowed by our property. then that “condition above the line” wont be true, and the program cannot step. So it’s stuck because we violated our property.

$$transfer(\mathsf{Binop}_{\oplus}, t_{pc}, -, t_1, t_2, -) = (t_{pc}, t_1 \vee t_2)$$

With the symbolic machine, we add this guy, the get_service step rule. We have these modular things (“services”) that encapsulate policy specific functionality. This step rule will kick the monitor over to the step rule defined by “Service” (e.g. unseal), and then service, when it’s done, will put the monitor back where it came from.

$$\frac{\begin{array}{c} get_service\ w_{pc} = (f, t_i) \\ transfer(\mathsf{Service}, t_{pc}, t_i, -, -, -) = (-, -) \\ f(mem, reg, w_{pc}@t_{pc}, extra) = (mem', reg', pc', extra') \end{array}}{(mem, reg, w_{pc}@t_{pc}, extra) \rightarrow (mem', reg', pc', extra')} \quad (\textsc{Svc})$$

The Concrete Machine

We are now at the point where our fancy idea must execute on mere mortal machines.

First, consider the tags. The tags must be represented using fixed length words. We define a function that maps these “concrete tags” to symbolic tags. A concrete tag is a valid user tag if it can be decoded into a symbolic tag.

Then, consider the lobster. The monitor must be able to protect itself from being tampered with as well. Since we don’t have any special memory protection, this translates to making sure that neither the special (not privileged, technically) instructions like “AddRule, PutRule, GetTag and JumpFpc” , nor any of the monitor code itself can be run by anyone but the monitor. We add special “Monitor” tags to these things, and the Miss Handler checks for these codes. This is also realized by the “Miss Handler,” which jumps to an address, or halts the machine.

Marquis Hackett

Qualms and Considerations
The technique is modular and easily replicable. “Easily,” in the sense that there are well defined steps to follow in order to go from property definition to concrete machine. However, this still involves a significant amount of proof-by-hand. Which (c.f. multiplying 10,000 operands) is prone to error. Therefore, even though this technique upholds the spirit of formal verification, I would prefer a technique that is more automated. (read: a method that requires less of the user (read: requires less of me)).

Another note: the real machine that is being used is very (adv.) simple. No hardware stack, No separate instruction memory, no call stack, no memory protection. That’s impressive.

Another note: I’ve liberally exchanged the terms micro-policy, policy, property, and safety property. I believe I’ve done so in a manner consistent with the true meaning of each of these terms, but intellectual humility dictates that I confirm my assumption. Also, why is it called a micro-policy?

Another note: Code on their github page. There is a missing “Makefile.coq” , but aside from that, It’s very satisfying to see an academic project that is maintained and will compile on command (presumably).