Exploiting and mitigating side channels

March 27, 2014

1 Summary

Today we discussed how to define exploit side channels, and programmatically define them in a general way characterized by the programming language's semantics. While many people believe side channels carry (in the worst case) relatively low information bandwidth, the first paper shows this is not true, especially for interactive applications (such as those which send keystrokes). The second paper shows how to model timing channels as executions through a program based on the numbers of instructions executed and rewrites the program to be timing insensitive.

2 Keyboards and Covert Channels

- Is it misleading to say you're messing with an input channel?
  - Really, you just want to modulate a channel, it doesn't matter whether it's an input channel or not.
- What's the difference between covert and side channel in this context?
  - Covert channels are active: you have to actively modulate timing delays to use this channel.
  - Side channel: some secret information is revealed through a means that is indirect (not through the API of the channel)
- Because the keyboard is a digital (not analog) channel, timing is your only option, in some sense. If the channel were analog, for example, a video encoding, you could probably perturb the distribution by some other means.

2.1 How does it work?

```
Hacked machine

| packet 1 |    | packet 2 |    | packet 3 |
            {w1}            {w2}
```

Where the w1, w2, etc..., are the interpacket delays, which are modulated to send the packet. Now you take the interpacket delays and change them so that they are such that:

```
delta mod w/2 =
  { 0 | if the bit we want to send is zero }
  { floor(w/2) | if the bit we want to send is one }
```

- If we're runnning two machines and observing the owned input, why don't they need to synchronized? (i.e., Machine A is owned, and machine B is observing the keystrokes coming in from A, why don't A and B need synchronized clocks?) Because you're just observing incremental deltas, but you still need your clocks to have relatively similar resolutions. You couldn't have (e.g.,) B running at a much slower rate than A, because you'd drop some packets and miss timing diferences. I think we could mess with the mod window to account for this?

3 PC Security Model

This paper defines PC timing security in a way such that you are always leaking the PC to the adversary. To define security properties over this logic you can use the normal techniques augmented with a primitive that outputs the current PC value after every command is reduced in the semantics, allowing the attacker to effectively watch the execution (rather than just the sequences of low writes, for example).

(I (Kris) want to note that in Clarkson's et al's formalization of hyper linear temporal formulas, you can actually define security properties in the same way as normal, but change the way you generate models to run through model checking by simply writing the current PC to each state, allowing us to switch between the PC and normal model of security.)

This paper considers multiple kinds of attacks:

- Timing
- Power
- Error channel
- Cache (don't handle)

Simple explanation for why you can't handle cache: depends on non local (or compositional) context of current execution that can't be cleanly defined for each independent instruction. E.g., if you execute the instruction:

```
a := *b
```

The cache timing will be different depending on if b is resident in the cache or not, at which level, etc...

So what do they consider in this paper?

- Instrument their semantics with a new thing to track the side channel (e.g., timing channel).
- Now you also get a transcript, a parametric addition to the side channel to hack on this extra information (number of steps, time between steps, etc..)

You can view this as lifting the previous semantics into a new semantics where you have the same transitions, but also have this new component (e.g., to your abstract machine state) that "counts" steps. In fact, I think this probably forms an adjunction in the categorical sense?

Note that this model assumes that the adversary can infer the control flow from the PC. We also assume that our abstraction is now a faithful representation of our machine.

We define PC security as noninterference even when the attacker can see the PC, which is quite elegantly formulated in a generic way by means of the abstraction I previously described.

Straight line code is PC secure:

- Instruction time needs to be independent of the data it's operating on.
- Strength reduction and cache violate this (which tell you something about previous execution)
- Potential research idea: proving compiler transformations preserve PC security?

Good thing about this model is that it includes a transcript to give information.
In this paper they built a profiling tool, transformation, and static analysis.

- Profiling tool allows you to hunt for inconsistencies and find places where your program is *not* PC secure.
- Transformation takes your code and rewrites it so that it will be PC secure: evaluate both sides and then make a conditional assignment based on the branch predicate.