Lecture 17: Hardware security

6.1600 - Fall 2022
C-4, Kalai, Zeldovich
MIT

Hardware Security

- Randomness failures
- Attacks w/o physical access
    * OS bugs & TrustZone
    * Cache attacks
    * Rowhammer
- Attacks w/ physical access
    * Probing attacks
    * Fault attacks
    * Supply-chain attacks

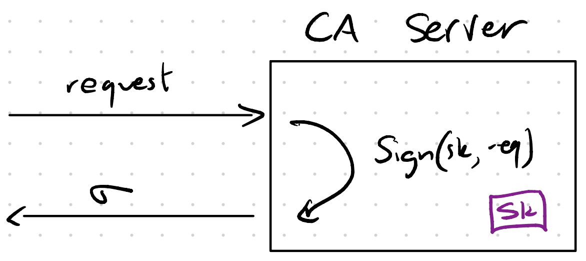
Logistics

* Lab 4 & Pset 4 out
  due 11/17
* HCG OH Tuesday
  11/15 1-2pm
  32-G970

Running example...

* You are running a CA that signs certs
  (Or: think of a cryptocurrency exchange)

* Business is premised on keeping secret key secret.
  ↳ Juicy target – small $ secret



Should enforce some policy...

only sign certs for *.mit.edu domains.

→ Can **prove** security of signature scheme under crypto assumption.

→ Can **verify** that crypto implementation faithfully implements sig algorithm... (on some ideal h/w)

... then you buy a computer, load the code onto it and run it on a machine with a bunch of other software

↳ What can go wrong?
↳ How to protect?

Theme:

In crypto & software security, we have clean/crisp characterizations of attacker's power

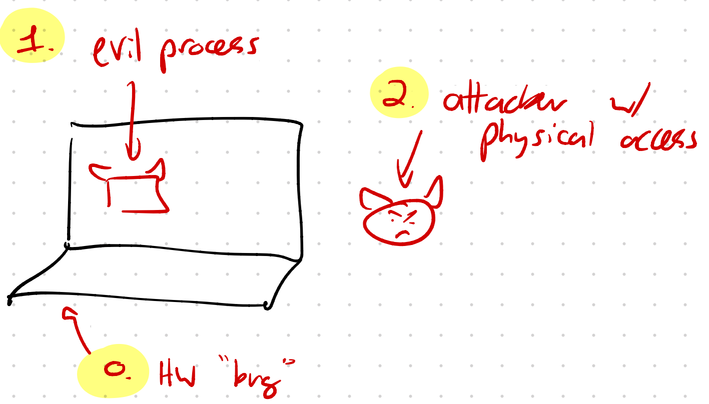
CRYPTO $\rightsquigarrow$ p.p.t

SOFTWARE $\longrightarrow$ Attacker chooses arbitrary inputs

In HW, it's often hard to precisely specify meaningful limits on the attacker's power.

$\Rightarrow$ Not often clean/satisfying solutions.

Three types of attack we'll discuss...



Randomness failure

* All of the crypto schemes we've discussed require randomness: secret keys, nonces (CPA), ...
* Where does a computer get random bits?
  - keypress timings
  - clock
  - temp sensor
  - ......

Very common failure: Embedded device has predictable randomness right after boot
↳ when it generates crypto keys
⇒ Attacker can easily guess secret key.

Idea: Build a special randomness-generating device into CPU
↳ RDRAND instruction

[Only helps if s/w developer uses it. Using time as randomness is common error.]
↳ Also bad PRG

Attacks

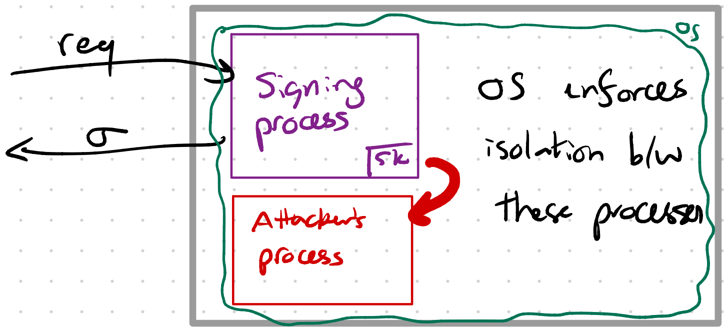
without

physical

access...

Hardware defenses against OS bugs

(Not really a hardware problem, but can defend w/ help from hardware.)



Problem:

Attacker can exploit OS bug to read memory of another process (even over network)

↳ OS is big - millions of LoC

↳ Much of OS doesn't need to touch crypto secrets

Hardware defenses against OS Bugs

Idea: Split OS into two parts
- Big ("insecure") part — no access to secrets
- Small ("secure") part — access to secrets

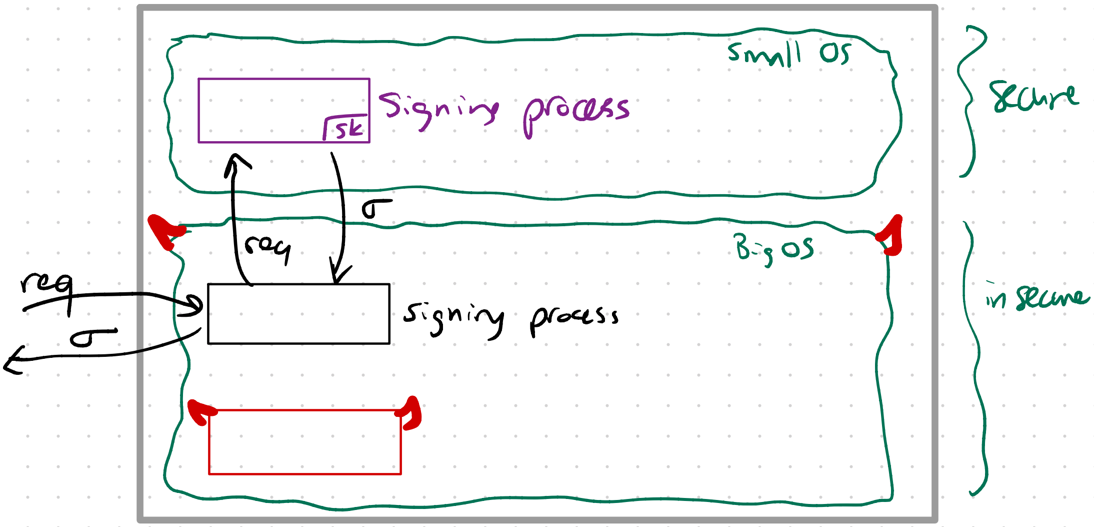
Hardware enforces isolation b/w parts ("TrustZone")

⇒ Assume that attacker compromises entire insec part

When machine boots, startup code marks memory as "secure" or "insecure".
- Insecure code can't read secure memory - segfault
- Insecure code can only call specific fns in secure code (e.g. sign)

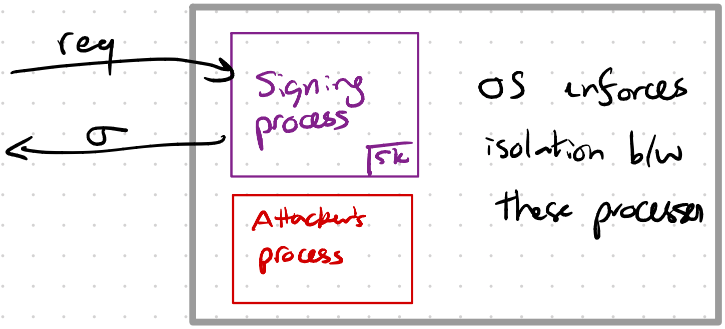
- Messy logic (parser, webserver, etc.) can live in insecure land
- Secrets & small piece of code using them lives in insecure land

TrustZone (simplified!)



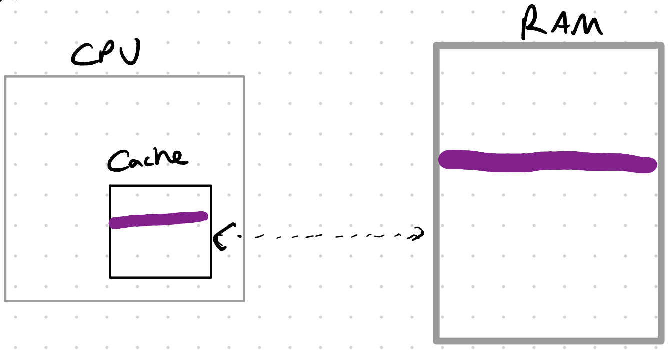
Cache Attacks (or: shared HW resources more generally)

Ex from Yarom & Benger 2014... simplified



Problem: * Attacker & victim process run on same CPU.
* Victim can leave traces of secrets in state of the CPU.

1. Victim runs, loads purple line or not
2. Attacker runs, loads purple line.
3. If access is fast, attacker knows that victim loaded purple.



↳ Attacker learns info about victim's access pattern.

Cache Attack: Example

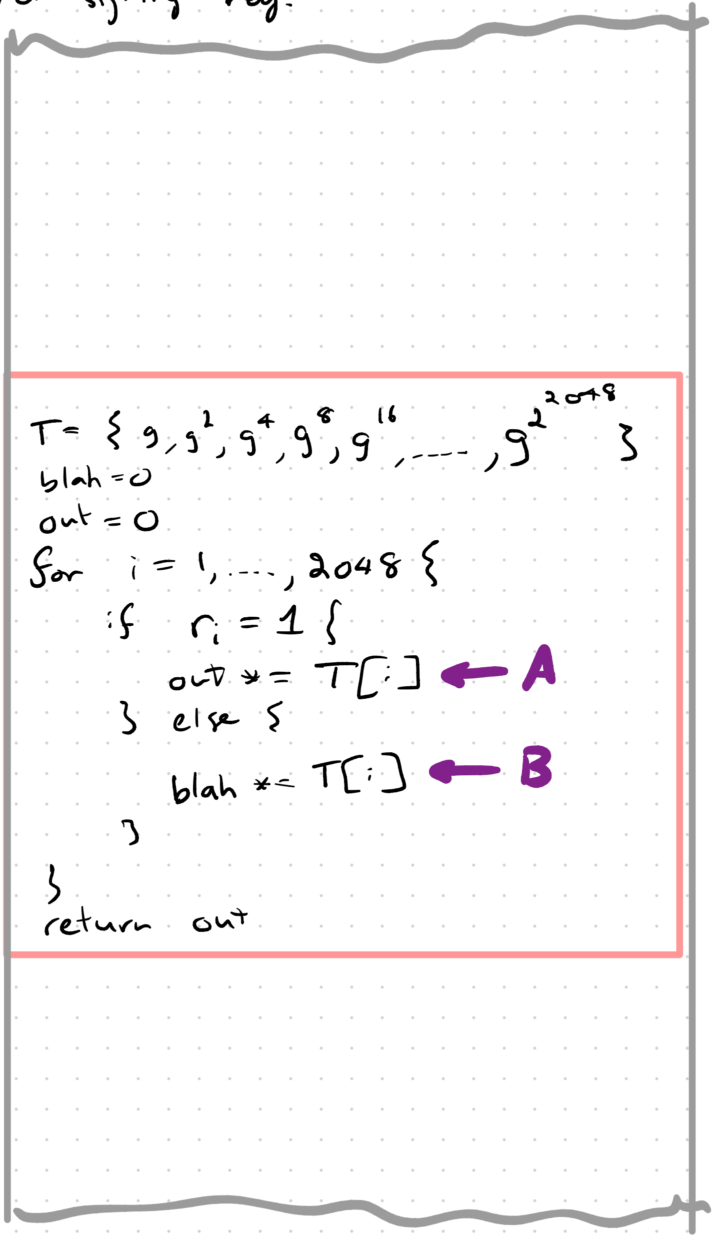
DSA signatures compute g^r mod p for secret $r \approx 2^{2018}$

↳ If attacker learns "r" it can recover the secret signing key. $(p, g \approx 2^{2048})$

If attacker & victim both use same crypto library, the OS will keep only one copy of library in phys RAM.

1. Victim runs
2. OS interrupts, runs attacker
3. Attacker runs, tries to access "A"
4. If fast, then $r_0 = 1$

... repeat to get all bits of r.



```
T = { $g, g^2, g^4, g^8, g^{16}, \ldots, g^{2^{2048}}$ }
blah = 0
out = 0
for i = 1, ..., 2048 {
    if $r_i = 1$ {
        out *= T[i]
    } else {
        blah *= T[i]
    }
}
return out
```

Cache attacks: Defenses?

Problem: * Hard to specify limits on leakage b/w processes via "microarchitectural side channels"

* CPU vendor keeps proc internals secret

Only complete answer: Use separate hardware for security-critical code.

No sharing of cache → No cache attacks

... still need to be careful about timing, etc.

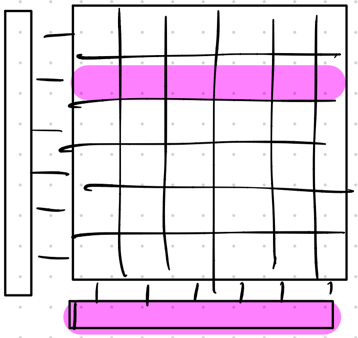
Examples

* Hardware security modules
* U2F token
* co-processors for crypto (next time)

Rowhammer

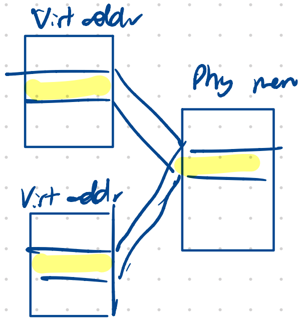
Surprise: By reading memory often, can induce bit flips in adjacent memory locations.

- Data in main mem (DRAM) stored in capacitors
  ↳ They drain over time, must be "refreshed" (64 ms)
- Reading chunk of mem drains capacitors
  ↳ must be rewritten
- Voltage fluctuations on one row cause neighboring rows to discharge more quickly



Attack: Read bytes in mem as fast as possible
$\Rightarrow$ Bit flips in nearby memory.

OS deduplicates memory pages
↳ If attacker process & victim process have chunk of identical data, OS stores them both in same phys RAM



$\Rightarrow$ Attacker can hammer arbitrary RAM locations!

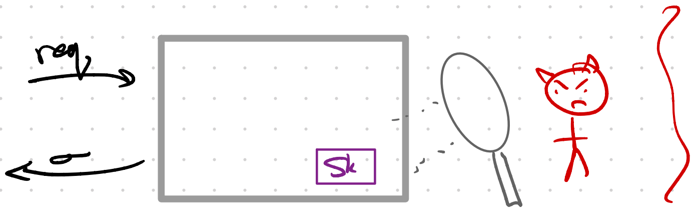
Mitigation: Refresh potential victim rows more often
↳ HW changes.

Now, on to:

Physical Attacks

Probing Attacks

Another threat: Attacker uses probe to read out values on wires in chip



Examples:

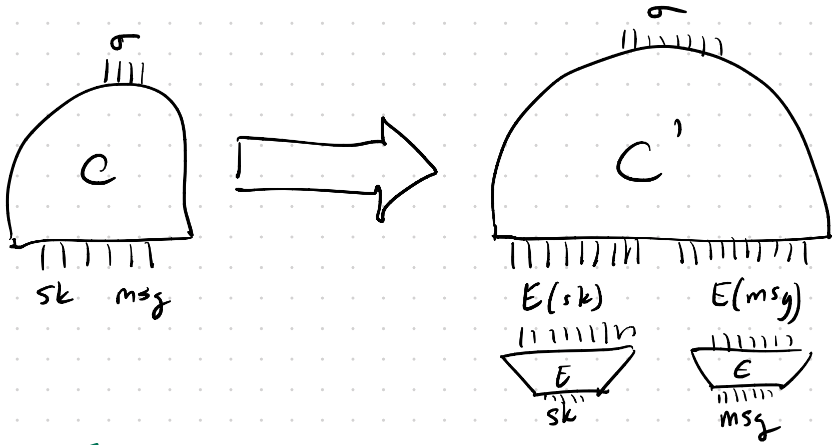
- power analysis — power use depends on secret
- probe on pins of chip
- optical emissions
- blinking lights on network router

Even 0.001 bits of leakage/sec is enough to leak a secret key in a few hrs.

Probing Attacks: Defense

Assume: Attacker can only probe values on t internal wires of signing circuit.
↳ Intuition: Probes are $$$

Then, there's a clever defense against probing attacks: "masking"

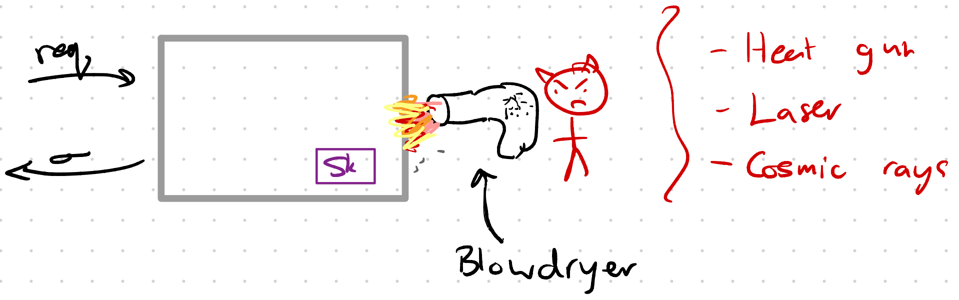


IDEA: Take a boolean ckt C implementing sig scheme.
Convert to ckt C' that implements sig scheme.
BUT — looking at any t internal wires of C' "leaks nothing" about sk.
(Technique: Again secure multiparty computation)

Still, only a very partial solution....
- input wires leaky
- what if attacker can get values on $t+1$ wires?

Fault Attacks

Another threat: Attacker induces "faults" (bit flips) in computation.



One bit flip can cause chaos:
- Leak secret signing key (RSA)
- Corrupt kernel data structures
  - ↳ Attacker can hijack machine

* If we assume that adv can't flip "too many" bits, can defend similar to probing attacks.
  ↳ Replicated HW used e.g. on satellites

* Pragmatic soln: verify signature before outputting it.
  ↳ Catches bit flips in signing step

Supply - Chain Attacks

- Attacker modifies the computer on its way to you
  * Snowden slides – Cisco router ... unlikely?
  * Hardware wallets on eBay ... likely?

- Modifications
  - mgmt interface
  - randomness
  - preloaded keys
  - extra comm

- How to defend? No great solns...
  * Buy in cash at random store? Doesn't really work for HW wallets
  * Inspect? Easy to hide... e.g. randomness, transistor
  * Build yourself? ☹
  * Trustworthy suppliers? DoD

↳ If the hardware is adversarial, you don't have a secure foundation

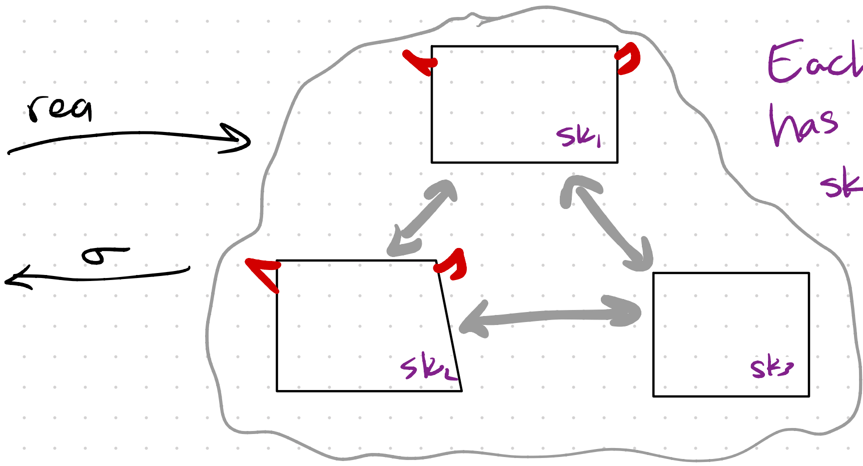
Supply-Chain Attacks

One meaningful defense: thresholding / "splitting trust"

Idea: Build system out of N computers, assume that attacker can compromise at most N-1 of them.

↳ Precise limit on attacker's power.

e.g. signing service w/ policy enforcement (1 BTC/day)



Each device has a "piece" of sk

$sk = sk_1 \oplus sk_2 \oplus sk_3$

- As long as attacker doesn't compromise all signing servers, can't learn sk. or violate policy.

  "Secure multiparty computation"

- Possible in theory to distribute any computation
- In practice, works only for simple comps