Security

To read more...

This day's papers:

Smith and Weingart, "'Building a high-performance, programmable secure coprocessor", 1998, Sections 1-6, 10

Supplementary reading:

Anderson, *Security Engineering*, Chapter 16.
`http://www.cl.cam.ac.uk/~rja14/book.html`
Costan and Devadas, *Intel SGX Explained*

hardware security categories

protection of software from software (page tables, kernel mode)

secondary topic of the paper for today

aid in producing vulnerability free code (bounds checking, no-execute bit)

protect code from people with access to hardware

primary topic of the paper for today

hardware security categories

protection of software from software (page tables, kernel mode)

secondary topic of the paper for today

aid in producing vulnerability free code (bounds checking, no-execute bit)

protect code from people with access to hardware

primary topic of the paper for today

major comments on the paper

use cases for secure coprocessors?

performance loss?

some secure coprocessor use cases

authentication tokens

certificate authorities

banking

usual goal: confidence private key isn't stolen

if device lost — plan to switch to new one

protection: dual-mode operation

kernel mode — operating systems runs with extra privileges

privileged instructions require kernel mode

kernel mode entered only using OS-controlled code

example privileged instructions:

- set page table
- disable interrupts
- configure I/O device

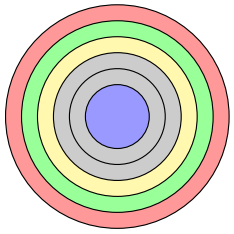
multiple protection levels

a lot of hardware supports multiple protection levels

lower level/outer ring — strictly more access

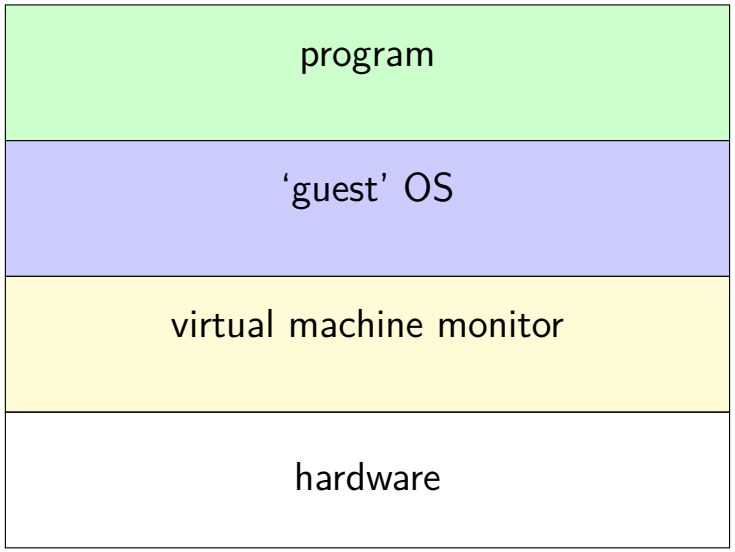
e.g. x86:

- system management mode ("ring −2")
- hypervisor mode ("ring −1")
- ring 0 ("kernel mode")
- ring 1
- ring 2
- ring 3 ("user mode")



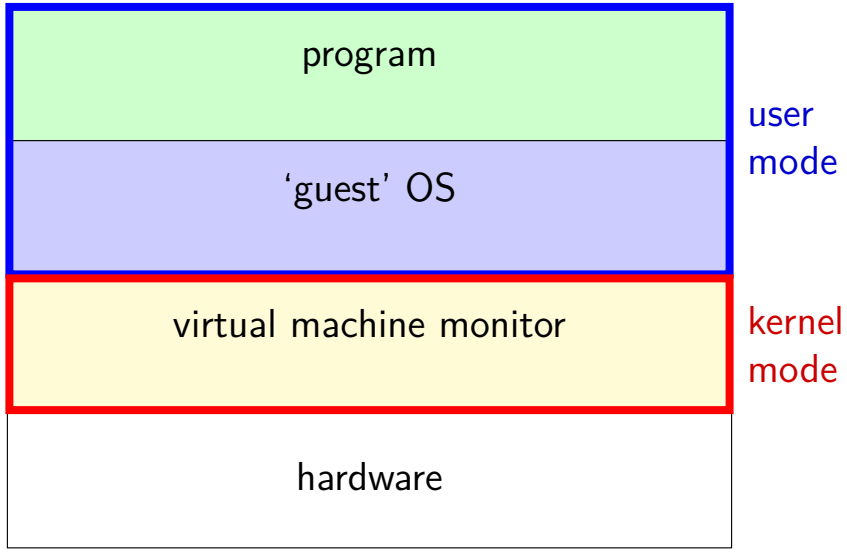
emulating multiple levels

conceptual layering



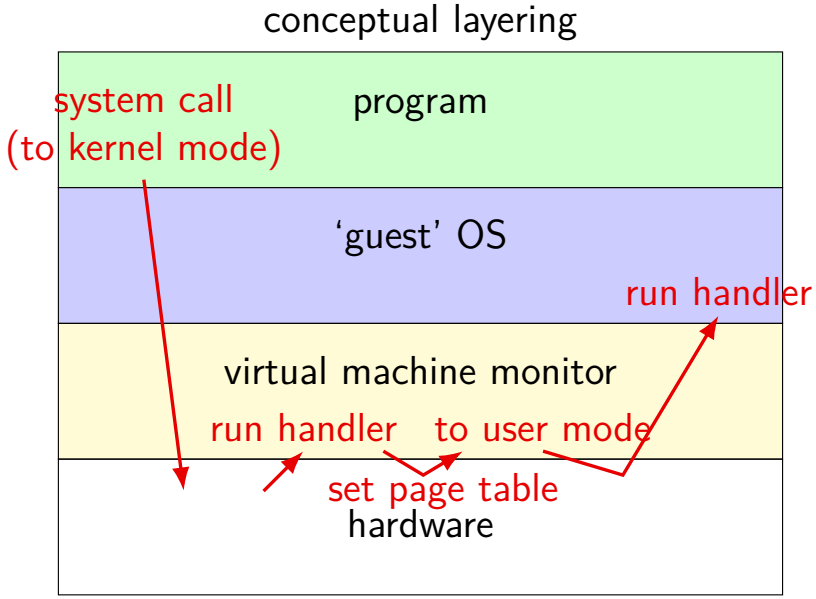
emulating multiple levels

conceptual layering



emulating multiple levels

conceptual layering



recall: page tables

program (virtual) address

0x12345678

page table lookup

0x00044678

real (physical) address

page table

virtual page #	physical page #	permissions
00000	(invalid)	none
00001	00434	read/exec
00002	00454	read/write
00003	00042	read/write
...	...	...
12344	00145	read/execute
12345	00149	read/execute
12346	00151	read/execute
...	...	...

recall: hierarchical page tables

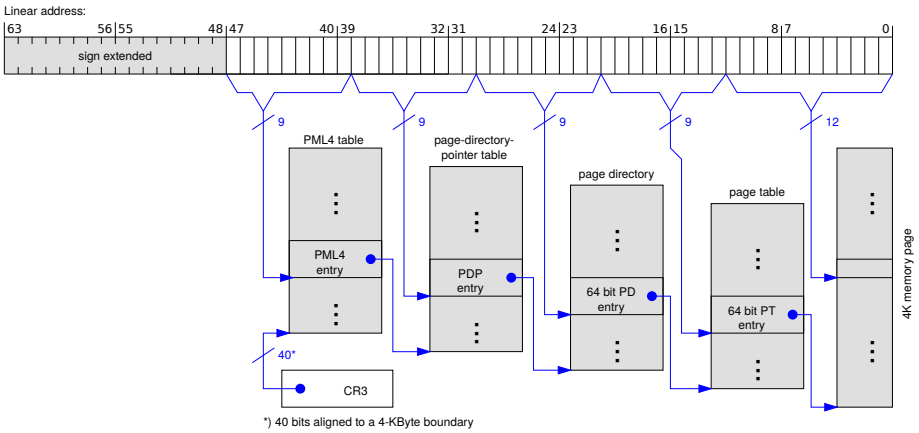
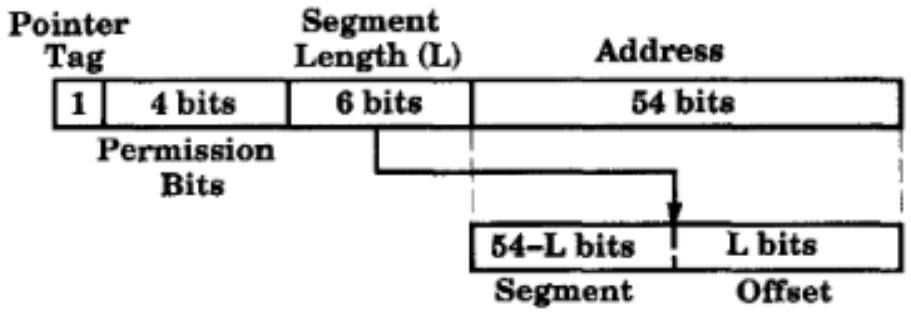


Diagram: Wikimedia / RokerHRO

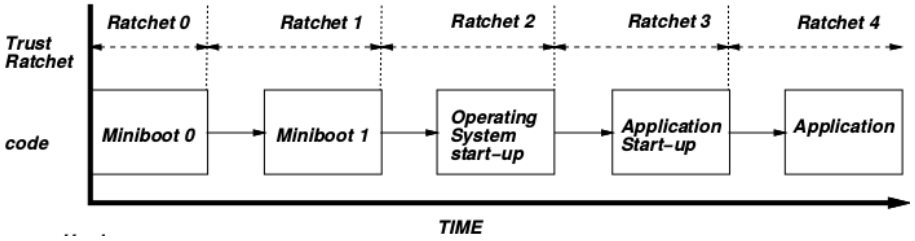
tagged architectures



key trick: seperate pointer instructions
otherwise pointer tag becomes 0

Figure from Carter et al, "Hardware Support for Fast Capability-Based Addressing"

hardware ratchets



	Ratchet 0 (Miniboot 0)	Ratchet 1 (Miniboot 1)	Ratchet 2 (OS start–up)	Ratchet 3 (Application start–up)	Ratchet 4 (Application)
Protected Page 0	READ, WRITE ALLOWED			NO ACCESS	
Protected Page 1					
Protected Page 2					
Protected Page 3					

hardware ratchets: code loading

	Ratchet 0 (Miniboot 0)	Ratchet 1 (Miniboot 1)	Ratchet 2 (OS start–up)	Ratchet 3 (Application start–up)	Ratchet 4 (Application)
Protected Segment 1 (Miniboot 1)	READ, WRITE ALLOWED		READ ALLOWED, WRITE PROHIBITED		
Protected Segment 2 (Operating System)					
Protected Segment 3 (Application)					

hardware security categories

protection of software from software (page tables, kernel mode)

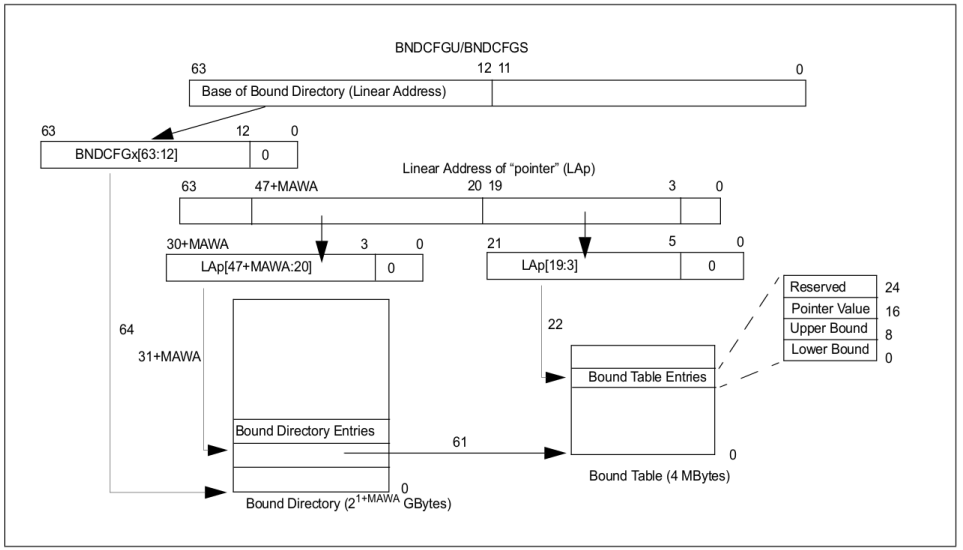
secondary topic of the paper for today

aid in producing vulnerability free code (bounds checking, no-execute bit)

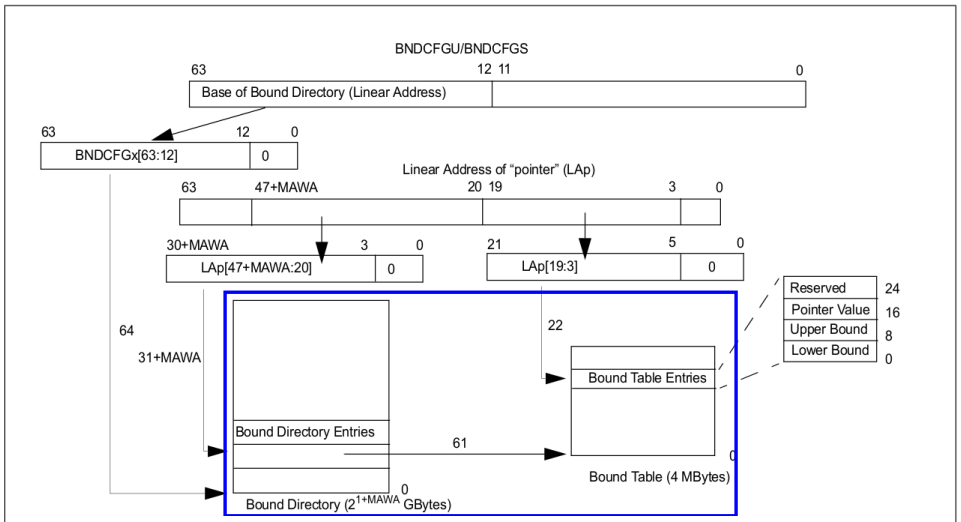
protect code from people with access to hardware

primary topic of the paper for today

hardware-assisted bounds checking

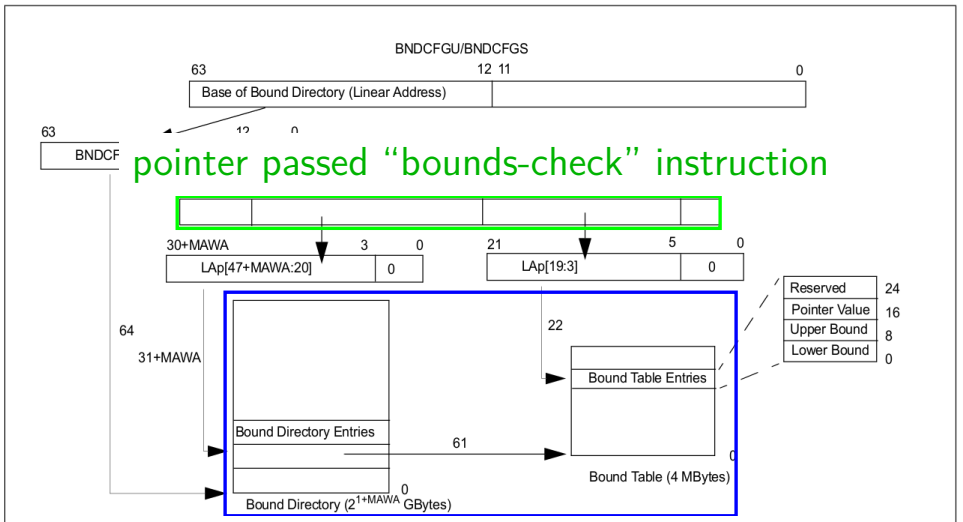


hardware-assisted bounds checking



"page table" for array bounds

hardware-assisted bounds checking



"page table" for array bounds

other hardware assistence (1)

write XOR execute

- memory can only be writable or executable, not both
- makes buffer overflows hardware (not impossible)

trap on access to user-accessible memory in kernel mode

- Intel name: "Supervisor Mode Access Pervention"
- operating system disables when intentionally accessing user data
- prevents accidental use of user pointers by OS

hardware security categories

protection of software from software (page tables, kernel mode)

- secondary topic of the paper for today

aid in producing vulnerability free code (bounds checking, no-execute bit)

protect code from people with access to hardware

- primary topic of the paper for today

tamper

tamper evidence

tamper resistence

tamper detection

tamper response

tamper

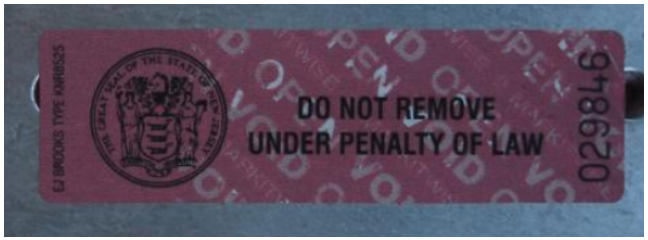
tamper evidence

tamper resistence

tamper detection

tamper response

tamper-evidence



Appel, "Security Seals on Voting Machines: A Case Study"

tamper

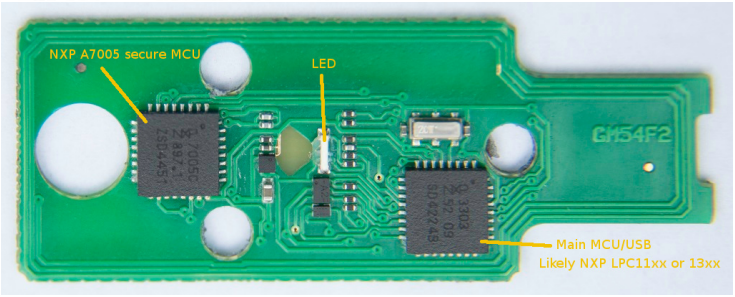
tamper evidence

tamper resistence

tamper detection

tamper response

tamper-resistence/evidence



2nd image: HexView "Inside YubiKey Neo" http://www.hexview.com/~scl/neo/

tamper

tamper evidence

tamper resistence

tamper detection

tamper response

tamper-detection

add sensor to detect tampering

e.g. checksum of code

e.g. switch if case is opened

tamper

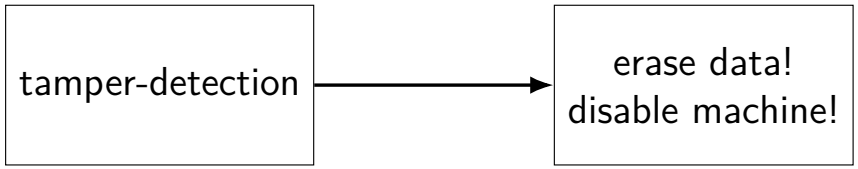
tamper evidence

tamper resistence

tamper detection

tamper response

tamper-response



secure co-processor protection goals

device has secret data

tampering must not reveal secrets

tampering must not let new software access secrets

kinds of “tampering”

replacing software

accessing the memory with another device

physically manipulating the device

kinds of “tampering”

replacing software

accessing the memory with another device

physically manipulating the device

securing the software

basic idea: load new software = erase old secrets

supporting software upgrades

verify with cryptography!

public key cryptography (1)

Smith and Weingart make extensive use of digital signatures

digital signatures use a public/private keypair

example use case: A wants to email B and have B know A wrote the email

public key-cryptography (2)

A generates keypair for communicating with B

public key: given to B; serves as identity/name

- assumed known by/safe to tell everyone

private key: kept secret by A

- assumed no one else has private key

public key cryptography (3)

two mathematical functions:

$$\textit{signature} = \mathbf{Sign}(\text{A's private key}, \text{message})$$

$$\textit{correct?} = \mathbf{Verify}(\text{A's public key}, \text{message}, \text{signature})$$

Verify will only say correct if private key was used

- computationally infeasible to "forge" signature

A uses **Sign** operation, sends message and signature

B uses **Verify** operation; rejects if it says "not correct"

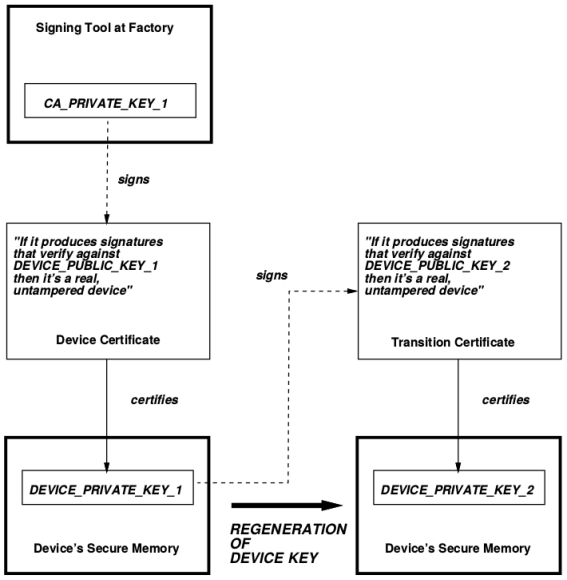
cryptographic software update

application is loaded with public key

updates to application must include $\mathbf{Sign}(\text{private key}, \text{the code})$

if not, secrets are wiped on update

signature chain



verifying signature chain

You get:

Sign$(factoryprivkey,$ "Device PubKey 1 is a device key")

Sign$(deviceprivkey1,$ "Device PubKey 2 is a device key")
Sign$(deviceprivkey2,$ "I generated this output")

need to check all signatures in the chain

can be used for application updates/messages

chain is device to OS to application

enforcing updates zeroing

checks signatures
zeroes data

	Ratchet 0 (Miniboot 0)	Ratchet 1 (Miniboot 1)	Ratchet 2 (OS start-up)	Ratchet 3 (Application start-up)	Ratchet 4 (Application)
Protected Segment 1 (Miniboot 1)	READ, WRITE ALLOWED		READ ALLOWED, WRITE PROHIBITED		
Protected Segment 2 (Operating System)					
Protected Segment 3 (Application)					

kinds of "tampering"

replacing software

accessing the memory with another device

physically manipulating the device

secure(?) packaging

Figure from Ross Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*

secure(?) packaging

Figure 14.2 The 4758 partially opened, showing (from top left downward) the circuitry, aluminium electromagnetic shielding, tamper-sensing mesh and potting material (courtesy of Frank Stajano).

Figure from Ross Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*

power analysis

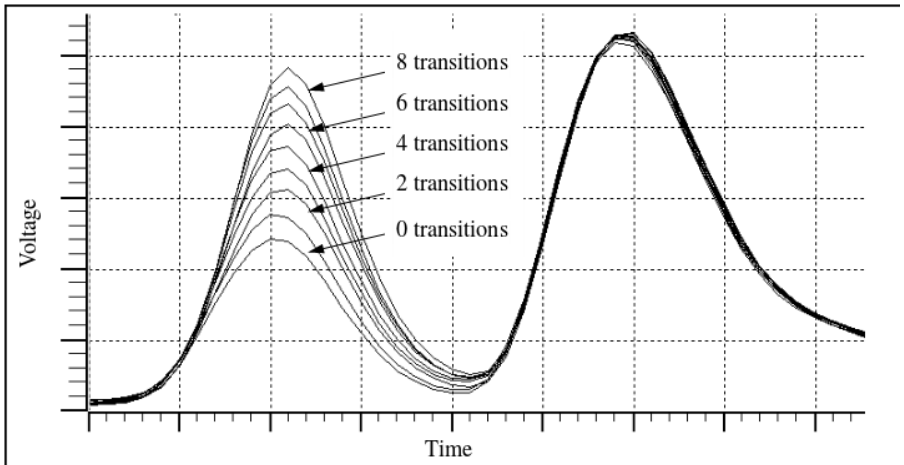


FIGURE 2. **Number of Bit Transitions versus Power Consumption**
These results show how the data effects the power levels. The nine overlayed waveforms correspond to the power traces of different data being accessed by an LDA instruction. These results were obtained by averaging the power signals across 500 samples in order to reduce the noise content. The difference in voltage between i transitions and i+1 transitions is about 6.5 mV.

Messerges et al, "Investigations of Power Analysis Attacks on Smartcards"

memory permanence

values can be "burned" into some memories

even RAMs that "go away" when they lose power

IBM's solution

circuitry to "buffer" power to processor

- limit information available from power consumption

active SRAM erasing circuitry

- cannot just cut power and hope

move values in SRAM to avoid "burning" them in

kinds of "tampering"

replacing software

accessing the memory with another device

physically manipulating the device

ways to make devices do weird things

all these can break CPU operation, or SRAM zeroing:

temperature

ionizing radiation

changing voltages

changing clock signals

... probably lots more

IBM's way of dealing with weirdness

sensors:

temperature sensor

radiation sensor

voltage sensor

phase-locked loops to sync clocks

focused ion beam (on a smart card)

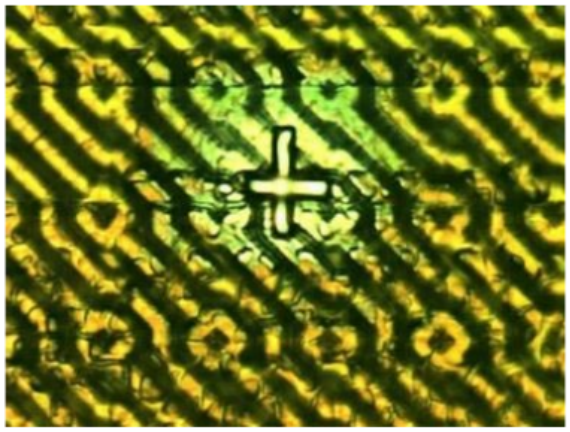


Figure 11: A FIB was used here to drill a fine hole to a bus line through the gap between two sensor mesh lines, refill it with metal, and place a metal cross on top for easy microprobing access.

Kommerling and Kuhn, "Design Principles for Tamper-Resistant Smartcard Processors"

attestation

attestation — know what code is running

mechanism

private key loaded at factory

loading code (miniboot) signs message saying:

- what application it loaded
- the public part of a keypair it generated

this message is a certificate for the application

attestation — verifying

application signs "yes, I really computed X" using its private key

anyone can verify this with miniboot's certificate

attestation use case — public cloud

I run a VM on Amazon

How can I verify Amazon is really running my code?

How can I keep Amazon from getting my data?

attestation use case — public cloud

I run a VM on Amazon

How can I verify Amazon is really running my code?

How can I keep Amazon from getting my data?

attestation use case — public cloud

I run a VM on Amazon

How can I verify Amazon is really running my code?

How can I keep Amazon from getting my data?

cryptography assumption

known public keys for signatures can setup encrypted communication

... even in the presence of insecure networks

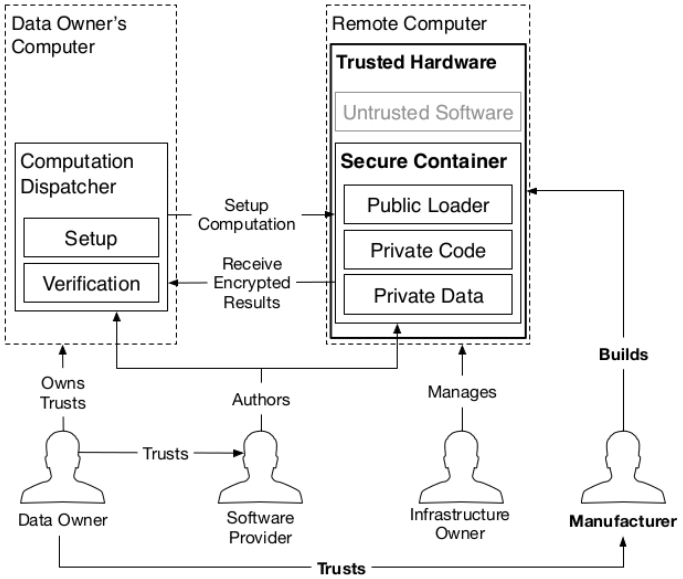
Secure Enclaves and SGX

Intel CPU extension called "SGX"

"Secure Enclaves"

provides isolated execution and remote attestation

trusted computing



Costan and Devadas, "Intel SGX Explained"

SGX isolation

run on same CPU as potentially malicious OS

CPU must enforce protections

OS gives memory to enclave

CPU prevents OS from accessing enclave memory

- modification to pagetable lookup
- memory encryption/authentication

effectively CPU microcode has mini-OS

SGX paging

OS can't control memory??

SGX can ask for pages to be removed from enclave memory

memory is encrypted before being released to OS

SGX and physical attacks

SGX includes memory encryption

processor encrypts data that goes off-chip

also uses a message authentication code to detect tampering with data in memory

SGX and side-channel attacks

SGX doesn't protect against side-channels

how long does computation take?

cache timing — like in the Bernstein paper (much earlier in semester?)

OS/HW owner can do a lot to observe system

much more than scenario in Bernstein

SGX and physical attacks

hope — physically reading key is hard

very small process size

hard to use key

observing cache behavior

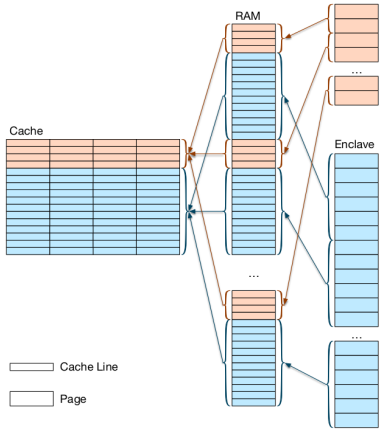


Figure 94: A malicious OS can partition a cache between the software running inside an enclave and its own malicious code. Both the OS and the enclave software have cache sets dedicated to them. When allocating DRAM to itself and to the enclave software, the malicious OS is careful to only use DRAM regions that map to the appropriate cache sets. On a system with an Intel CPU, the the OS can partition the L2 cache by manipulating the page tables in a way that is completely oblivious to the enclave's software.

normally, it's hard to isolate OS behavior from effects of other things on the caches, but the OS can fix that

Costan and Devadas, "Intel SGX Explained"

other side-channel tricks

OS can get make every enclave access page fault

full page-level memory access pattern

OS can run on other hyperthread!

sometimes with shared branch predictors

OS can make interrupts happen all the time

is this a security problem?

research areas in HW security

hardware verification
- what if I don't trust Intel?
- what if I don't trust my outsourced fab?

running sensitive code efficiently on same HW

support for OS security
- with low overhead
- easy to verify correctness

efficient side-channel resistence