System Security

Yannick Merkli, ymerkli@ethz.ch

ETH Zurich, HS 2020

Abstract

This document is a handwritten lecture summary of the *System Security* lecture at ETH Zurich. This summary was created during the fall semester 2020.

SysSec notes

1 Intro + Side channels

- **RSA:**
  1) primes $p, q \rightarrow N = p \cdot q$
  2) Euler totient: $\phi(N) = (p-1)(q-1)$
  3) Choose e s.t. $\gcd(e, \phi(N)) = 1$
  4) Compute d s.t. $e \cdot d \bmod \phi(N) = 1$
  5) public key: (e, N) private key: (d, N)

- Attacking crypto systems: most formal models don't take implementation into account → side channels.

- Timing side channel: key-dependent branching (e.g. found in Square-and-Multiply).

  - Defense: Masking by Blocks
    → pick random X for each message M:

$$SICM(m) = [(m \cdot X)^d \bmod n] \cdot [(X^{-1})^d \bmod n] \bmod n = m^d \bmod n$$

2 Side Channels

- Simple Power Analysis (SPA): single execution (trace), depends on key
- Differential Power Analysis (DPA): many executions (traces), depends on key + input
- Power Cryptanalysis: mainly for Smartcards, RFID, etc. (attacker needs access to the device)
- Power consumption: $P_{static} = I_{DD} V_{DD}$, $P_{dyn} = \frac{1}{2} C V_{DD}^2 f$
- Why does power leak data? Transistors need to be charged (e.g. $0 \rightarrow 1$ needs power) depending on whether 0s or 1s are loaded. Thus, consumed voltage will track # of bit transitions.
- SPA: measure power consumption of instruction sequences that depend on key.
- DPA: power consumption of an instruction sequence depends on key and input
- Protection: - Desynchronization (random injection, dummy instructions);
  - Noise generator; - Filter at power input; - Software/Hardware balancing;
  - Shamir's countermeasure: decouple power consumption from charging by introducing a set of capacitors/batteries.

- Acoustic attack on RSA: need to provide chosen ciphertext
  ↳ based on high-frequency sounds caused by vibrations of electronic components (sound is a proxy for power consumption)
  ⇒ Different keys cause different sounds

  Attack tries to find longer sound patterns (not individual CPU operations these are too high frequency) that leak instructions

  Key is found bit-by-bit, crafting special input.

3 Tempest & HSM

- TEMPEST: side channel attack based on EM radiation
  Information leaking EM radiation can come from many sources (e.g. monitor cable, keyboards, light (→ reflections of screens from eye, spoon, coffee, ...))

- Tamper Resilience:
  - Tamper Resistant: Bank vault approach, prevent break-in (e.g. ATM).
  - Tamper Responding: burglar alarm approach, real-time detection of intrusion, erasure or destruction of sensitive data.
  - Tamper Evident: If a break-in occurs, evidence of the break-in is left, detection of intrusion

  ↳ Smartcards are tamper resistant, Cryptoprocessors are tamper resistant, tamper responding and tamper evident.

- Smartcards: PIN + card possession enable user authentication
  Card holds a key: tamper resistant, attacker can't extract key even when in possession of the card.
  BUT... Smartcards are vulnerable to side channels (e.g. photonic emission)

- Hardware Security Module (HSM): Special purpose hardware for highly secure crypto operations (used in banks, cryptocurrencies, ...)
  ↳ HSM offers functionality to the outside ⇒ exposes an API

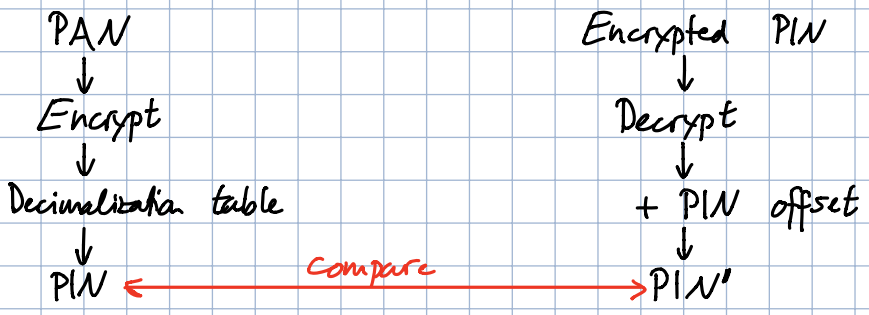
- API attacks: HSMs expose limited API
  - Security API: The top-level software component of a cryptoprocessor, which governs its interactions with the outside world.
  - Idea: Can we trick a secure HSM into leaking secrets by sending clever requests?
    ↳ difficulty of building leakage free APIs

  - API attack on PIN verification at Bank:
    Attacker has encrypted PIN + matching PAN. Wants to find PIN by sending requests to PIN verification.

PIN generation (inside HSM)

1) Encrypt PAN using bank key
2) Take first 4 characters of encrypted PAN
3) Use decimalization table
4) Add PIN offset

4556 2385 7753 2239
3F7C 2201 00CA 8AB3
3572
7816 + 4344



- Decimalization attack: change decimalization table, offset to find PIN
  1) Change decimalization table: 1123456789012345
     ↳ All 0s become 1s
     ⇒ Accept: no 0s in PIN
     Reject: at least one 0 in PIN
  2) If reject, change offset to localize the 0s
- Mitigation of API attacks: Access control, limit functionality
- Chip and PIN is broken: another attack on Smartcards.
  ↳ Can MITM terminal – card connection

4 Intro to Security on Commodity Systems (= PCs)

- Application Security: PCs run various applications with different security levels → isolate and protect
  Application Security requirements:
  - Launch time integrity: correct application was loaded or started
  - Run-time isolation: no inference from malicious software
  - Secure persistent storage: Confidentiality, integrity protection of persistent data

  ↳ One approach: Hardware-supported OS-based security (this chapter)
  → OS, hardware is trusted; applications, peripherals are not

- We consider X86 systems.
  Intel platform overview:
  - Processor: one or multiple CPUs
  - Chipset: connects processor to RAM and peripherals
  - Peripherals

today part of kernel

- Privilege rings: rings 0,1,2,3 → ring 0: Kernel; ring 1,2: device drivers; ring 3: applications

- **MMU:** Translates virtual addresses to physical addresses by walking page tables. Applications cannot change their own page tables, need to ask kernel.

- Paging-based security: security relevant data in page table entries
  - Supervisor bit: if set, page only accessible by ring 0
  - RW bits: read-only / writable page
  - Execution disabled (ED) bit: if set, page is not executable

- DMA attack: RAM is tightly controlled by CPU, but DMA can circumvent this. CPU is only asked initially, then DMA can access any memory location → can read out entire memory.
  Thunderbolt is vulnerable to same attack.
  Solution: destroy ports, disable DMA, ... IOMMU

  - IOMMU: problem: malicious peripheral in kernel mode can access any memory location when using DMA
    Solution: IOMMU, similar to MMU but restricts MMU access for peripherals to DMA target → IOMMU controls DMA access to physical memory, IOMMU set up by OS

    ⇒ IOMMU makes sure that compromised peripherals cannot do DMA attacks by restricting the physical memory space of the peripheral to the DMA target.

- **Attacks with physical access:** harder to defend for OS
  Possible solutions:
  - Password protect BIOS to prevent booting from external resource: broken, can just reset BIOS
  - Disk encryption:
    - password-only → bad, can use bruteforce
      ↳ disk encrypted with key that is protected by password
    - leverage secure element (TPM) → more secure
      ↳ disk encryption key in TPM, password unlocks TPM

  Disk encryption is block-wise and transparent middleware between disk and OS (data is always stored encrypted and only decrypted when loading into memory).
  Encryption key must be kept in memory... is this secure?

- **Cold Boot Attack:** Discovered that RAM capacitors don't immediately decay, especially when cooled down → data persists for up to ~10min

  ↳ Attack: user logs in, disk encryption key put into memory. User locks PC, key kept in memory. Attacker opens up PC, removes power, cools down RAM (~ −50°C), plugs RAM into another (aquisition) platform and recovers key.

Less invasive variant: remove power and boot from external media

↳ Protection:
- erase key from memory upon every suspend, retype password
- prevent booting from external media
- physical protection

- Launch time integrity: create chain of trust where each next loaded component is measured (hash verified)
  ↳ BIOS → Bootloader → OS → Application
  TPM support: leverage TPM, TPM performs chain of trust measurement

  ↳ Secure boot: OS boots only if chain of trust is valid.
  Authenticated boot: System logs chain of trust but OS boots even when chain is invalid.

- Case Study: San Bernadino iPhone case
  Phones leverage hardware support for storage encryption. Processor has device specific key (difficult to extract). Disk encryption key derived from PIN and processor key → storage must be decrypted on same device where CPU is. PIN bruteforce prevented by throttling. # PIN attempts stored in NVRAM.

  ↳ How to attack?

  NAND mirroring:
  1) NVRAM uses NAND chip → remove, rewire, reconnect chip
  2) Eavesdrop and read out NAND contents
  3) Create 1:1 copy of NAND chip
  4) Restore PIN: power on phone, try 6 PINs, power down, remove NAND and restore from backup, put NAND back in, repeat

⇒ General issue: Replay attacks
Processors can be enhanced with cryptographic keys → data confidentiality and authentication, but no integrity and freshness.

5 Introduction to TEEs

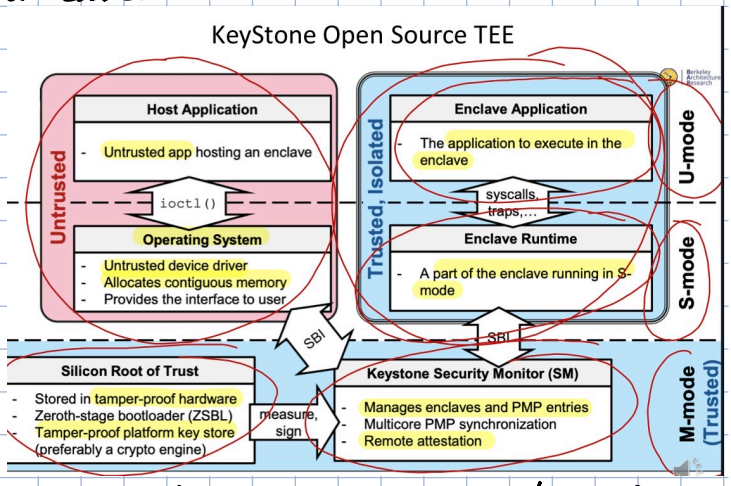
- Problem with OS-based security: OS is complex → lots of bugs

  ↳ Hardware Supported Secure Execution Environments
  ⇒ only hardware and target application trusted

- Isolation: OS shouldn't be able to inspect application's memory
  OS should still handle: memory, scheduling, peripherals

- Attestation: assure to remote party that correct process is running

- **KeyStone open source TEE**: based on RISC-V
  - RISC-V physical memory protection (PMP), U-, S-, M- mode
  - Entropy source, root of trust available



  - Separate OS from sensitive apps → OS can't control entire system, enclaves have separate memory.
  - OS can setup enclave but once running can no longer control
  - **Keystone Security Monitor (SM)**: trusted, controls how OS & enclaves are setup. Rather small (~ 10k LOC)
  - **Trusted Silicone**: can do trusted operations such as key store, hashing, encryption / decryption, etc.
  - S- / U- mode: separate privilege within app → S- mode is higher privileged, can e.g. do syscalls
  - Attestation: remote party communicates with OS, OS asks KSM for secure communication and KSM hash & verify enclave
  - PMP: Special registers to control permissions of U- / S- mode accesses to a specified memory region. Whitelist based, dynamically configurable by M-mode. Keystone uses N PMP registers where PMP0 for SM, PMPN for OS. 1 PMP entry per enclave.

    Each PMP registers has rwx bits for memory regions. By switching these bits dynamically, SM decides who can run what when.
    ↳ e.g.: execute enclave2: set PMPN (OS) bits to 000 and set PMP2 (enclave 2) bits to 111 for its region.

- **ARM TrustZone**: Strictly partition untrusted apps from trusted apps
  - Realized as additional CPU state: secure / normal world
    ↳ effectively only 2 enclaves
  - NS (non-secure) bit to tag secure data access
  - **Monitor mode** handles switch from/to normal/secure world, using special instruction.
  - Secure world is managed by **small trusted OS**

- Physical memory partitioning enforced by TrustZone Address Space Controller (TZASC)
- Problem: Only single security domain. Would require complex trusted OS to isolate sensitive apps in secure world. One app compromising secure world compromises entire system
  ⇒ prohibit non-vendor apps from running in secure world

- Sanctuary (based on ARM TrustZone):
  - Concept: small trusted app in secure world that manages and isolates multiple enclaves running in normal world.
  - Trusted OS in secure world sets up / tears down enclaves (in normal world) and configures TZASC.
  - Sensitive apps (in enclaves) run on their own core and get memory region assigned from trusted OS (via TZASC)
  - Memory separated into non-secure, enclave 1, enclave 2, ..., secure world. TZASC enforces access based on core ID and current world of the core.
    Normal world can't access enclave & secure world memory.
    Enclave can't access other enclave's & secure world memory
  - Sanctuary security services (in secure world) provides remote attestation and sealing.

- KeyStone vs TrustZone / Sanctuary:
  - Both isolate processes by controlling memory access with dedicated hardware
  - Keystone: enforcement at the core, policy at every core, synchronization
  - TrustZone / Sanctuary: slave-based enforcement, policy stored only once, requests can be flooded / spoofed by malicious core.

- Intel SGX: another TEE architecture using enclaves
  - Enclave runs in hardware protected memory. Enclave memory is encrypted and integrity protected at the processor boundary
  - Applications in user space untrusted but can call into enclave code
  - OS manages page tables, processor enforces that OS cannot access enclave pages. Enclaves can only access their own memory.
  - Processor Reserved Memory (PRM): memory region protected by CPU from non-enclave memory access.

Enclave Page Cache (EPC) part of PRM, 4 KB pages store enclave code and data.
Enclave Page Cache Metadata (EPCM) ensures correct assignment of physical pages to enclaves and virtual pages
↳ protect against address translation attacks

- SGX is MMU-based isolation: flexible, builds on existing mechanism, size of enclave limited (≈ 128 MB), full memory utilization, more attack surface (side channels)

- Remote Attestation in SGX:
  - Verify that enclave is running correct code, hasn't been tampered with, runs on a genuine SGX platform → Quoting enclave measures enclave (≙ hash over enclave pages and enclave page metadata) and signs report with attestation key. QE attestation key is signed by Intel to prove platform authenticity.
  - Attestation includes (signed) DH value to establish shared secret
  - Sealing key (secure persistent storage) and provisioning key burned into CPU during manufacturing

- SGX trusted components: Intel, CPU, Quoting Enclave, SGX trusted libraries

- Virtual vs Physical Memory isolation:

Virtual Memory isolation (SGX)	Physical Memory Isolation (Keystone, Trustzone)
+ Flexible	+ Simple
+ Full utilization of memory	+ Clear separation
− More attack surface	− Not flexible
	− Some memory wasted

- AMD Secure Encrypted Virtualization (SEV)
  - AMD security core in addition to normal cores
  - SEV allows memory content of a VM to be transparently encrypted with a key unique to the VM
  - OS / hypervisor are untrusted and setup VMs. Keys are never visible to software.
  - AMD SEV provides only confidentiality, no integrity

- How to use TEEs? Putting entire app in enclave increases TCB → identify security sensitive part and put only that in enclave.

- Rollback attack: Attack where attacker provides an old version of secure persistent storage data → need monotonic counter to protect.
  None of the systems discussed in this chapter have good rollback protection (SGX has monotonic counter but it's too short, rolls over)
  Monotonic counter requires NVRAM (expensive)
  Possible solution: ROTE, Matetic et al.

6 Microarchitectural attacks / Meltdown

Architecture
- Abstract model
- Instruction Set Architecture (ISA) specifies interface between HW and SW
- ISA example: X86, ARM, RISC-V
- Architectural state: Registers, Main Memory
  ↳ can be accessed by apps

Microarchitecture
- actual implementation
- Follows ISA specification
- Examples: intel i7, AMD Ryzen, ...
- Microarchitectural state: Caches, Branch Prediction history, Reorder buffer
  ↳ cannot directly be accessed by apps

- X86:
  - syntax: op dest, src
    ↳ add eax, ebx → eax = eax + ebx
      mov eax, [ebx] → move memory content at address ebx into eax

- Microarchitectural optimizations: memory is too slow, processor often waiting
  ↳ Hide memory latency: caches, pipelining, out-of-order execution

- Caches: put accessed memory content into cache. If a memory address is cached → data access much faster
  ↳ Cache is shared among all applications (on a processor). Cache location depends on data address. Loading new data evicts old data.

- Pipelining: each instruction can be split into: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM) and register writeback (WB)
  ⇝ Pipelining: do IF of 2. instruction directly after IF of 1. instruction finished

- Out-of-order execution: Instructions that do not depend on each other can be executed out-of-order. Can parallelize execution stage and utilize all execution units (ALU, FPU, ...). Retire in-order.

- Cache Side Channel Attacks: Cache misses leak information. Attacker must control an application on the same processor/system.

  - Flush & Reload: Attacker and victim have shared memory
    1) Attacker flushes shared memory from cache
    2) Victim accesses memory at flushed location → memory is reloaded into cache
    3) Attacker accesses shared memory and times access:
       - fast access: victim accessed this address
       - slow access: victim didn't access this address

  ⇒ Attacker can find out what data a victim accesses
  ↳ real problem in e.g. cloud environments

- Prime & Probe: Works also without shared memory
  1) Attacker fills cache with his own data (prime)
  2) Victim loads some data → fills cache
  3) Attacker probes memory for his own data:
     - fast access: victim didn't access
     - slow access: victim accessed

- Meltdown:
  - Microarchitectural implementation of memory access: mov eax, [0x400]
    1) Check if address is cached (if yes, goto 6)
    2) If not, translate virtual address to physical address
       ↳ Walk page tables
    3) Issue memory request for physical address
    4) Wait for reply (*)
    5) Save value in temporary register and cache the page walk translation
    6) Retire instruction:
       - Check if permissions are met (check PTE bits)
       - If yes, save value in the register (eax)
       - If not: raise an exception
         ↳ flush pipeline and call OS exception handler

(*) Other instructions can use the data before the instruction retires, due to out-of-order execution.

  - Meltdown:
    - An invalid memory access still fetches the data from memory, permissions are only checked in step 6.
    - Returned data (in temporary register) becomes available to out-of-order instructions.
    - No architectural change visible, however, transient instructions have microarchitectural side effects.
      (transient instructions = OoO executed instructions that should not have been executed)

access kernel memory from unprivileged process

Virt. mem.

```
1: mov eax, [kernel_address]
2: mov ebx, [probe_array + 4096 * eax]
```

L1 cache

probe_array is the base point to a large array

A page whose value depends on eax is put into cache

    - Multiplicative factor (4096) shifts each index by 4096 B = 4 KiB = = size of a page → each index on its own page.
      CPU often prefetches data to cache that it thinks will be used soon, BUT x86 doesn't prefetch across memory page boundaries. By mapping each index to a different page, we make sure the prefetches doesn't introduce other data to cache.

↳ Instruction 1: will cause an exception. However, permission check for 1: is only done at the end → 2: is executed OoO and can use the eax proxy stored in a temporary register. Depending on the value of eax, 2: then modifies a different cache line. Afterwards, the attacker can check which page from the probe_array is cached, which gives him the value of eax and thus the value stored at kernel_address.

Attacker can avoid exception from 1: by using custom exception handler. Any side-channel can be used to extract kernel_address content (Flush & Reload, Prime & Probe, ...)

Attack based on 2 conditions:
- eax available OoO before exception thrown
- eax-dependent entry is cached before 1: retires

What causes the attack? Architecture state is consistent but microarchitectural state is inconsistent → attack is a consequence of microarchitectural optimization.

1) Access kernel memory from an unprivileged process
2) Use the returned value in transient instructions to modify different cache lines based on the address
3) Preferably avoid the exception (e.g. custom handler, or use TSX)
4) Use a side-channel to know the value read from kernel memory
5) Repeat to read every physical address of interest

1) The attack allows to bypass **hardware** memory privilege checks
2) It exploits OoO execution and a race condition in the permission checks
   a) The race condition allows inaccessible memory data to be loaded to the CPU core
   b) OoO enables subsequent instructions to operate on the data
   c) The architectural state stays consistent
   d) The microarchitectural state is not rolled-back
3) We can use a side channel to move the information from the microarchitectural state to the architectural state
4) Using a cache side-channel researchers were able to dump memory with a speed of up to 503 KB/s [1]

7 Software-Only Root of Trust

- Trustworthy execution on untrustworthy platforms **without any hardware support** → Why not use HW support?
  ↳ Legacy devices, TPM/SGX have flaws, software-based attestation requires no secrets, enables new applications

- Setting: **trusted verifier V** knows expected memory content of **untrusted device D**. V wants to obtain proof of D's memory content, obtain memory integrity. D executes verification function VF.

- **Reflection**:
  1) Fill memory with pseudo-random content (except where code is)
  2) Clear system state, disable interrupts
  3) Compute hash over entire memory
  4) Return hash and system state
  5) Verifies checks duration of computation, hash, system state

  ↳ Assumes that it's difficult for attacker to quickly modify all memory.

- **Genuinity**: Slightly different setting → V wants to check code integrity, code execution and that the code ran on the machine it was expected to run.
  1) V sends checksum code to D
  2) Verification function uses pseudo-random access pattern → causes unpredictable cache misses → alters performance
  3) Incorporate these hardware parameters into hash

  ↳ Assumes that simulating VF is much slower due to complexity of simulating architectural feature.

- **SWATT** (software based attestation for embedded devices):
  1) V sends nonce to D (as seed for pseudo-random access pattern)
  2) Pseudo-random memory traversal to compute memory checksum.
  3) V times checksum computation and verifies checksum

  ↳ Malicious D must verify each memory access and replace memory reads of changed memory with expected content → time overhead
  ⇒ If adversary does spoof memory → detectable time overhead
  If adversary doesn't spoof memory → wrong checksum

  Assumptions:
  - V knows hardware configuration of D (in particular, clock speed)
  - No proxy attack: D did not contact faster host
    ↳ challenging in wireless networks → want wired connection
  - Attacker model: can change SW, can't change HW (especially not clock speed)
  - Optimal implementation: code cannot be optimized

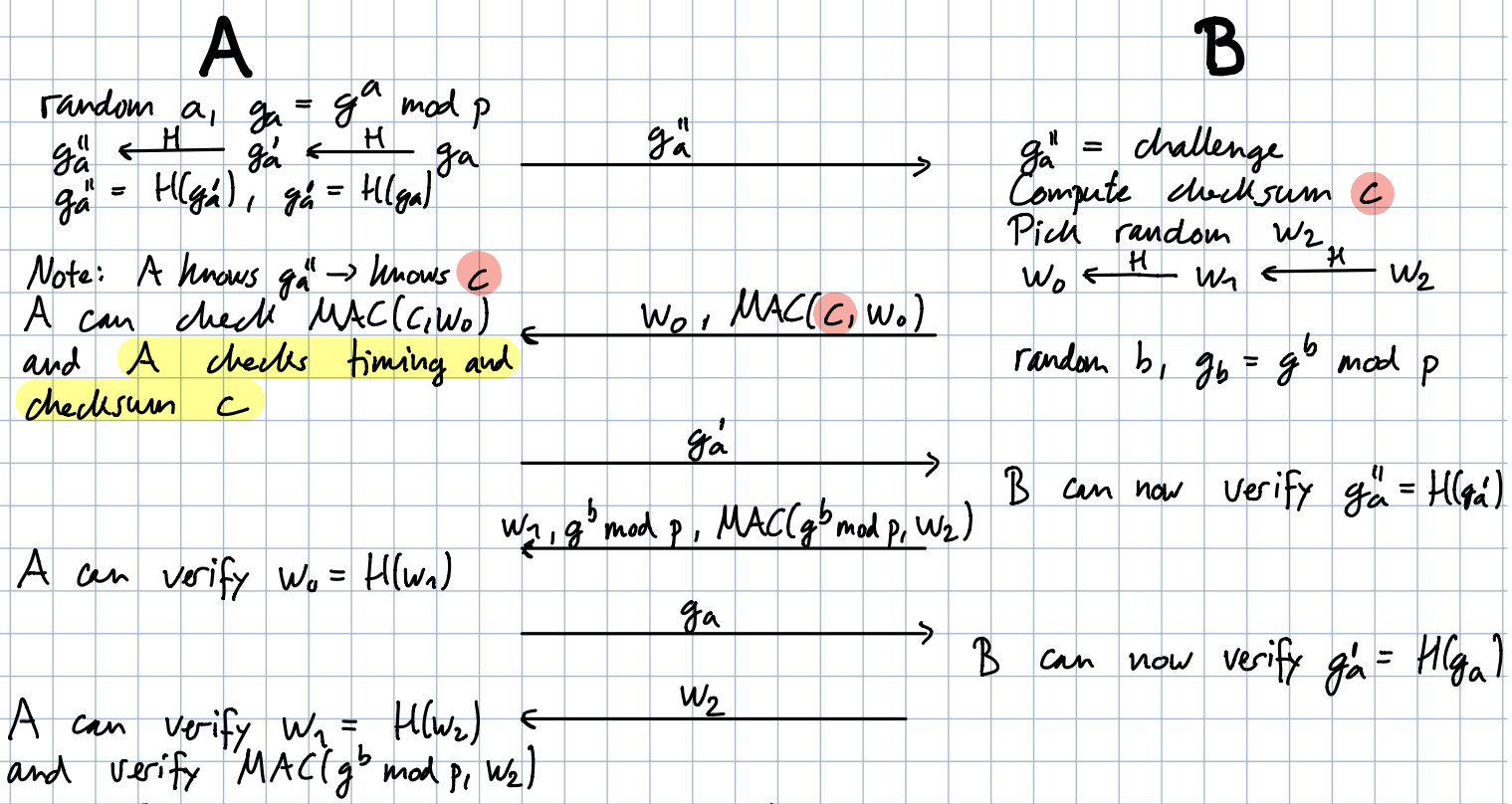
- No algebraic optimizations → checksum has to be computed entirely, given a memory change, checksum cannot be "adjust" without recomputation.

Drawback: checksum over entire memory, doesn't scale to large memory sizes, memory may contain dynamic data, ...

⇓

- ICE: SWATT with improved checksum that can check small memory areas (memory seen includes checksum code itself)
  - Adds checksum function execution state to checksum → include program counter (PC) and data pointer.
    - In memory copy attack (≙ attacker computes checksum over correct copy of memory) one or both will differ from original value. Attempts to forge PC and/or data pointer increase attackers execution time.

- ICE key exchange: Leverage ICE to compute checksum faster than any other node, use checksum as short-lived shared secret
  ↳ use authenticated DH with the short-lived shared secret for authentication. Use one-way hash chains with delayed disclosure.



⇒ Protocol can prevent MITM attacks without authentic information or shared secret. Attackers can know entire memory content of both parties before protocol runs.
↳ Allows secret establishment between nodes of sensor network

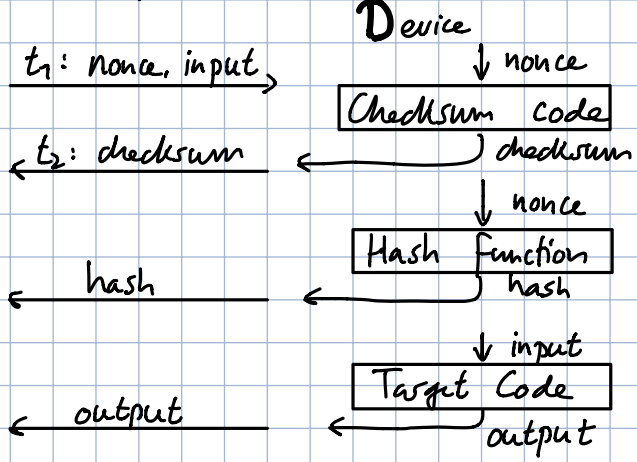
ICE key establishment assumptions: attacker can't compute faster than sensor node, each node has secure unique identity, secure random numbers

- PIONEER: SWATT is not applicable to modern machines → e.g. reading out PC takes ~200 cycles ⇒ too large overhead, gives speedup to adversary. Further, modern machines are unpredictable (OOO execution, cache and virtual memory, thermal effects, ...)

Verification Function: responsible for performing an integrity measurement on the executable, setting up a TEE for the executable and invoking the executable. Consists of 3 parts:

- Checksum code: Computes checksum over entire verification function and sets up a TEE in which send function, hash function and the executable are guaranteed to run untampered. An adversary trying to manipulate checksum computation causes computation time increase → detectable.
  ↳ correct checksum and correct timing gives dispatcher guarantee that there is a dynamic root of trust on the untrusted platform
- Hash function: SHA-1 for integrity measurement, computes hash of executable as a function of the nonce.
- Send function: Returns checksum and integrity measurement to dispatcher

PIONEER protocol:



↳ once you know checksum was correct, timing no longer relevant

Execution tampering attacks: Malicious OS/VMM gets control through installed malicious exceptions and interrupt handlers → attacker generates interrupt during verification function

↳ Solution: replace interrupt handlers after checksum loop (before or during would allow adversary to skip and gain time advantage).
What if adversary interrupts replacing handlers?
⇒ Stack trick: place part of checksum result on the stack right after stack pointer ⇝ if an interrupt happens: PC, flags, etc. are written to stack right after stack pointer overwriting checksum result
→ attacker doesn't have enough time to recompute checksum

⇒ Verifier can obtain untampered execution guarantee for code execution on untrusted platform: Code integrity, launch-time integrity

Drawback: requires defense against proxy and overclocking attack

8 Hardware-assisted security in ARM

- ARM: family of RISC. Lower power, less cost compared to e.g. x86. Commonly used in smartphones, IoT, embedded, ...

- ARM TrustZone architecture:
  - SOC on ARM: CPU is not an isolated chip but the CPU and many other components such as 4G/5G, WiFi, Bluetooth, ... Components are connected via some bus.

  - TrustZone: processor can operate in two domains
    → normal world, secure world
    CPU runs exclusively in one world (↝ NS bit)
    - Normal world: non-secure apps, non-secure (rich) OS, hypervisor
    - Secure world: secure apps/libs, secure OS
    ↳ In both worlds, we still have typical user & privileged mode

    Secure Monitor (SM): extra CPU mode, privileged instruction (Secure Monitor Call (SMC)) allows controlled entry to secure world.

  - Memory protection: TrustZone extensions to achieve memory partitioning
    - Address Space Controller (TZASC)
    - Memory Adapter (TZMA)
    - Separate translation tables in MMU

    } based on NS-bit (propagated on bus)
    NS = 1 → normal world

    Memory calls always go through TZASC and TZ-aware MMU.

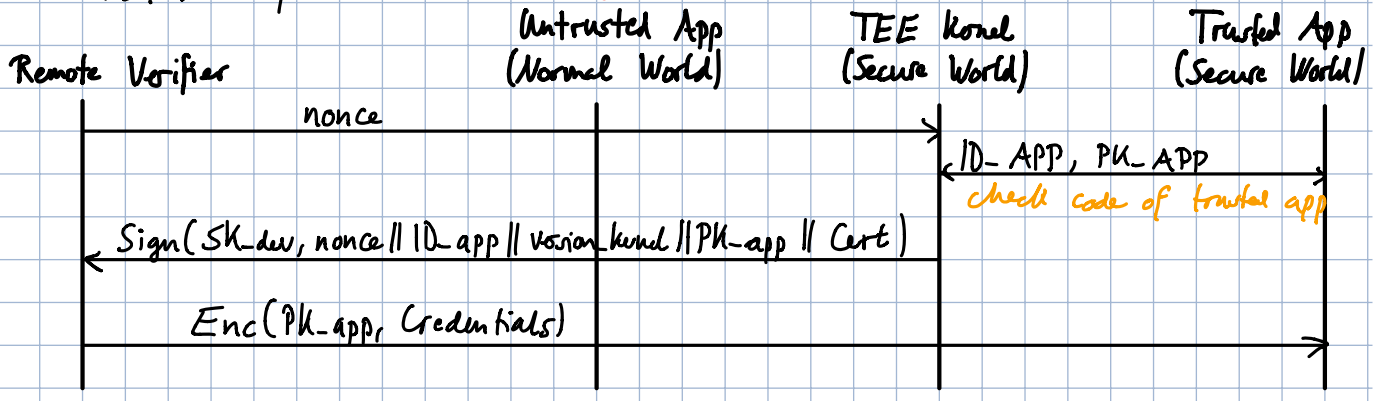
  - Device access:
    - TZ Protection Controller (TZPC): control access to peripherals
    - Interrupt controllers: prioritization extension (e.g. handle secure interrupts with higher priority) → prevent DOS by normal world

  - ARM Cortex-A: for smartphones etc., explained above

  - ARM Cortex-M: TrustZone for Microcontrollers
    ↳ low power, real time, fast context switch (N/S world switch using new instruction (Secure Gateway (SG)) instead of secure monitor)
    Memory is partitioned, state depends on where code executed
    → instruction pointer in secure memory ≙ state = secure
    → Secure and non-secure memory sections, non-secure callable (NSC)
    ↳ non-secure world can call NSC, which executes SG instruction to switch into secure world. Transitions also due to exceptions and interrupts.
    Main goal: avoid performance penalty of secure monitor mode (Cortex-A)

- TZ-based security features:
  - **Trusted Execution Environment:** reduce TCB for apps
    - Typical design: untrusted app in NW, trusted part in SW, TEE kernel (trusted OS) in SW
    - Secure and non-secure part of app communicate by passing pointers to memory address where input resides (readable by NW, SW)
    - NW can make SMC syscall to untrusted OS to access SW
    - Design for SW: only allow few "trusted" apps, need permission from ARM
    - Trusted OS provides basic functionality (libraries, secure storage, device access,...) to enable easy trusted app development

  - **Remote Attestation:** not part of TrustZone but can be implemented
    ↳ Attestation protocol sketch: assumes TEE kernel is trusted



  - **Secure Boot:**
    - Motives: Brand protection, OS controls access to safety relevant system resources, ...
    - Main approach:
      1) Boot in secure world supervisor mode. Code that executes needs to be integrity protected (e.g. on-chip ROM)
      2) Verify boot loader (e.g. check that boot loader was signed by manufacturer) → requires an immutable root of trust on device, e.g. manufacturer public key in on-chip ROM
      3) Once boot loader is verified, start it in normal world and boot loader can then verify integrity of the rest of OS

  - **Full Disk Encryption:** protect user data from theft
    - Main approach: derive disk encryption key from secure world device key and user provided password / PIN. Rate limit password/PIN attempts.
    - Challenges: NVRAM not always available, off-chip NVRAM can be susceptible to replay attacks

  - **Hardware Backed Key Store:** enable 3rd party apps to store keys with hardware protection → protect against device theft, malicious NW apps
    - Create security API (create key, sign, encrypt/decrypt,...), private key never leaves secure world.
    - Challenge: can untrusted OS sign arbitrary content with stored key via API... → user authorization, rate limit

- **Trusted UI**: include touchscreen drivers in TEE kernel
  - often mentioned as an advantage of TrustZone (over e.g. SGX)
  - User should be able to distinguish UI drawn by TEE kernel
  - Observe: secure output is easier than secure input

- Security Analysis:
  - TCB size: TEE kernels have non-trivial complexity (some vulnerabilities found). TrustZone only has two security domains (SGX has one security domain per enclave → better).

  - Other approach to more secure apps: use custom language and interpreter for trusted apps, minimize everything run in secure world

  - Message passing: Untrusted app provides pointer to some data to secure world, trusted app accesses it (secure world has access to all memory)
    ↳ **Boomerang attack**:
    → confused deputy attack (attacker = untrusted app, victim = untrusted OS, deputy = trusted app)
    1) Untrusted app prepares input struct with malicious pointer
    2) Untrusted OS passes it to trusted OS that sanitizes known parts of struct → trusted OS has no real understanding of application specific data, can't cleanly sanitize!
    3) Trusted app acts upon pointer (e.g. writes kernel data)
    → privilege escalation

  - Physical attacks:
    - Secure world only on-chip memory: good physical attack protection but limited TEE kernel and trusted app size.
    - Secure world also off-chip memory: more complex trusted app but physical attacks become easier

  - Microarchitectural attacks:
    - Side channels are common concern on any platform where different security domains share resources.
      ↳ In TrustZone, NW and SW share CPU, cache, memory controller, bus, ...
    - Side channel attacks on TrustZone exist but typically have some preconditions:
      - victim code has secret-dependent branching
      - Adversary is able to interrupt victim at high frequency
    - Fault injection another common attack vector → introduce a fault that e.g. skips security-critical check
      ↳ e.g. software-based cross-core voltage manipulation to generate hardware faults in secure world execution

  - Experience: billions of ARM devices in use, only very few reported attacks and vulnerabilities → intended goal has been largely achieved

9 & 10 TCG attestation

- Want correct execution of small amount of code even under presence of malicious software on system. Want external entity to verify this.
  Assume remote adversary that can compromise local OS and apps running on the OS and that can completely control network communication.
  Local hardware assumed trusted, but adversary can reboot, use malicious USB device.

- Some current approaches: evaluate based on TCB
  - Program code in ROM: keep entire program in ROM
    Advantage: simple, no injection possible
    Disadvantage: Cannot update, control-flow attacks (ROP) possible, entire system in TCB
    ⇒ not practical for current systems, can't tell what's happening from externally

  - Secure or verified boot: only load code with valid signature
    Advantage: Only approved code can be loaded
    Disadvantage: entire system in TCB (large OS certainly has vulnerability)
    → adversary only needs to compromise single component, unclear what is currently running on system (time-of-check-to-time-of-use (TOCTTOU)), software or certificate revocation requires state (→ correct time is critical to prevent rollback attacks.
    ⇒ weak security guarantees

  - Virtual-machine-based Isolation: isolate applications by executing them inside different virtual machines.
    Advantages: VMM smaller than OS (assumed to be secure), smaller TCB, isolation between applications
    Disadvantages: VMM still large and part of TCB, relatively complex solution, complicated interaction between applications
    ⇒ smaller TCB, step in right direction

  ⇝ All these systems lack attestation for software integrity
  ⇒ Attestation enables verifier V to verify what software is executing on untrusted device D

- General approach to attestation:
  1) Establish isolated execution environment
  2) Externally validate correctness of execution environment
     ↳ use external root of trust to establish local root of trust
  3) Autonomous launch and operation of execution environment

- Three core mechanisms:
  1) Isolated execution
  2) Remote attestation
  3) Sealed storage

- **Trusted Platform Module (TPM)**: Low-cost chip that helps establishing secure operations and remote attestation. Modern microprocessors provide special instruction to interact with TPM (SKINIT, SENTER)

  - Core TPM goals: platform identity, remote attestation, sealed storage, secure counter. TPM is <u>not</u> a CPU, no general purpose computation.

  - **Basic TPM functions:**
    - Platform Configuration Registers (PCR) for integrity measurement chain: $PCR_{new} = SHA\text{-}1(PCR_{old} \| SHA\text{-}1(data))$
      ↳ securely store log of hashes, rely on 2nd preimage resistance of hash function.
      Note: only preimage resistance (not 2nd) of SHA-1 is broken.
      - Static PCRs: only reset at boot time
      - Dynamic PCR: initialized to -1 at boot time, reset to 0 upon entry to IEE

      Note: a list of all application hashes can be stored in non-secure part of system. PCR verify integrity via hash chain.

    - On-chip storage for storage root key (SRK)

    - Manufacturer certificate, e.g. $\{K_{TPM}\}_{K^{-1}_{IBM}}$

    - Remote attestation with PCRs and attestation identity key (AIK)
      ↳ AIK for signing PCR values: $\{PCR\}_{K^{-1}_{AIK}}$

    - Sealed storage with PCRs and SRK (only accessible under certain integrity measurement)

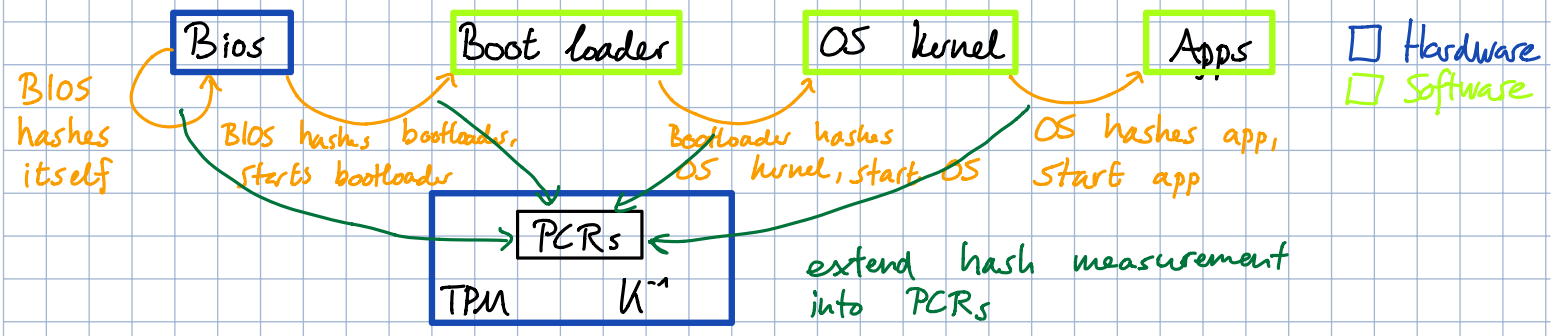
    - random number generator

  - **TPM is passive and not tamper proof** (private key can be read out, but requires expensive equipment).

  - TPM is not part of main processor, is connected via low pin count (LPC) bus → **LPC bus is very slow, attacks are possible (e.g. eavesdrop, inject)**

  - TPM-fail attack: secret dependent execution time during signature generation. Local adversary can recover ECDSA key.

- **Attested Boot (TCG 1.1):** Measurement of all executed software and configuration files defines platform configuration.
  To verify, remote verifier can inspect platform state by inspecting all measurements and compare to known hashes of trusted software.



- Attestation: remote verifier asks system what code it's running. System sends signed PCRs $\{PCRs\}_{k^{-1}}$ and if needed a list of application hashes.

- Note: Into which PCR we extend doesn't matter for attestation but matters for sealed storage → we don't want PCR to change with every new application.

- TCG 1.1 use cases:
  - Securing corporate network access: ensure that all systems that connect to corporate network have correct software.
  - Secure Online Banking: user wants to ensure malware-free platform
  - Secure Cloud Computing: verify cloud environment

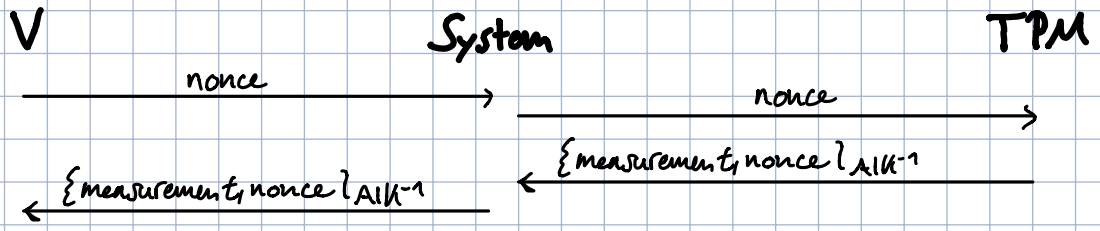
- **TCG 1.1 shortcomings:**
  - Integrity measurements are done at load time, not at runtime
    ↳ time-of-check-time-of-use (TOCTOU) problem, cannot detect dynamic attacks
  - Coarse-grained, measures entire system. TCB includes entire system. If one single application is vulnerable → PCRs match but attacker may be able to infect entire system.
  - **No guarantee of execution**
  - Every system is different → huge database of correct hashes required.

- **Dynamic Root of Trust (TCG 1.2):** special CPU instruction to create TEE → high assurance of code execution, achieve run-time attestation.

  DRTM computing primitives:

  - **Created isolated execution environment (≙ TEE):** create data that can only be accessed within TEE. Achieved via special instruction (SKINIT / SENTER) from AMD/Intel late launch extensions.
    Secure Loader Block (SLB) to execute in TEE.
    ↳ Create TEE with SKINIT/SENTER atomically:
    1) Soft reset CPU (CPU state similar to INIT)
    2) reset dynamic PCRs (to 0)
    3) Enable DMA protection for entire SLB
    4) Send SLB to contents to TPM (i.e. extend SLB content into dynamic PCRs)
    5) Begin execution at SLB entry point

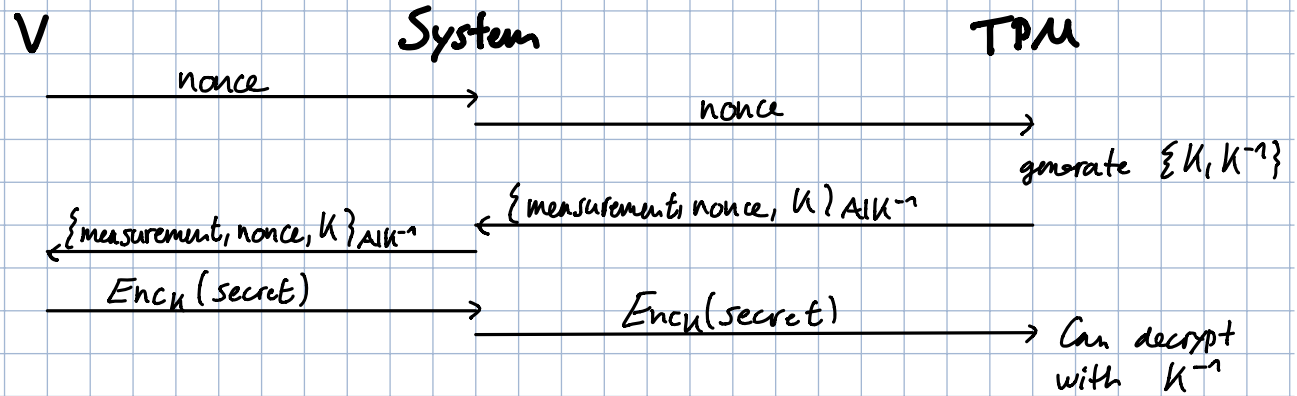
  - **Remote Attestation:**
    1) Trusted verifier sends nonce to TPM (nonce for freshness)
    2) TPM returns $\{$ measurement, nonce $\}$ signed with platform key



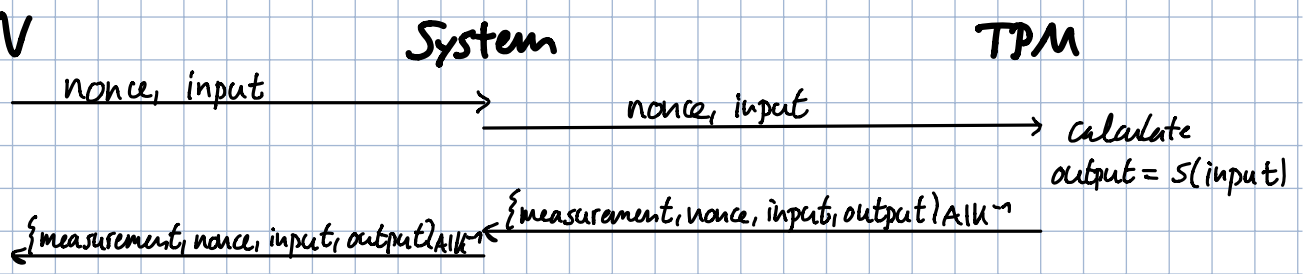
Trust in TCG1.2 attestation rooted in hardware TPM → very different from TCG1.1 where trust is rooted in everything that has executed so far (BIOS, boot loader, OS, application)

Note: attestation says a TPM signed measurement, but not which.
⇒ possible attack vector

  - **Secure channel to TEE:**
    1) Trusted verifier sends nonce to TPM
    2) TPM generates $\{K, K^{-1}\}$ and sends back
    $\{$ measurement, nonce, $K\}$ signed with platform key



- Verify output $O = S(I)$:
  1) Verifier sends nonce, input to TPM
  2) TPM sends back {measurement, nonce, input, output} signed with platform key



Security properties of late launch / DRTM:
- Similar properties to reboot (~ TCG1.1) BUT without reboot
- Many things removed from TCB (BIOS, boot loader, DMA-enabled devices, long running OS and apps) → smaller TCB!

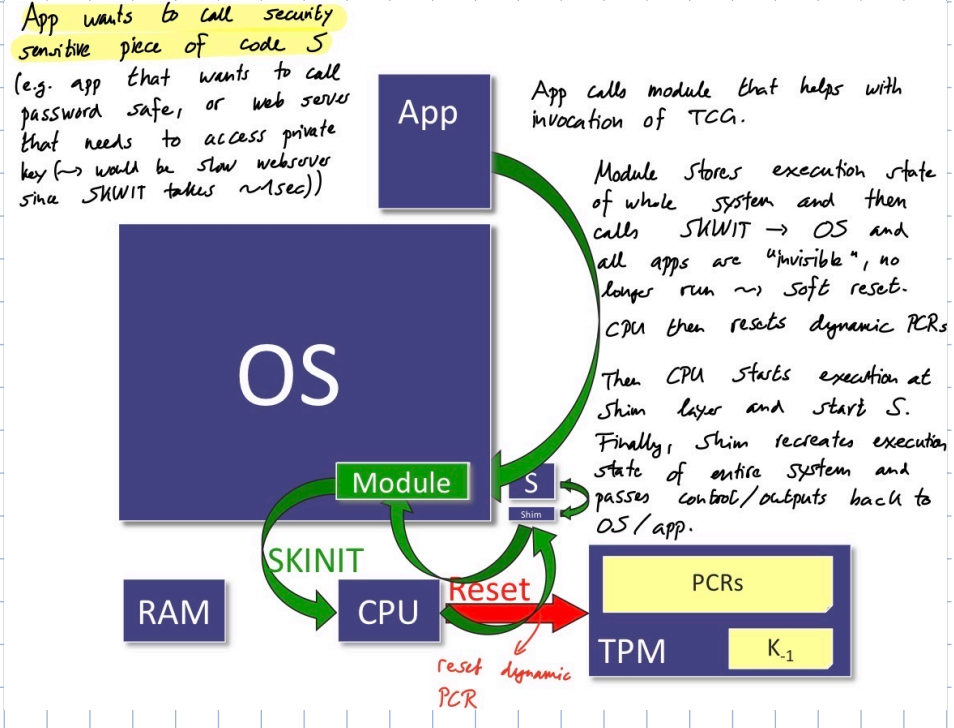
  ↳ Only hardware and application need to be trusted

TCG1.2 allows execution of high-assurance software next to existing legacy OS without the need to trust the OS.

TCG1.2 drawbacks:
- Memory is not encrypted
- Heavy weight operation: requires ~1s to execute SKINIT/SENTER
- No concurrency: entire system shuts down except single thread execution that calls SKINIT/SENTER
- DRTM cannot be virtualized (need to ensure that SKINIT/SENTER runs completely by itself.

- The Flicker System: real-life TCG1.2 system. Isolate security-sensitive code execution from all other code and devices. Attest to security-sensitive code and its arguments and nothing else
  ↳ Convince remote party that security-sensitive code was protected

  ⇒ Adds ~250 LoC to software TCB



- General question: can a local application perform attestation?
  ↳ No, this would not be secure. The OS has full control over local application and could e.g. just move the instruction pointer forward during signature verification → skip to successful verification without actually verifying.
  ⇒ We need a trusted remote verifier!

- Why can SKINIT/SENTER not be simulated by hardware?
  ↳ Dynamic PCRs are initially -1, SKINIT/SENTER resets to 0. Hardware enforces that only SKINIT/SENTER can reset dynamic PCRs to 0. Due to 2nd preimage resistance, initial dynamic PCR value -1 can be externally distinguished from initial dynamic PCR value 0. This allows to verify whether SKINIT/SENTER was actually run.
  ⇒ How can you verify a reboot actually happened? You can't! Adversary could simulate this. HOWEVER, adversary cannot simulate resetting dynamic PCRs.

- CUCKOO attack:
  - Problem: How can we ensure that the attestation originates from the correct host?

  - Attack: Adversary has access to some correct TPM chip and extracted the private key from that chip. Instead of using the local TPM, the malicious software uses the remote TPM of the attacker (redirect traffic) or the malicious software uses the extracted private key (which the adversary extracted from some TPM) and does attestation purely in software on the system (given a TPM private key, one can produce arbitrary attestations).
    ↳ TPM signed attestation only guarantees that the measurement was signed by a real TPM chip, but not which real TPM chip.

  - Mitigation:
    - Cryptographic secure channels: require authentic public key which prevents another TPM from sending attestations.
      Authentic public key:
      - Trust the BIOS: reboot and trust BIOS to output public key via existing interface
        ↳ Problem: adversary could fake rebooting, fake BIOS screen and display wrong public key
      - Seeing-is-believing: place e.g. barcode on PC that encodes TPM public key

    - Hardware secure channel:
      - reuse an existing interface (e.g. output TPM public key via USB or Bluetooth)
      - Add special purpose interface only for TPM

- SGX vs. TCG TPM

	SGX	TPM
+	• Memory encryption and authentication to protect against memory bus tampering and eavesdropping • Robust against LPC-bus tampering • Can run unprivileged code • Multi-threaded execution of enclaves • Parallel execution of enclaves and untrusted code • Enclaves are interruptable • Sealed storage	• Sealed storage • Monotonic counters for rollback protection + rate limiting • Simple attestation • No 3rd party support required • TCG sometimes more suitable for cloud computing with shared memory
−	• No / small monotonic counter • Complex, requires 3rd party support • Memory access patterns reveals information about computation • Concurrent execution suffers from side-channels • Encrypted memory may be a problem in cloud computing with shared memory	• No memory encryption • Vulnerable to LPC bus tampering • No multi-threaded execution of TEE • No parallel execution of TEE and untrusted code

- Possible attack vector for TCG: get your malware hash approved to known hashes database → focus of malware developers shifts to getting malware certified by TCG.

11 IoT Security

- **Lightbulbs go nuclear:** ZigBee Light Link (ZLL) protocol used for "smart" lightbulbs. Light switches can send scan request and send commands. Requests are encrypted under ZLL master key...
  ↳ ZLL master key shared between all ZLL devices → ZLL master key can easily be extracted (and is available online...)

  Can also **attack ZLL lightbulbs without ZLL master key:** Can use touchlink command to factory reset bulbs and then identify bulbs by blinking for a specified amount of time...
  Devices only accept commands if RSSI of signal is above a certain threshold → Assumption: attacker outside house can't get enough RSSI ⇒ Wrong, knowledgable attacker can easily get enough RSSI with correct antenna.

  ZigBee Over The Air (OTA) updates: standardize update mechanism, allows to update devices from different manufacturer.
  Problem: firmware updates are signed with symmetric MAC with the same firmware update key for all ZLL devices.
  ↳ Infecting IoT lightbulbs:
  1) Extract firmware update key from SoC using differential power analysis
  2) Circumvent proximity check (bug in ZLL stack) and perform factory reset.
  3) Make bulb join attacker's non-ZLL ZigBee network
  4) Perform OTA firmware update with malicious image, "signed" (MACed) with extracted key

  ↳ Can build an **IoT worm**, infected devices can infect new devices. How many devices to infect area A?
  → $N = 1.128 \cdot \frac{A}{\pi R^2}$, R: wireless ranges
  ↳ Assumes uniform device location distribution
  One could also use a drone to gap sparse regions (few devices).

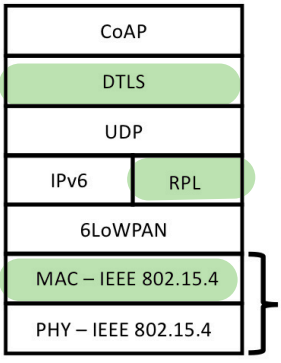
  Note: classical network defense (e.g. firewall) is <u>not</u> effective against this since worm spreads without using any classical infrastructure.

  Possible attacks: hardware bricking, 2.4 GHz jamming, ...

THE S IN IOT STANDS FOR SECURITY

- Communication technologies in IoT: lots of different standards and use cases in IoT. Best IoT stack selected based on use case.
  → Security Challenges in IoT:
  - Constrained capabilities and resources (limited power, memory, computation, communication range, ...)
  - Diverse communication technologies (lots of optimization per use case due to constraints)
    ↳ lack of interoperability, proprietary technology

- IETF open, standardized (not really) IoT stack



Constrained Application Protocol a.k.a HTTP for tiny devices (RFC 7252)

Datagram TLS (RFC 6347) (equivalent security to standard TLS)

RPL: IPv6 routing protocol for low power and lossy networks (RFC 6550)

Adaptation layer for IPv6 and UDP. (RFC 4944)

Also used by ZigBee, WirelessHART, MiWi, ...
802.15.4e implements frequency hopping

security in the IoT stack

RPL: AES encryption and authentication of routing control messages
MAC: AES encryption and authentication

- Encryption is not enough... : Even if everything is perfectly encrypted, there is still lots of information that can be gained from headers, timing information, etc.
  - WiFi, ZigBee, BLE all have unencrypted device identifiers (MAC address, short address, ...) → adversary sees which devices are communicating (e.g. MAC address contains manufacturer)
  - Existential Leakage: Occurs when the transmission of a single message implies a real-world event (even when encryption used)
    ↳ e.g. a door is locked using a wireless lock

    ⇒ Very difficult to hide. Eliminating this would require sending dummy traffic → often not possible, would quickly drain the battery of IoT devices.

    ⇒ One possibility: use stronger power device that sends with same MAC as real IoT device but uses wrong encryption key
    → receiver will discard msg, adversary can't tell the difference. Need to be careful that IoT devices won't read these msgs and drain battery.
    → Also: RF fingerprinting allows to distinguish devices

- **Statistical Leakage**: Occurs when deviation from normal implies real-world event.
  - ↳ e.g. company event leads to more activity of occupancy sensor in parking lot

  Statistical leakage can be eliminated by discretization of time into intervals and maintaining the same rate in each interval (possible but can be costly)

- Encryption and hiding transmission times is not enough...:
  Even perfect encryption and perfect hiding of transmission times still allows information leakage. 2 examples:

  - **Inference of speech in encrypted VoIP**: Phones/voice in VoIP are compressed using variable-bit-rate (VBR) codec → different compression based on what is currently spoken. High performance/low overhead encryption is applied → plaintext length = ciphertext length → **plaintext length leakage due to length-preserving stream cipher**

    ↳ VBR encoding leaks phoneme boundaries → using ML, one can segment stream into phoneme and then classify phonemes. Using word boundaries and language restrictions, one can then identify words via phonetic distance and score recovered sentences using METEOR score.

    Entire reconstruction process:
    1) Phoneme segmentation
    2) Phoneme classification
    3) Language model correction
    4) Word segmentation
    5) Word classification

  - **Side-channel leaks in web apps**: Traffic analysis (packet size, sending time,...) allows to infer much information about website visited → website fingerprinting.

    - Auto-suggestions leak character pressed: length of auto-suggestion depends on pressed characters. By observing # sent bytes, one can infer the searched for word.

    - Size of GIF image reveals investment allocation: Investment website displays allocations as pie chart. GIF uses run-length-encoding that stores line-by-line, thus different pie-chart GIFs produce slightly different GIF sizes. As found's value changes over time, allocation fluctuates. After observing for a few days, investment allocation can be derived.

- Device Pairing: important aspect of IoT, how do I setup a security association between two nodes to achieve encrypted and authenticated communication? → device pairing

Pairing = process of establishing a security association between two devices that share no prior knowledge
↳ Security association (e.g. shared key) is only known to the two devices.

↳ Pairing process needs to be resilient against MITM attack.

Secure device pairing with no infrastructure support (ad hoc) is challenging.
Pairing method shall minimize additional I/O hardware needed (especially important for resource-constraint devices) and shall avoid complex user interaction.
Need to consider security, usability and deployability simultaneously.

Two types of secure pairing:
- No OoB channel: don't use OoB channel, only the in-band channel
  ↳ Often relies on physical channel properties and use special encoding to detect MITM attacks.
- Out-of-band: use secondary channel (assumed to guarantee authenticity despite MITM)
  ↳ Many proposed OOB channels:
    - Infrared, point devices towards each other
    - Scan QR codes (seeing-is-believing)
    - Audio (Loud and clear)
    - Devices touching (Resurrecting duckling)
    - Compare two displayed codes
    - Input code from device A to device B

    } humanly perceivable

    ↳ User must actively verify whether both devices agree on the exchange information
    ↳ Challenge in humanly-perceivable OOB: untrained/unaware users don't know how to use or don't understand the importance of proper device pairing.

    - WiFi, Bluetooth, RFID, NFC, Body channel, shake both devices, ambient noise, ...

    } physically constrained

    ↳ Reduce or avoid user involvement. Assumes MITM resiliency due to inherent physical constraints.

Bluetooth Low Energy (BLE) pairing: 4 authentication methods

Legacy protocols, broken by passive adversary

- Just works: no authentication
- Passkey entry: display 6-digit PIN on device A, user types PIN into device B

- Numeric comparison: compare two 6-digit PIN codes
- Out-of-band: directly provided with shared secret, which is assumed to be established via OOB channel.

⇒ If no eavesdropper is present, all methods are secure.

Appendix

- Sticky bit:
  - Sticky bit on directory: Files in that directory can only be renamed or deleted by the file owner, the directory owner or root.
  - Sticky bit on files: no effect

- SUID, SGID: x-bit of user octal (rwx) and group octal (rwx) can be an s. If SUID/SGID is set on a file, this file when executed will run with the permission of the owner / group of the file.
  - lowercase s: suid/sgid set and user/group executable bit set
    uppercase s: suid/sgid set and user/group executable bit not set