Security

At least three different categories:

- *Data integrity*: Backup etc., taken care by normal OS functions
- *Protection against user errors*: done by separating users and processes
- *Protection against malicious users*: more complicated, involves trade-offs between ease of use and level of protection

Major problem: *Identification of users*

Most common scheme: Passwords

+ : Easy to use and understand

- : All too often too easily guessable

- : Exposure problem

- : Where to store passwords

Protection mechanisms

Access to shared resources must be controlled

Two aims:
- Protection against users' mistakes
- Increase in reliability

Principles:

- separate policy (*what*) from mechanism (*how*)
  Important for flexibility
- Users should have as much privilege as necessary to get job done

Standard way: each user separate domain of protection
Need *trusted way* of making system privileges available
Disadvantage: primary target for breakins (`setuid`-programs in UNIX)

Access matrix

abstract view of protections
have row for each domain and column for object
Entry indicates access right
Example:

	F_1	F_2	F_3	Printer
D_1	read		write	
D_2				print

Have capabilities like read, write, copy, owner, control
Unsuitable for implementation because matrix far too large

Implementation Issues

- Access lists for objects (store columns)
- Capability lists for domains (store rows)

Capabilities allow great flexibility
Example: Hydra

- Auxiliary rights: each process can pass access to procedure to other processes
  $\Rightarrow$ dynamic change of access rights
- Rights amplifications: procedure can act on specified type on behalf of any process which is allowed to execute it
  Rights *cannot* be passed on
  $\Rightarrow$ flexible and more secure mechanism for granting higher privileges temporarily

UNIX access rights

coarser than access lists
for each file, have three categories of possible users

- owner
- group (pre-defined set of users)
- all others

owner can grant permission to

- read
- write
- execute program/find files in directory

- OS is large pice of code (Linux ca. 15M LoC)
- security holes give full control to the attacker
- verification of such large programs is infeasible

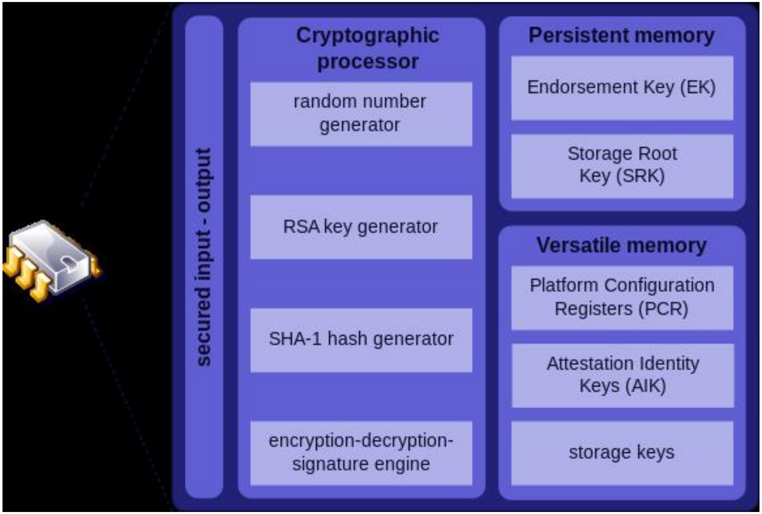
$\Rightarrow$ OS not suitable as part of *trusted computing basis* (the code which must be assumed to be secure in order to guarantee security)

Active area of research: remove OS from trusted computing basis
need to introduce additional HW as anchor for security
Example: Trusted Platform module

Trusted Platform Module

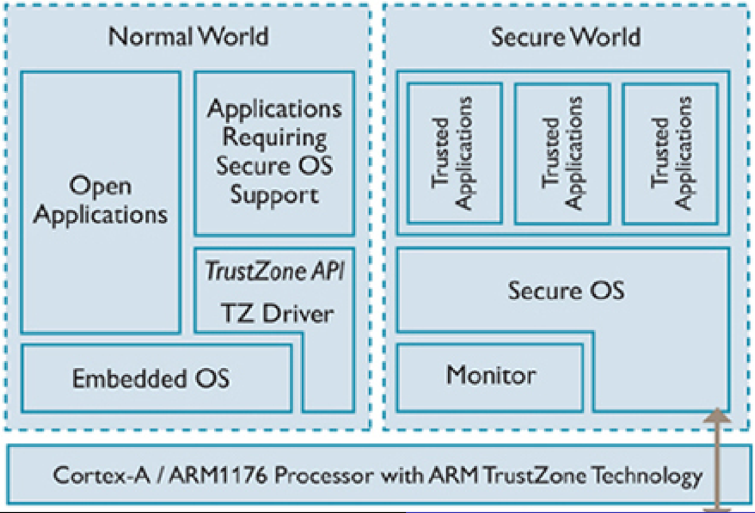
- present in a most modern laptops/desktops except Apple
- provides cryptographic operations which make it possible to
  - certify SW and HW configuration to third parties
  - provide secure boot (all files used for botting have not been corrupted)
  - secure storage
    allow access to data only if system is in known state
    achieved by linking encryption keys to system state

TPM


secured input - output
Cryptographic processor
random number generator
RSA key generator
SHA-1 hash generator
encryption-decryption-signature engine
Persistent memory
Endorsement Key (EK)
Storage Root Key (SRK)
Versatile memory
Platform Configuration Registers (PCR)
Attestation Identity Keys (AIK)
storage keys

Other approach: TrustZone
present in almost all mobile phones and tablets
have in aaddition to normal processor secure co-processor with corresponding HW
Important use case in Android: have private/public key pair where private key is stored within secure memory and not accessible directly
have operations to sign and decrypt using the private key
use public key to encrypt keys held in keystore.
$\Rightarrow$ prevents offline guessing attacks

TrustZone


Normal World
Open Applications
Applications Requiring Secure OS Support
TrustZone API
TZ Driver
Embedded OS
Secure World
Trusted Applications
Trusted Applications
Trusted Applications
Secure OS
Monitor
Cortex-A / ARM1176 Processor with ARM TrustZone Technology