# TRUSTED COMPUTING

Lect 8

# Contents

- **RoTs in SW**
  - Unclonable Identity in HW - PUFs
  - OS: rings
  - Trusted Computing Base
  - Isolation, different models
  - Virtualization: different approaches
  - Sandboxing
  - SELinux
  - Linux Hardening
  - Containers

**Little about**

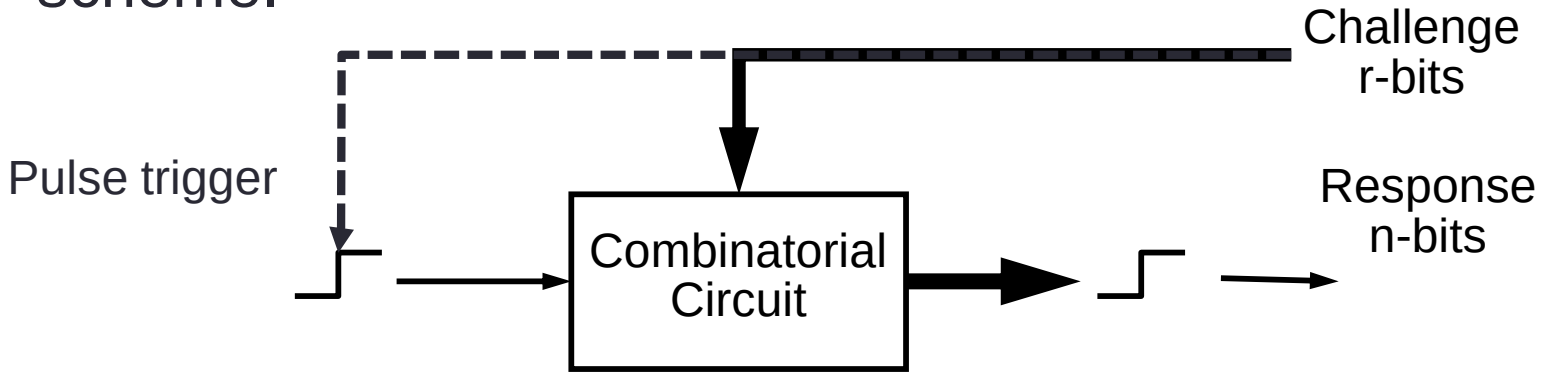
- Android
- IoS
- BYOD

# PHYSICAL UNCLONABLE FUNCTIONS (PUFs)

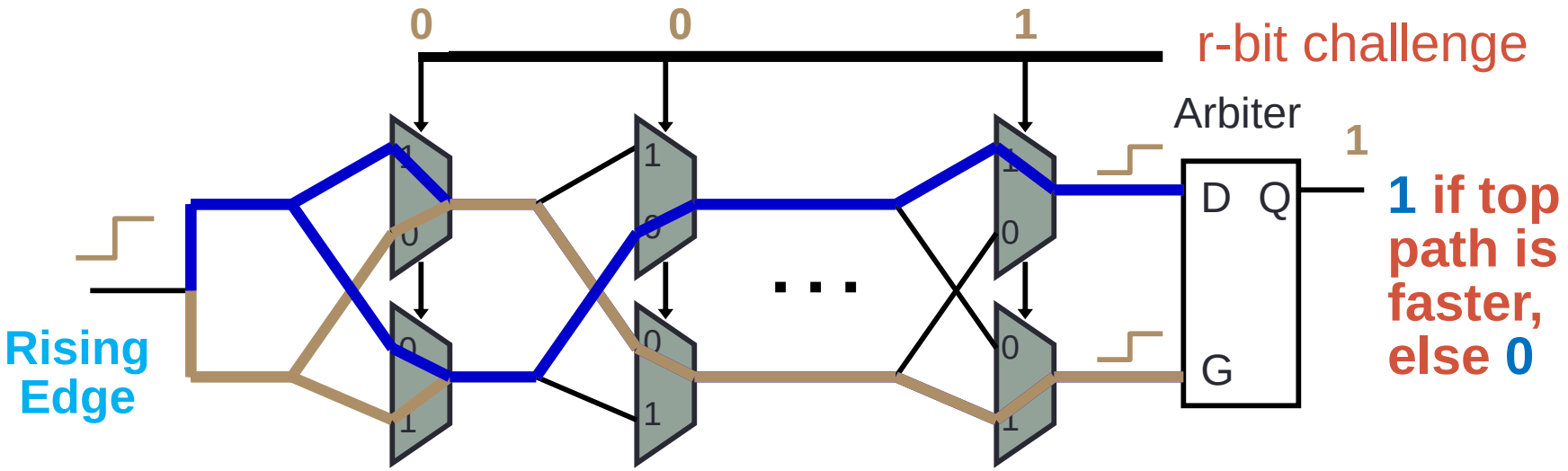
# Silicon PUF – Concept

- Because of random process variations during ASIC manufacturing, no two Integrated Circuits even with the same layouts are identical
  - Variation is inherent in fabrication process
  - Hard to remove or predict

- Practice shows that variation is detectable which allows us to use it for identification in a challenge-response scheme.



# Example: silicon PUF using MUXs




Each challenge creates two paths through the circuit that are excited simultaneously.

The digital response of 0 or 1 is based on a comparison of the path delays by the arbiter

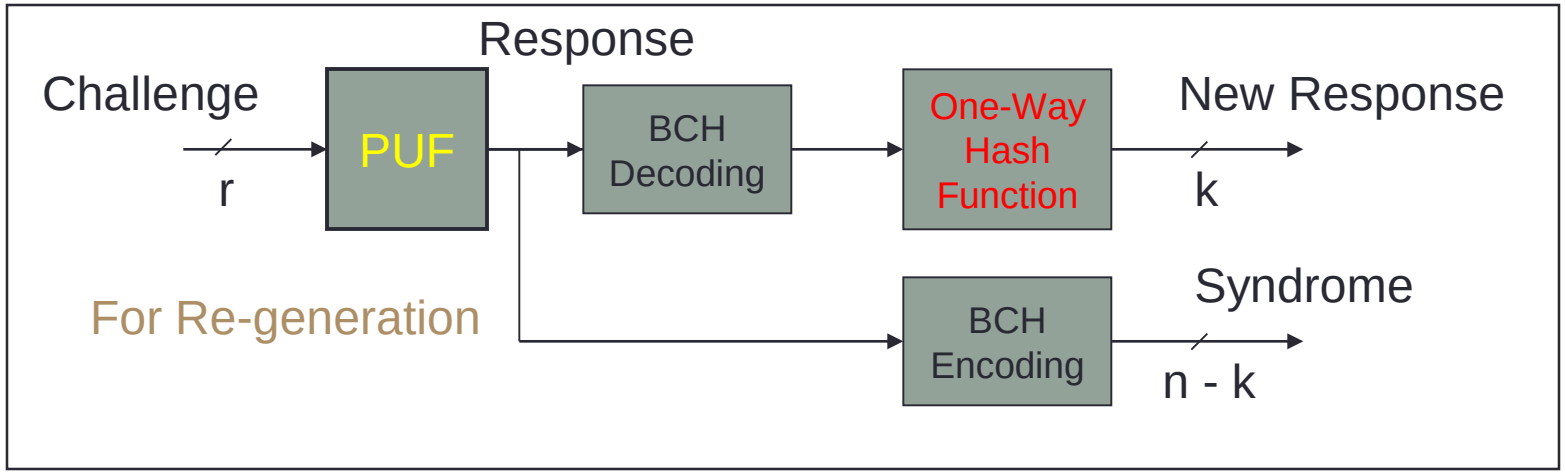
Only uses standard digital logic!

# Problems

- Stability
  - Over time when device ages
  - Temporal conditions: temperature, radiation, photons
- Attacks
  - Modelling of the device by observation of challenge-response pair
  - Side-channel attack
  - Attacks using fault insertion
- Conclusion: nice idea but still not without problems

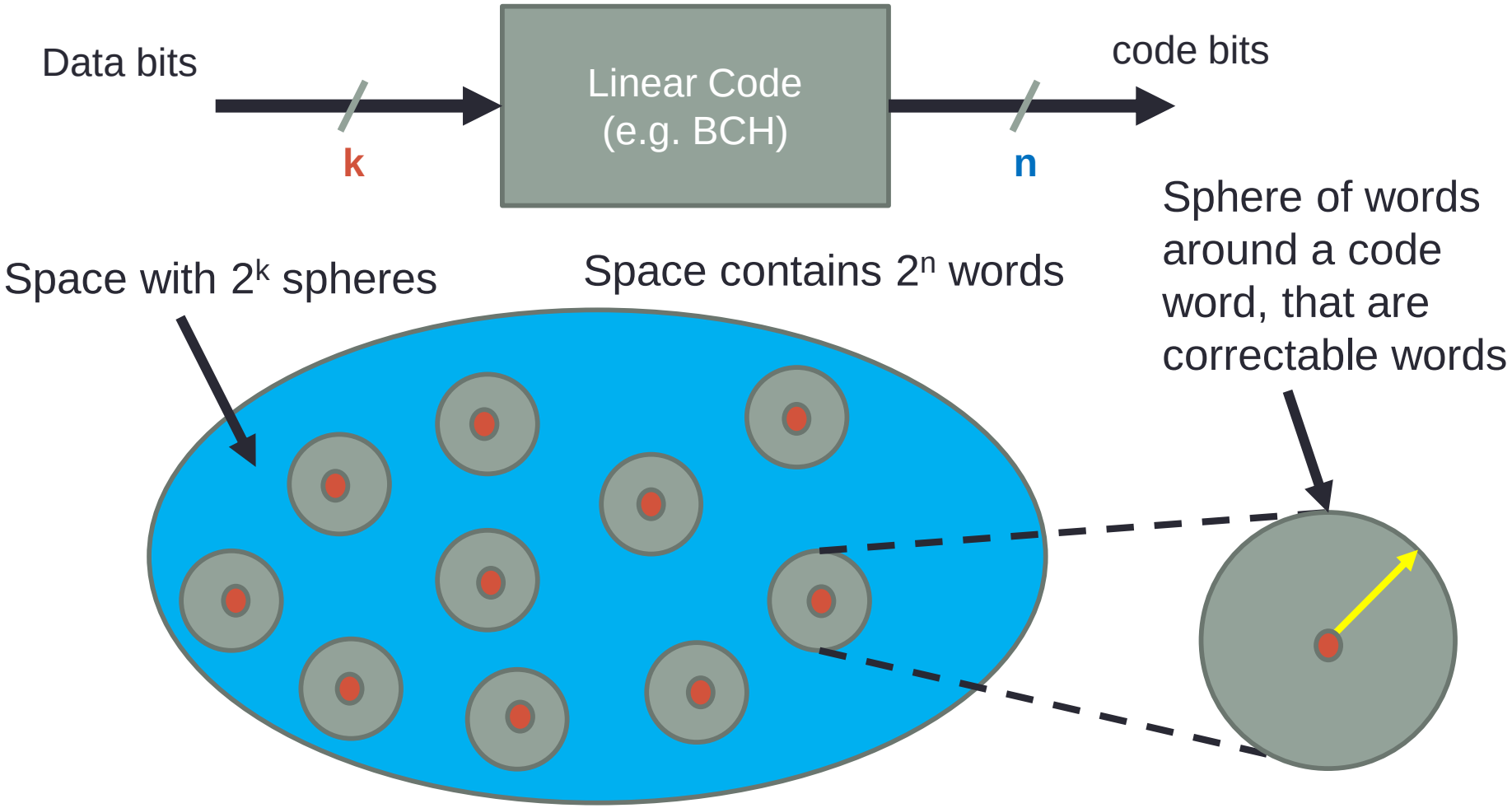
# PUF – solving some issues

Increase security and reliability by adding extra control logic




- Hash function (SHA256) precludes PUF "model-building" attacks since, to obtain PUF output, adversary has to invert a one-way function
- Error Correcting Code (ECC) can eliminate the measurement noise

# Coding theory – 1 slider



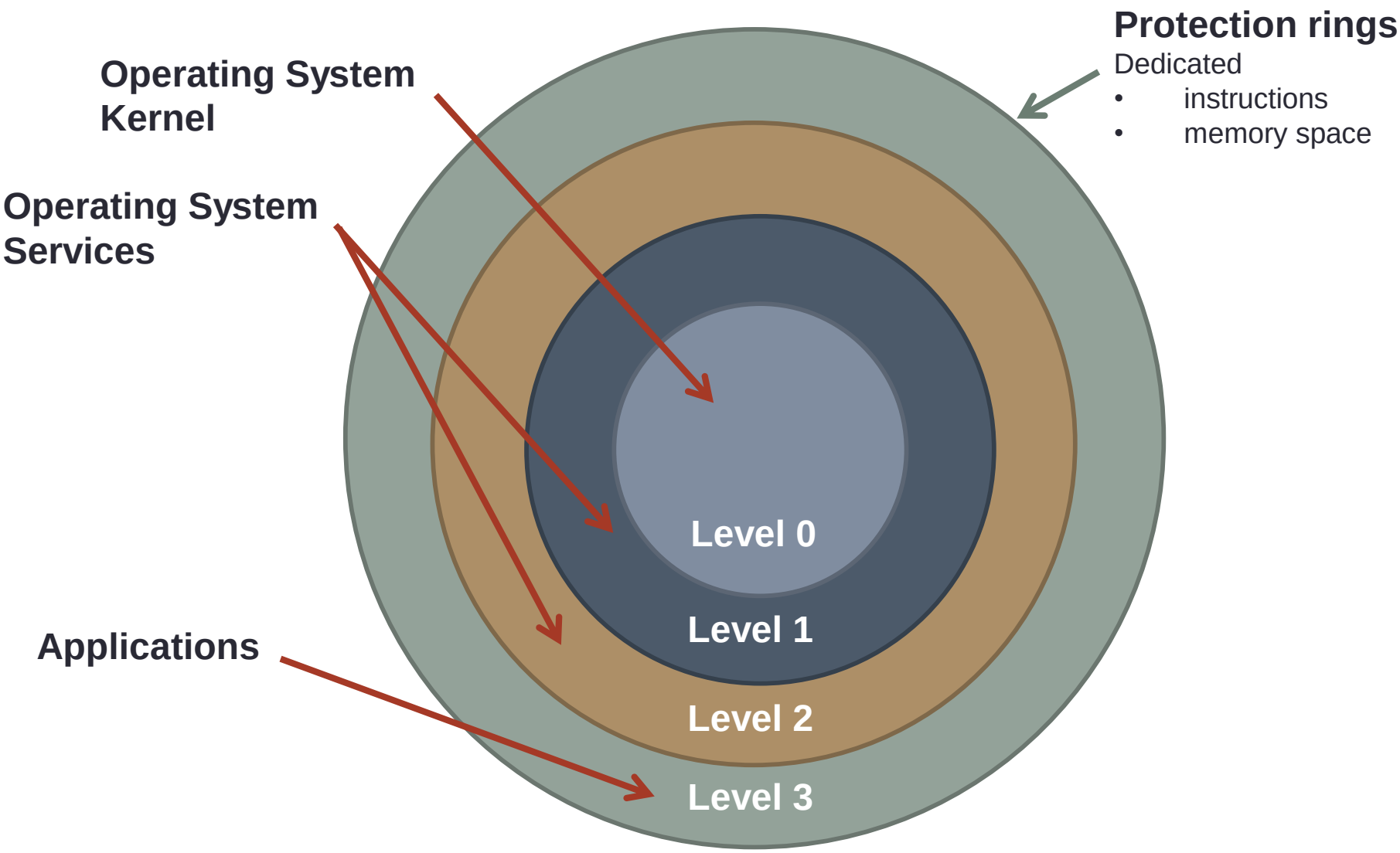
# RoT in Software

- Possible but we have limitations
  - owner of the device on which software runs should not be an attacker (he/she and the device ”work together”/”have the same interests”)
  - Does not work when the device in the ”enemy’s territory”

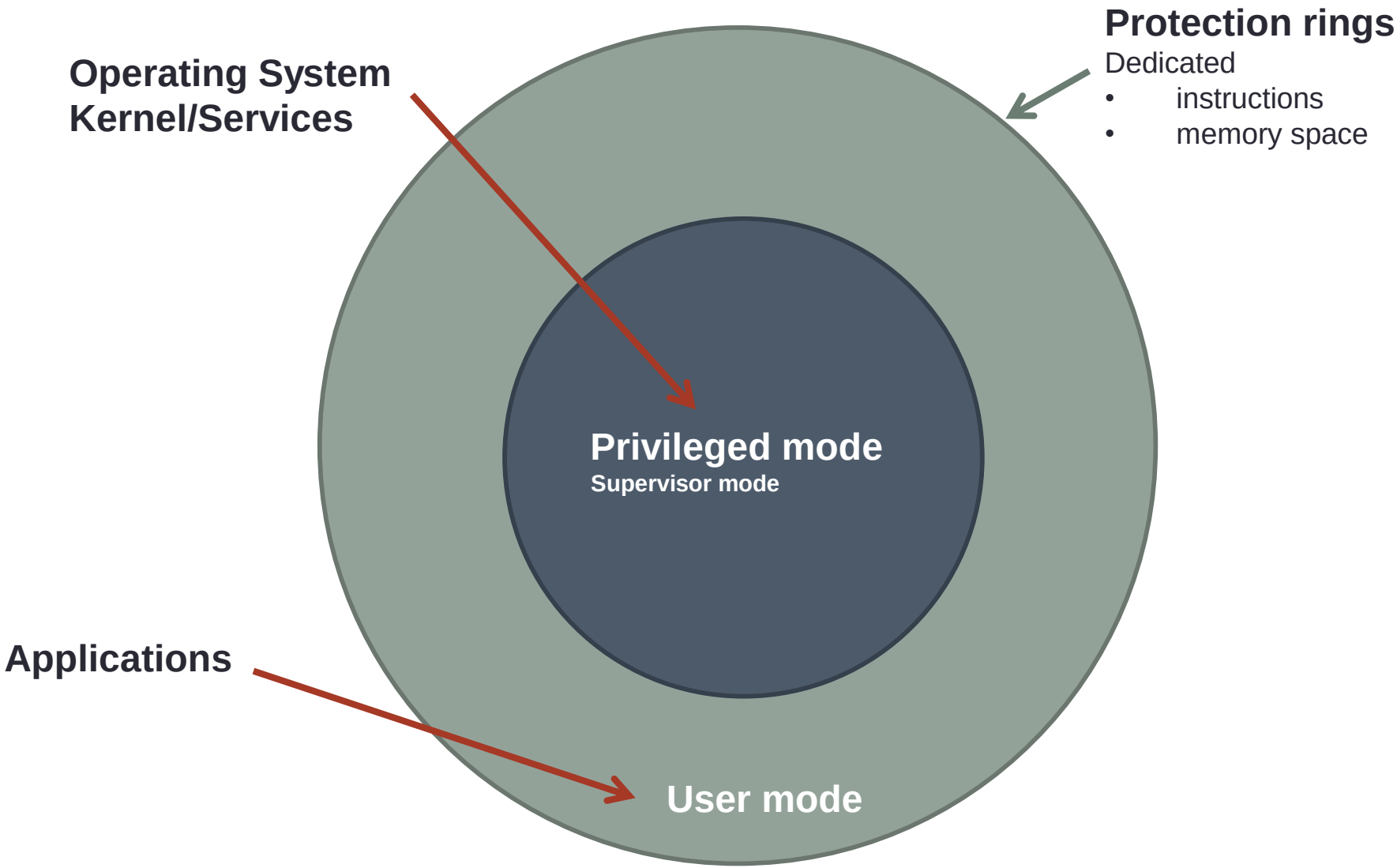
# OSes

- OSes come in many different versions; Linux(es), Windows X=7,8,(9), OS X, AIX, etc
- Most simple OSes have no means to securely enforce (multi-user) access control, FemtoOS, TinyOS, etc
- Even if so
  - Correct configuration of full-blown OS is not trivial
  - Without protection user can attack system from ”below”

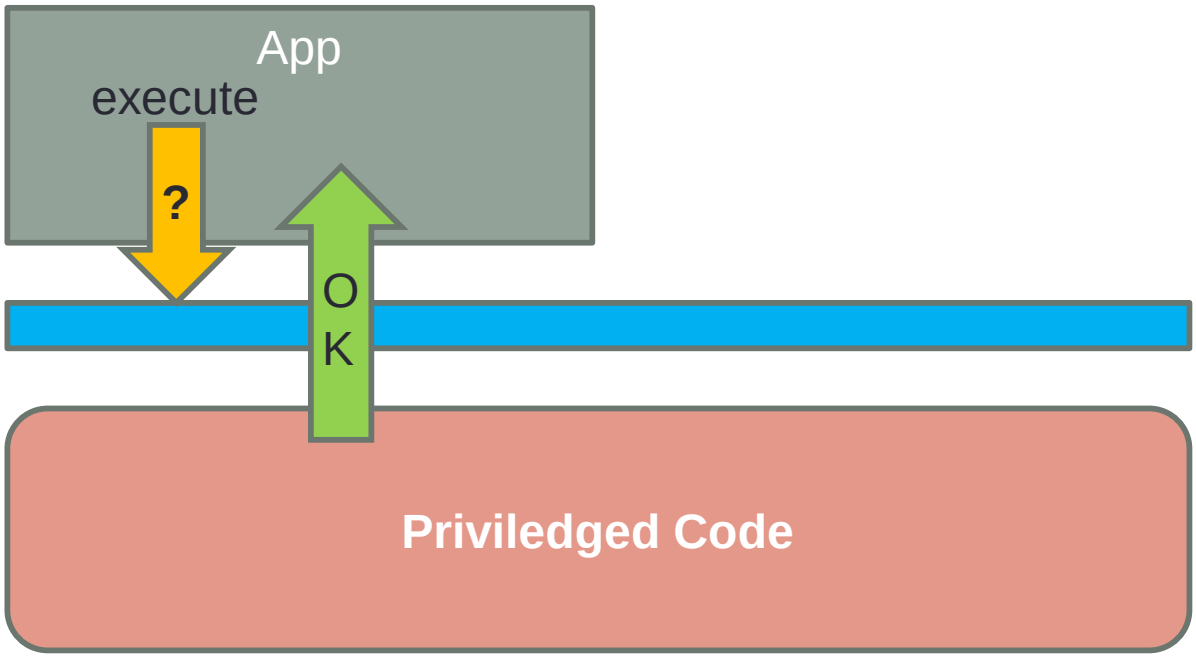
# Recall security ring architecture



# ARM standard approach

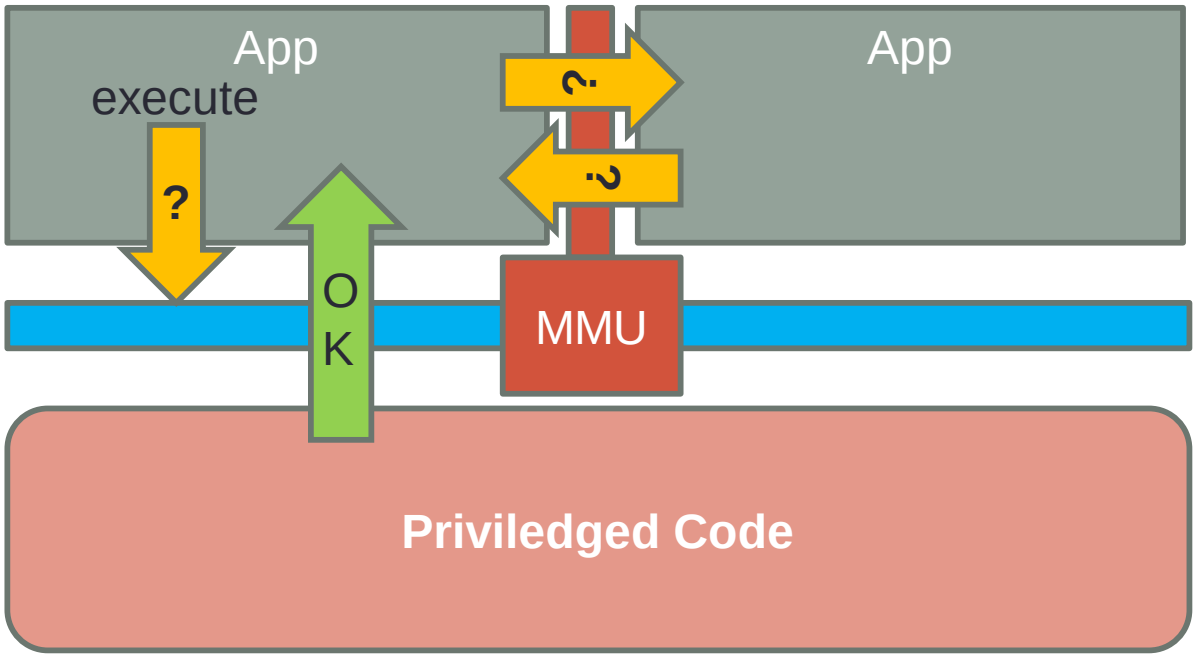


# Protected Mode (rings) protects




Protect OS from app

# Protected Mode (rings) protects

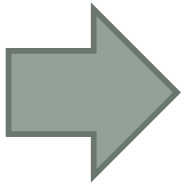



1 Protect OS from App

2 Protect from other Apps

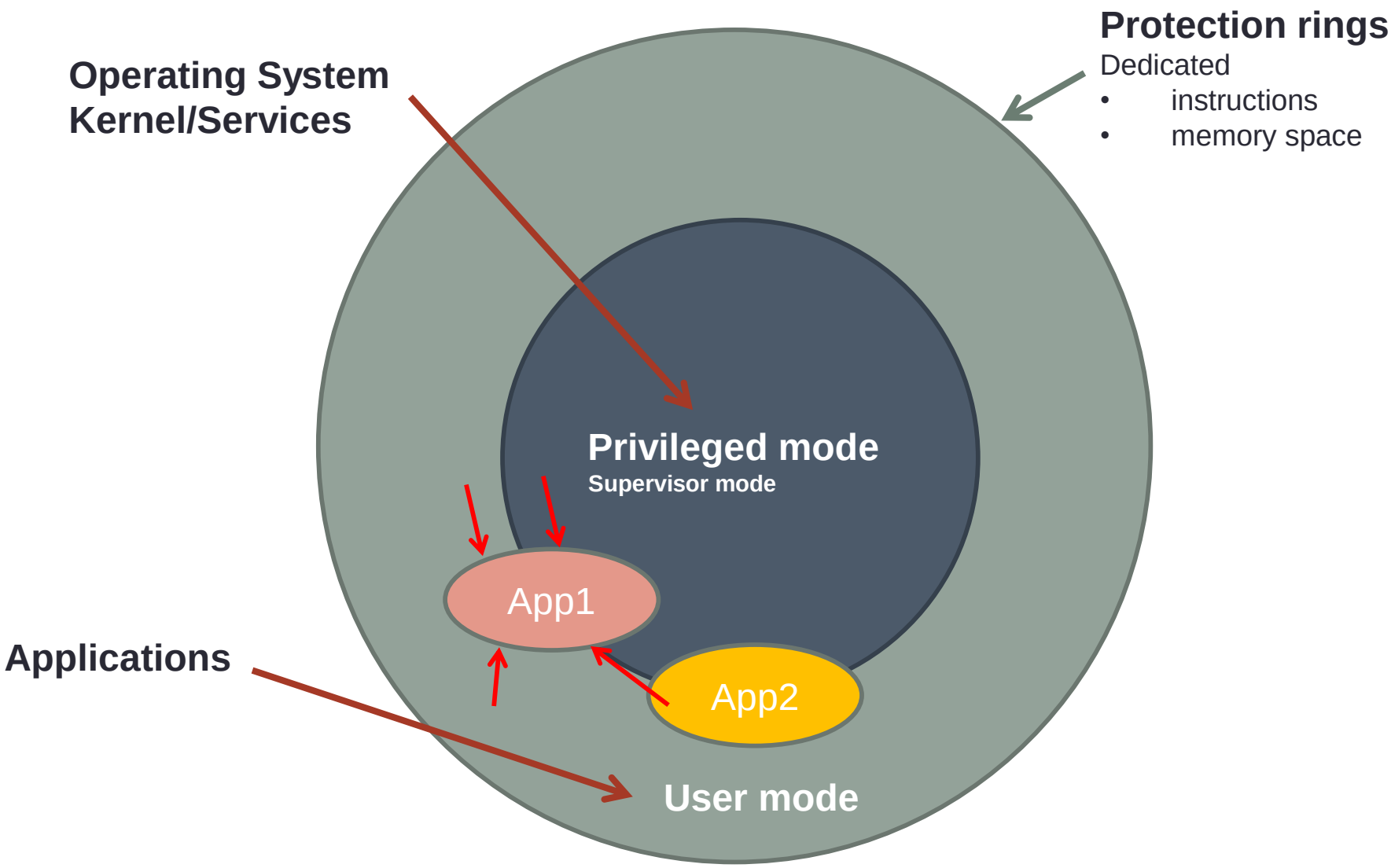
# Separation of sensitive ops and data

- Since too much code in running in user space and even in the privileged space:



Sensitive applications and data cannot be given good guarantees that other running code cannot tamper or get access.

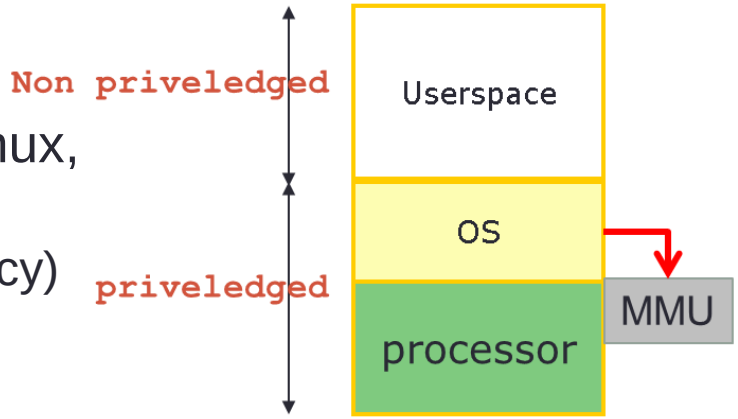
# Security problem for applications



# OS for trusted computing

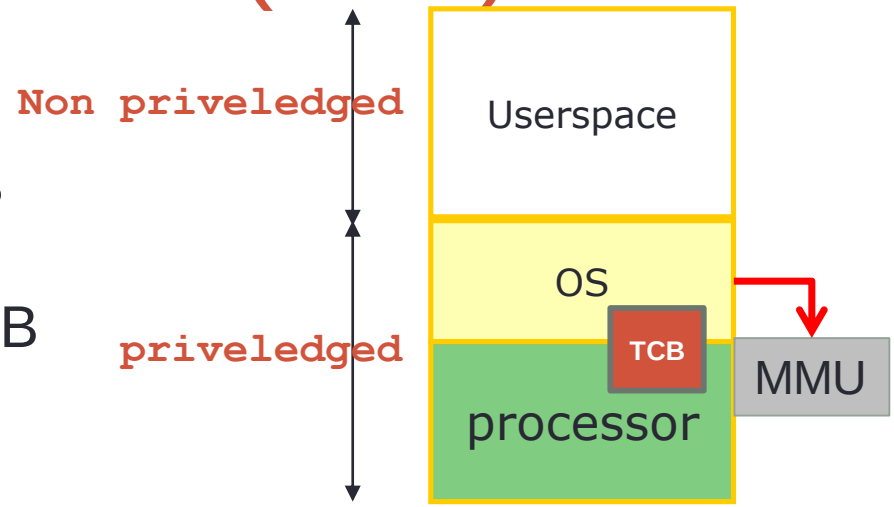
- **Primary security objective of OS in a trusted platform = secure isolation**
  - Mainstream OS (e.g. Windows, Linux)
    - Large and complex code base
    - Optimized for ease of use, performance and reliability
  - Trusted OS (e.g. Trusted Solaris, SE-Linux, SPEF)
    - Not user friendly (because of security policy)

Hypervisor (and not the OS) controles the MMU



# Trusted Computing Base (TCB)

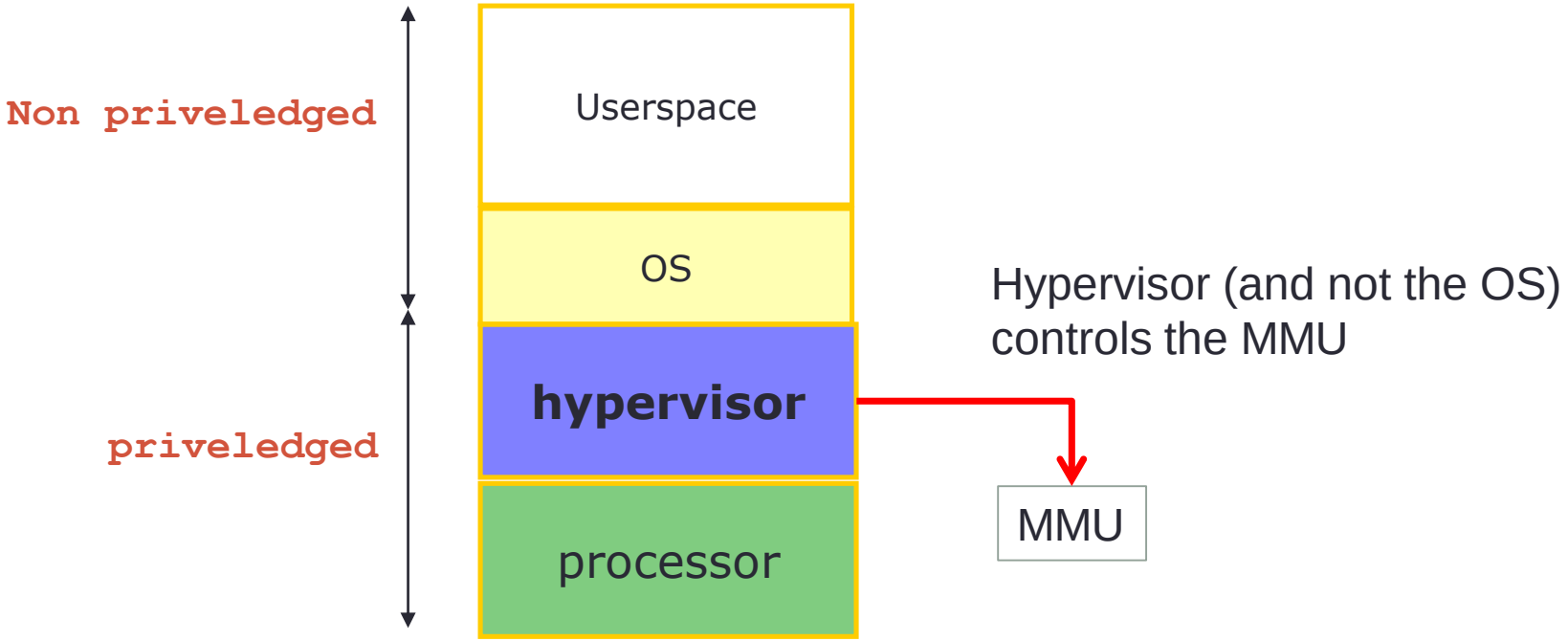
- In stead of having to trust the complete OS we organize the OS that our trust depends only on a (much) smaller part called the TCB



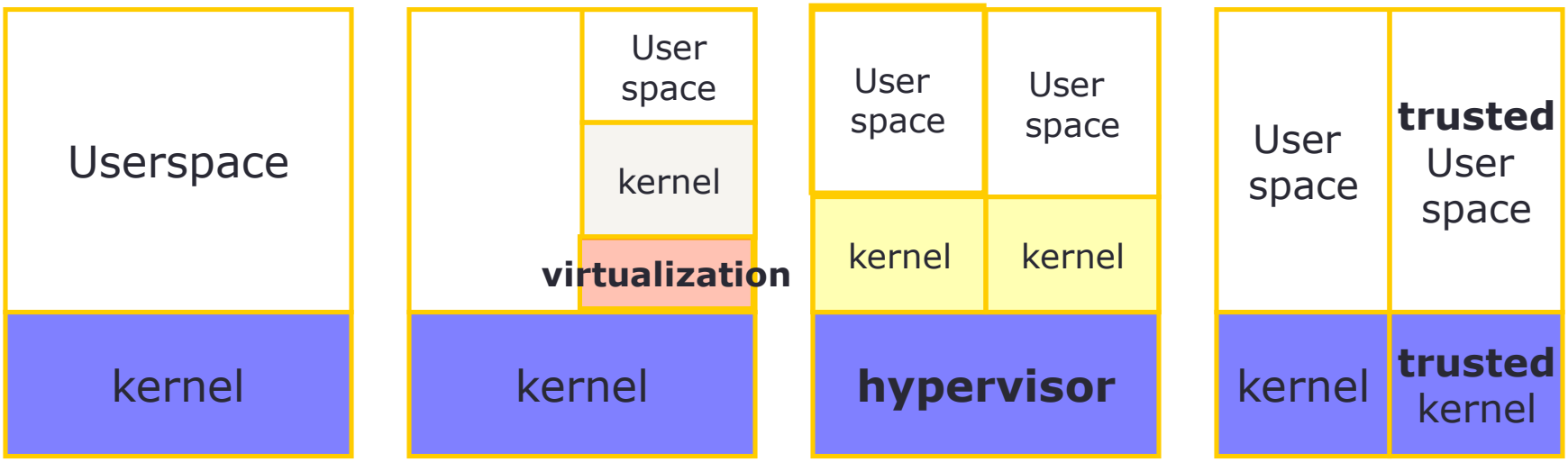

- The TCB consists of the mechanisms (from hardware, firmware, and software) that together are responsible for enforcing a computer security policy.
- **Recall that this indicates that the TCB has RTM capabilities.**

# Hypervisors and micro Kernels

- Execute in priveledge mode
- Schedule the systems(OSs) that execute on it



# Execution environment setups for a trusted platform




Normal OS (Windows, Linux SE Linux)

Virtual Machine (VMWare, Java VM)

Hypervisor

CPU with trusted mode (e.g.TrustZone)

Material on TCG is based on slide material from Dries Schellekens
http://www.esat.kuleuven.be/cosic/seminars/slides/Trusted_Platforms.ppt

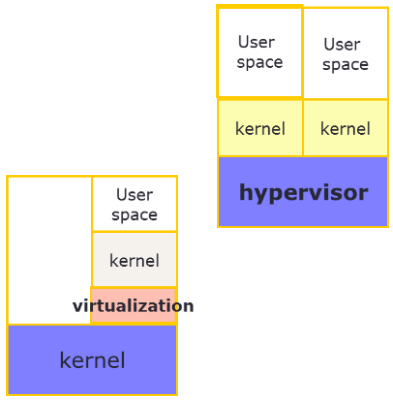
# Virtualization

- Abstraction of computer resources
- Pioneered by IBM to keep using legacy system solutions on new hardware without rewriting code
- Turned up to have stability and security benefits (isolation)
  (at the expense of performance)
- There are many ways to do this and there exist therefore many different types of approaches to virtualization

# 2 Virtualization approaches

**Type 1 and Type 2 virtualization**

- Type 1: runs on ”bare metal”
- Type 2: runs on host




**Full/Pure vs impure/para virtualization**

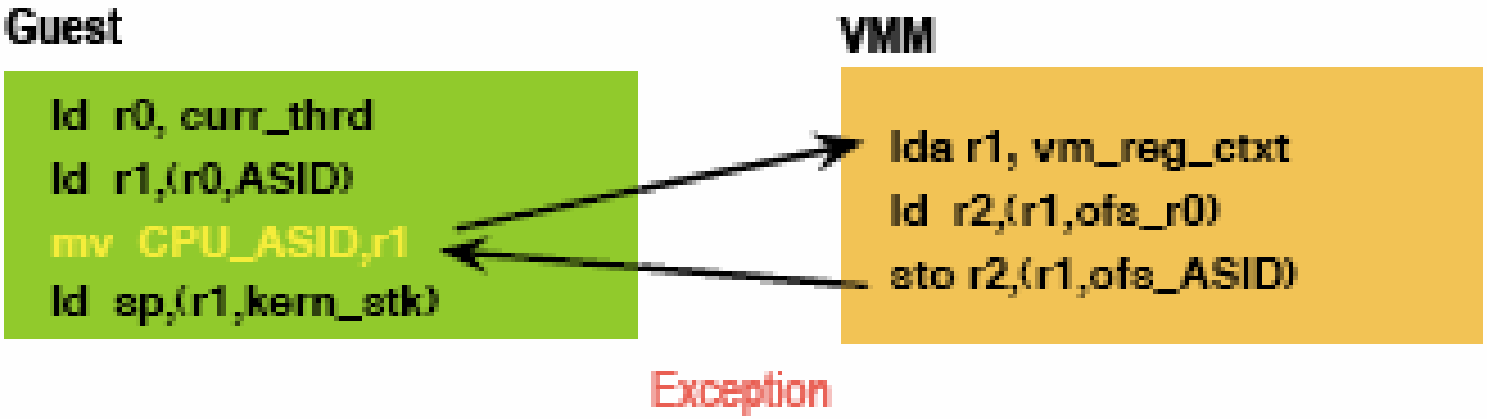
- Full/pure virtualization: ensure that sensitive instructions are not executable within the virtual machine, but instead invoke the hypervisor: **needs hardware support**
- Impure virtualization: remove sensitive instructions from the virtual machine and replace them with virtualization code.

# Example of a trusted microkernel: L4

- Used in various Qualcomm chipset phones, available for Linux and Symbian
- Academic work ongoing
- Free to use
- Commercial: OpenKernel Labs

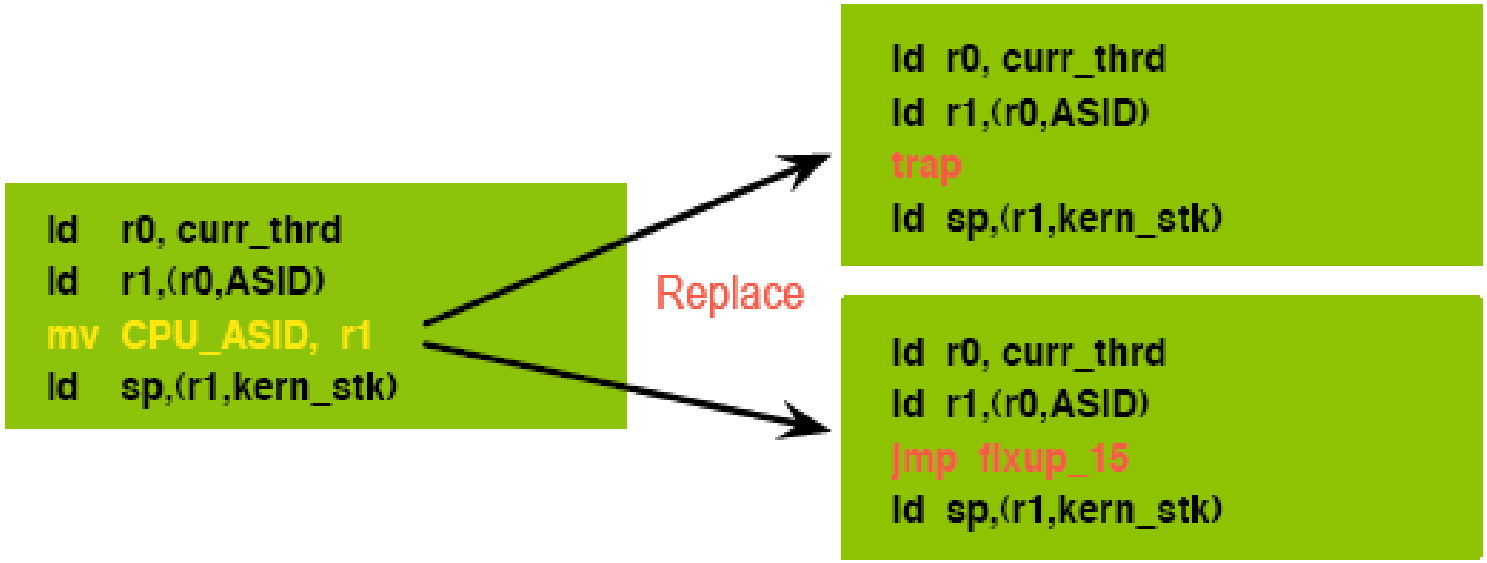
# Pure Virtualization

- Most instructions are executed directly on the hardware.
- All senstive instructions are priveledged. They are trapped and instead executed by the hypervisor that runs in priveledged space (kernel mode, superuser mode, etc).
- Needs hardware support (Modern main CPUs have this)



# Impure Virtualization

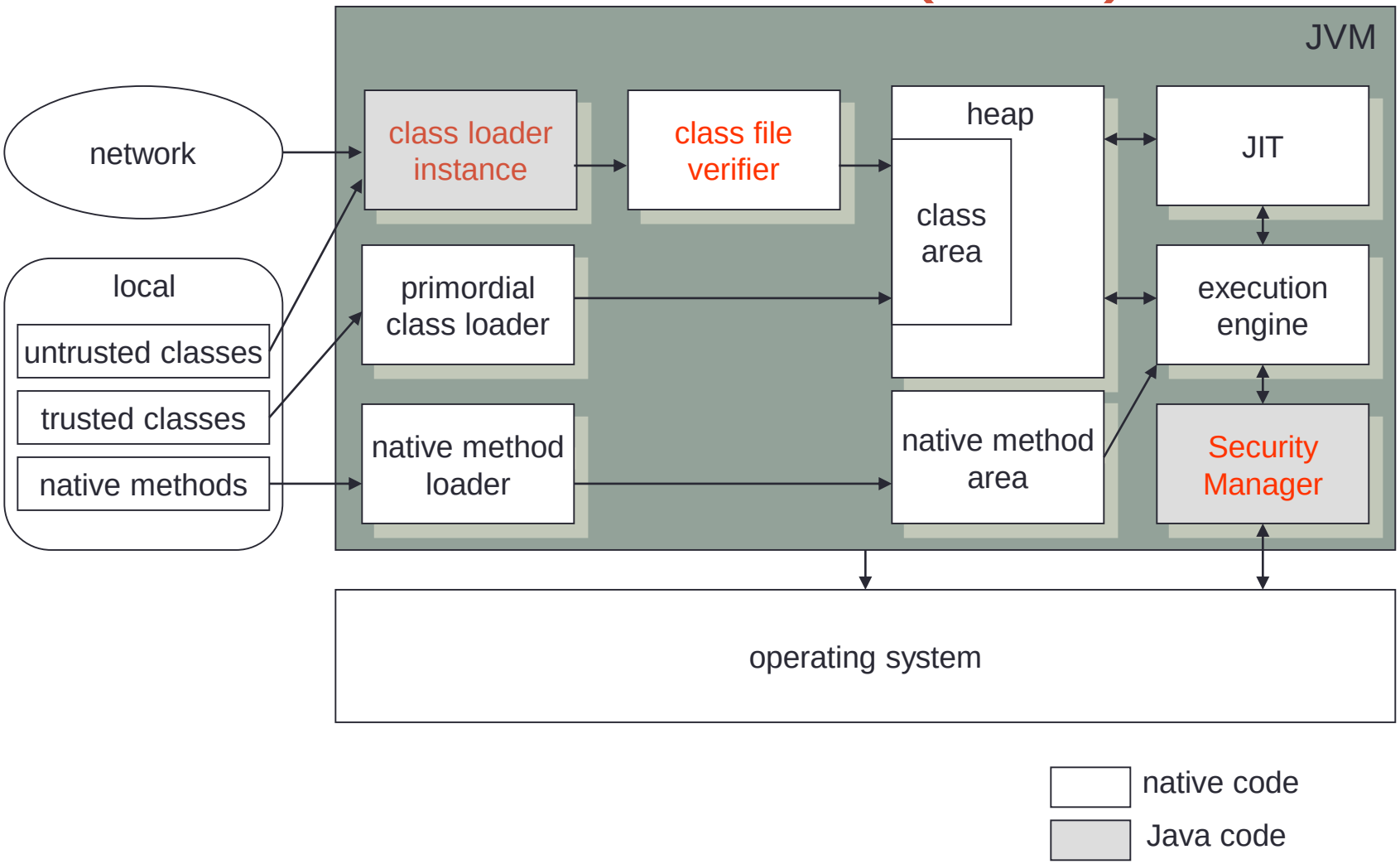
- Most instructions are executed directly on the hardware.
- All senstive instructions rewritten (e.g. during load time or during porting (para virtualization)): either trap to hypervisor or jump to a user-level emulation code



# JAVA
## AS TRUSTED EXECUTION ENVIRONMENT

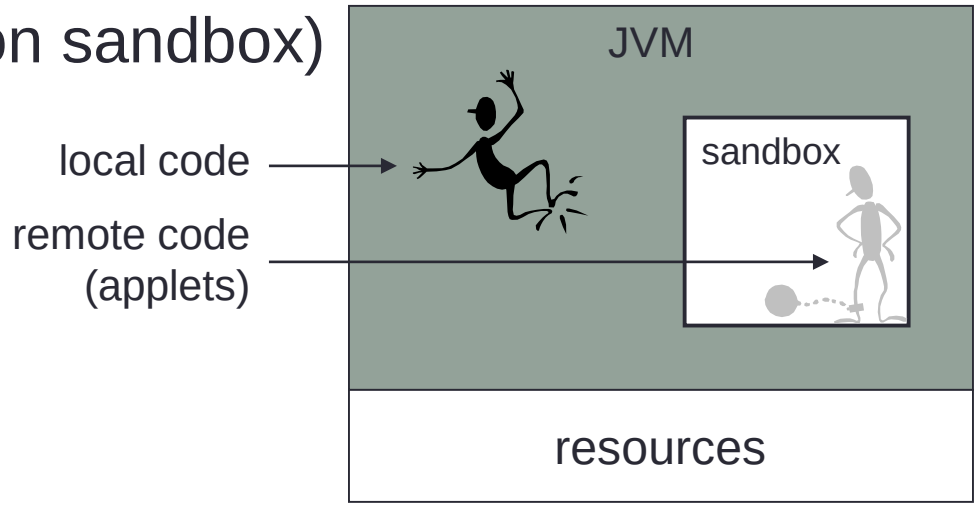
Trusted systems in SW

# The Java Virtual Machine (JVM)



# The sandbox

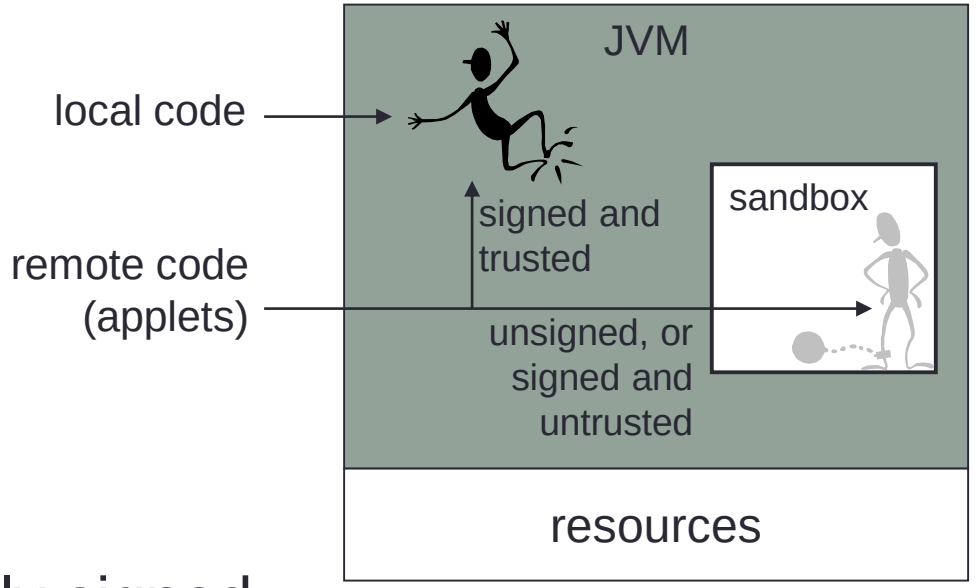
idea: limit the resources that can be accessed by applets (this creates an execution sandbox)




- introduced in Java 1.0
- local code had unrestricted access to resources
- downloaded code (applet) was restricted to the sandbox
  - cannot access the local file system
  - cannot access system resources,
  - can establish a network connection only with its originating web server

# The concept of trusted code

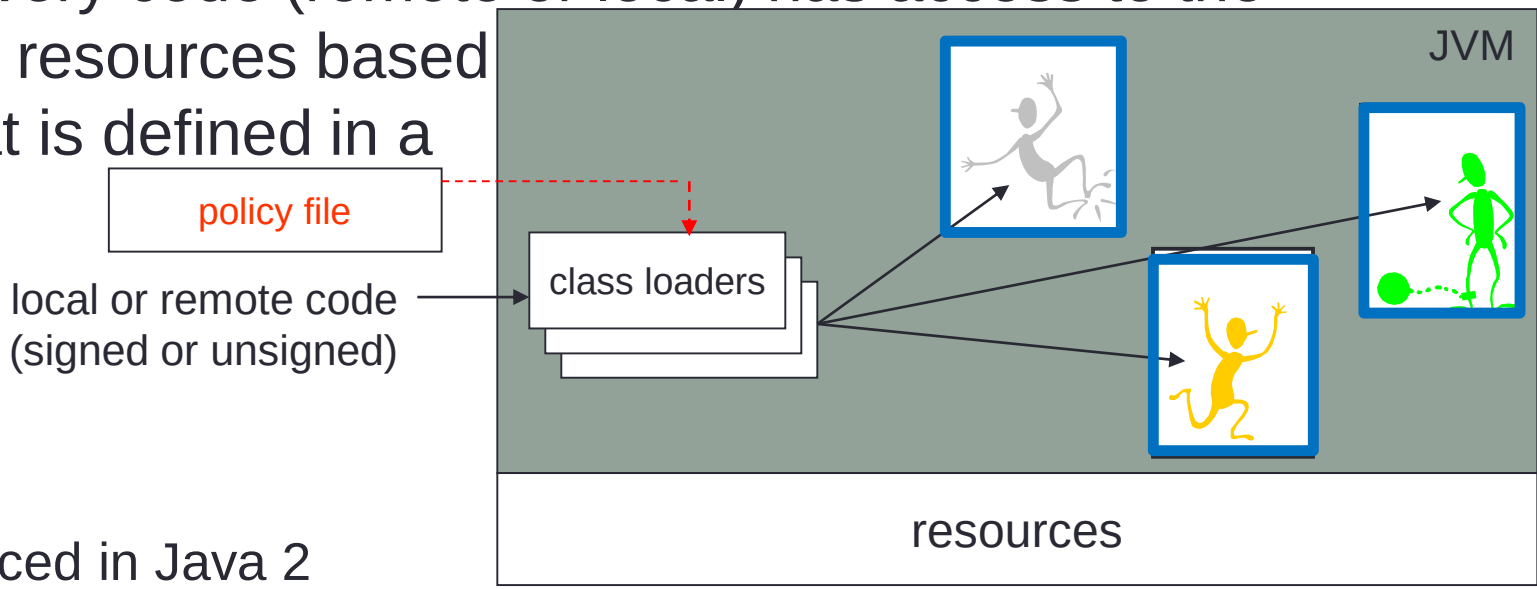
idea: applets that originate from a trusted source could be trusted




- introduced in Java 1.1
- applets could be digitally signed
- unsigned applets and applets signed by an untrusted principal were restricted to the sandbox
- local applications and applets signed by a trusted principal had unrestricted access to resources

# Fine grained access control

idea: every code (remote or local) has access to the system resources based on what is defined in a




- introduced in Java 2
- a *protection domain* is an association of a code source and the permissions granted
- the code source consists of a URL and an optional signature
- permissions granted to a code source are specified in the policy file

```
grant CodeBase “http://java.sun.com”, SignedBy “Sun” {
   permission java.io.FilePermission “${user.home}${/}*”, “read, write”;
   permission java.net.SocketPermission “localhost:1024-”, “listen”;};
```

# Java's impact

- The Java system has been an example for many other languages and execution environments.

  E.g.

  - ActiveX
  - .Net
  - STIP
  - Dalvik (Android)
  - iOS
  - Linux Containers

# SELINUX

Trusted systems in SW

# Example of a trusted OS: SELinux

Motivation

- Discretionary Access Control (DAC) in Linux provides not enough choices for controlling objects. Root is too powerful and considerable risk of “permission escalation attack”

- Mandatory Access Control (MAC) allows you to define permissions for how all processes (called *subjects*) interact with other parts of the system such as files, devices, sockets, ports, and other processes (called *objects* in SELinux).

# Security goal for SELinux

- General: Minimize the effect of malicious code and exploits
  - ”Sandboxing” programs and data
  - control access rights, minimize the privileges
  - Policy driven so root user is constraint, so MAC instead of DAC

MAC = Mandatory Access Control,
DAC = Discretionary Access control

# SELinux Concepts

# Type Enforcement & Domain Transition

- Type
  A type is assigned to an object and determines who gets to access that object.
- Type Enforcement
  when a object is accessed

- Domain
  Domain defines what a process can do.
- Domain Transition
  when a process invokes another process

Note: Actually ’domain’ is a type (or a process)

# SELinux

- In SELinux, rights of a process depend on its security context. The context is defined by the identity of the user who started the process, the role and the domain that the user carried at that time.
- 
- The rights really depend on the domain, but the transitions between domains are controlled by the roles. Finally, the possible transitions between roles depend on the identity.

# Multi-Level Security (MLS)

- MLS portion of Security Context is composed of 4 parts
  - Low/High
  - Sensitivity/Category

MLS

- Includes syntax to define dominance of security levels
- Subjects with range of levels considered trusted subjects

- Implements a variation of Bell-La Padula
  (remember your basic computer security or OS course)

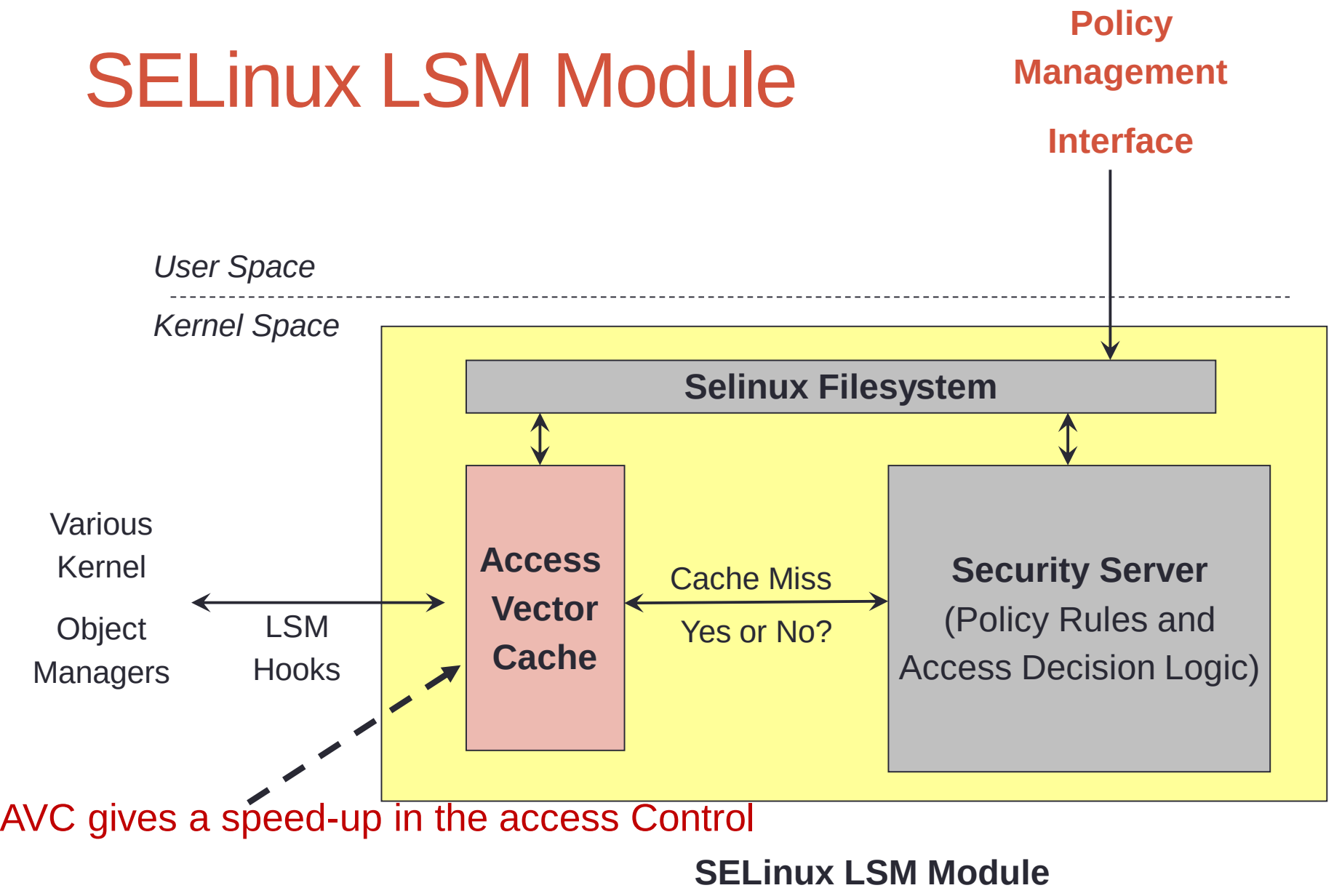
# Architecture

# LSM (Linux Security Module)

- Kernel framework for security modules
- Provides a set of hooks to implement further security checks
- **Usually placed after existing DAC checks and before resource access**
- Implications?
  - SELinux check is not called if the DAC fails
  - Makes auditing difficult at times.(a access violation can have its reason in DAC and in MAC)

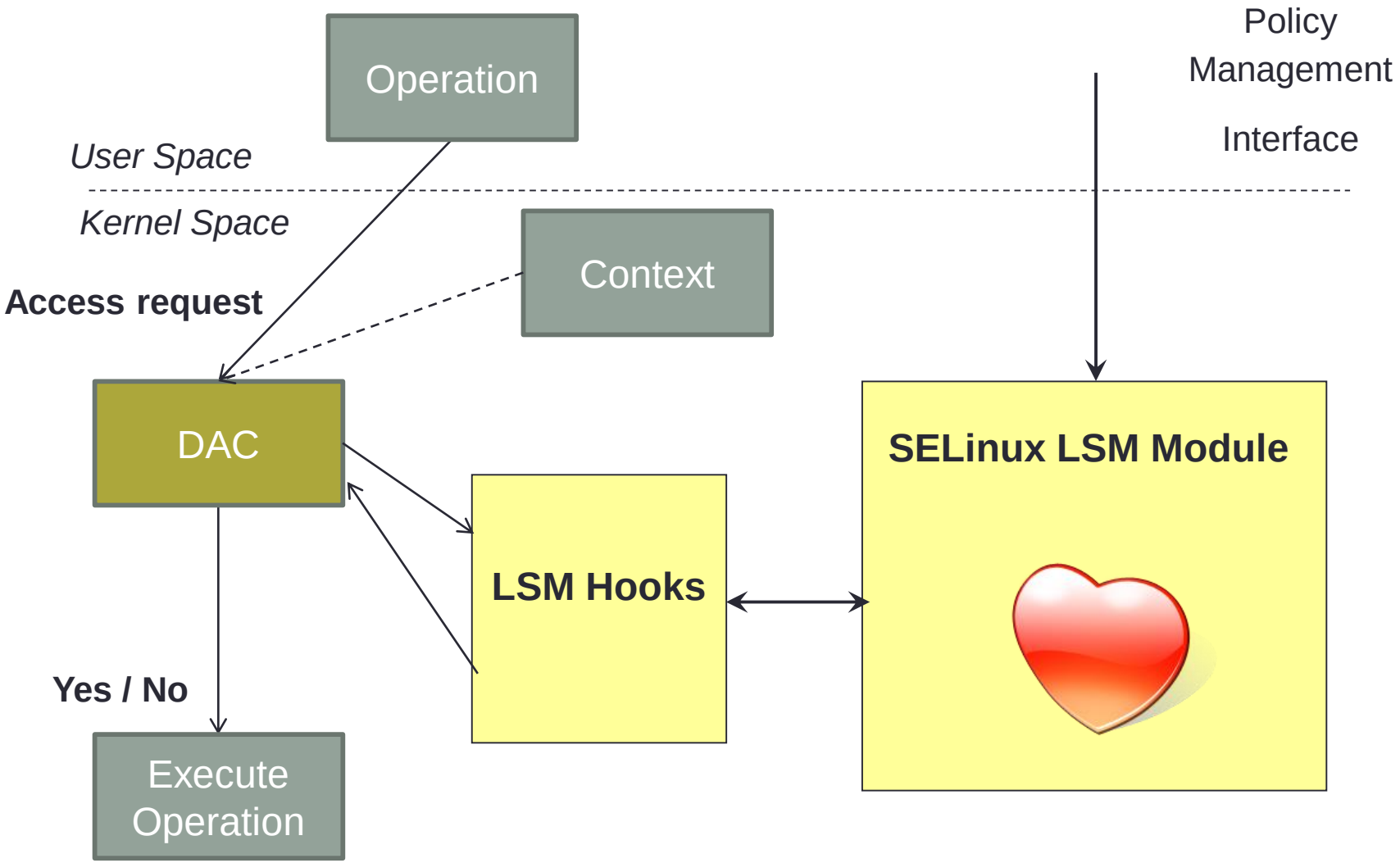
- The LSM is also found in other Linux OS, e.g. Android,Ubuntu,RedHat,…

# SELinux LSM Module

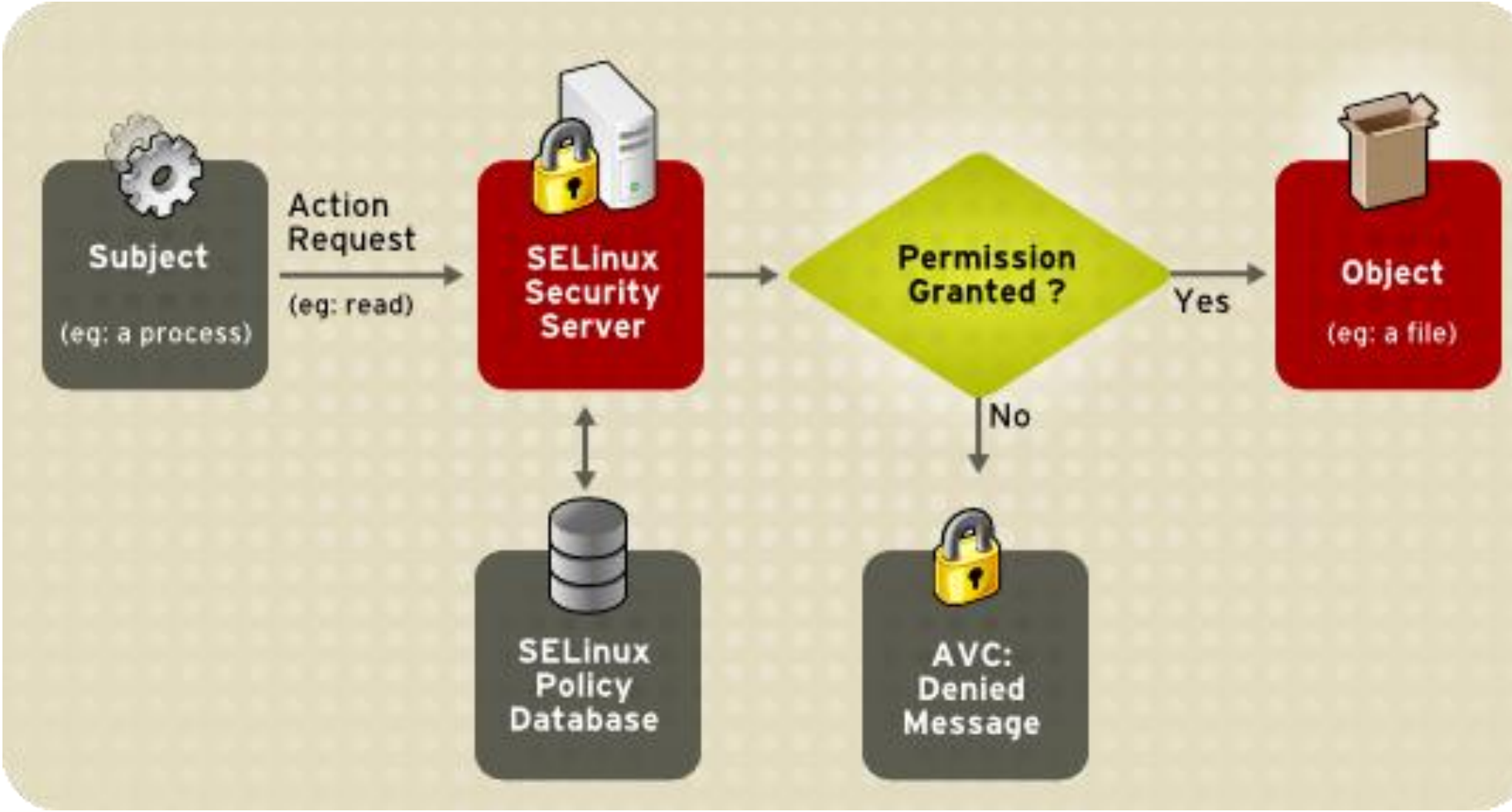



**SELinux LSM Module**

# SELinux basic architecture setup



# In Centos




Source Centos

# SELinux Policy Language

# Reference Policies

- Maintained by NSA and FC Mailing Lists

- Compiles into three versions
  - **Strict:** every subject and object exists in a specific security domain, and all interactions and transitions are individually considered within the policy rules.
  - **Targeted:** every subject and object runs in the unconfined_t domain except for the specific targeted daemons. Objects that are in the unconfined_t domain have no restrictions and fall back to using standard DAC Linux security.
  - **MLS:** Multi-Level-Security

# SELinux modes

In many distributions you have the following choices

- Enabled
  - Enforcing: fully functional
  - Permissive: not blocking anything, but logging
    - SELinux support is available in the kernel, so applications will load with all SELinux libraries and behave differently
    - Expect unexpected behavior on some occasions.
- Disabled
  - SELinux support is not available in the kernel, so applications will load differently

# Why should you run SELinux?

The best reason to implement SELinux is to

- enforce mandatory access controls to **confine user programs** to the least privilege required for their operation.

ISOLATION

LIMIT root user

Other noticeable improvements include:

- Access control for kernel objects and services
- Access control over process initialization, inheritance, and program execution
- Access control over file systems, directories, files, and open file descriptions
- Access control over sockets, messages, and network interfaces

# Why not run SELinux

- Getting the policy file is difficult
- SELinux breaks your code
- SELinux does not like TEXT RELOCATION in your code/library

# SELinux and TEXT Relocation

So relocating the “TEXT” means changing the executable code segment so that the absolute addresses (of both functions and data — variables and constants) are correct for the base address the segment was loaded at. Doing this, causes a Copy-on-Write for the executable area.

SELinux will normally (in enforcing mode) not allow this. But can be allowed by applying special rules on the code file (often a shared library).

# Alternatives

- **AppArmor and Tomoyo**
  - AppArmor was originally developed because SELinux was viewed as too complex for typical users to manage
  - **Implements a pathbased MAC (Unlike SELinux, which is based on applying labels to files, AppArmor works with file paths)**
  - Does not require xattr for filesystem
  - AppArmor in SUSE and Ubuntu

- **Simple MAC Kernel (SMACK)**

  Smack implements mandatory access control (MAC) using labels attached to tasks and data containers, including files, SVIPC, and other tasks. Smack is a kernel based scheme that requires an absolute minimum of application support and a very small amount of configuration data.

# Hardening Unix kernels

Many unix variants including Android have adopted some of the SELinux features to implement hardening solutions of the kernel, e.g. AppArmor

Android use of SELinux see (for background reading)
https://source.android.com/security/selinux/images/SELinux_Treble.pdf

# Linux hardening:
# IMA Integrity Measure Architecture

The goals of the kernel integrity subsystem are to detect if files have been accidentally or maliciously altered, both remotely and locally, appraise a file's measurement against a "good" value stored as an extended attribute, and enforce local file integrity. These goals are complementary to Mandatory Access Control(MAC) protections provided by LSM modules, such as **SElinux** and Smack, which, depending on policy, can attempt to protect file integrity. The following modules provide several integrity functions:

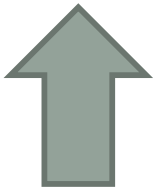
- Collect - measure a file before it is accessed.
- Store - add the measurement to a kernel resident list and, if a hardware Trusted Platform Module (TPM) is present, extend the IMA PCR
- Attest -if present, use the TPM to sign the IMA PCR value, to allow a remote validation of the measurement list.
- Appraise - enforce local validation of a measurement against a 'good' value stored in an extended attribute of the file.
- Protect - protect a file's security extended attributes

The first three functions were introduced with Integrity Measurement Architecture (IMA) in 2.6.30. The EVM/IMA-appraisal patches add support for the last two features.

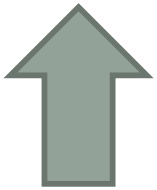
Source: http://linux-ima.sourceforge.net/

# Example IMA output at startup

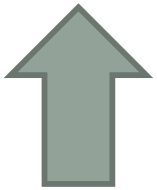
```
10 1609097048173d7c1659ff30713741b98020d0d8 ima a36e075c9dc23bd71886d55e0daee556143f8e7e boot_aggregate
10 3d82394c528268a62687c8b59b34c39137b4fb3c ima ff65625e34131617ef3ac5fead5cb285b6aa73b9 /init
10 231309f206f12296cff6c59c6e3bceb45dc285bb ima 602acdac8864b113d4546bf8f2d7b96c81607792 /init
10 9d20d222ae9a3a9c80b2a9f0b3c08a5786847ed6 ima ff4c31959c04f865c859d8df331a6bf6b6967cef ld-2.10.1.so
10 88a2bee776f4bd1305ce6ded08611fdecb5bf0db ima 44e727b6c99370d373d0acef95631e0950ef8c00 ld.so.cache
10 0c86c858b30abf3f2c3b0269a56204f642f3ef71 ima a0ed012c34fee03e9fecee8d4a7bcab1ff98f091 libnash.so.6.0.87
10 676a27e4112c9fb5677b7ef38cc488d87fca0846 ima bf094171a04b06d642c931d7c2ce0a707a659827 libbdevid.so.6.0.87
10 04c918d9ccb56ca92a118b72f8b2f60da0791887 ima 9355215ed19d72287c446d0f3b7b41dcaddbd254 libdevmapper.so.1.02
10 6585544486aee180aa950b1bb6132434cf0f71d6 ima caca8bbf8ff4957584d114c753b7c4f15936af7c libparted-1.8.so.8.0.0
```



PCR 10 values



Measurement hashes



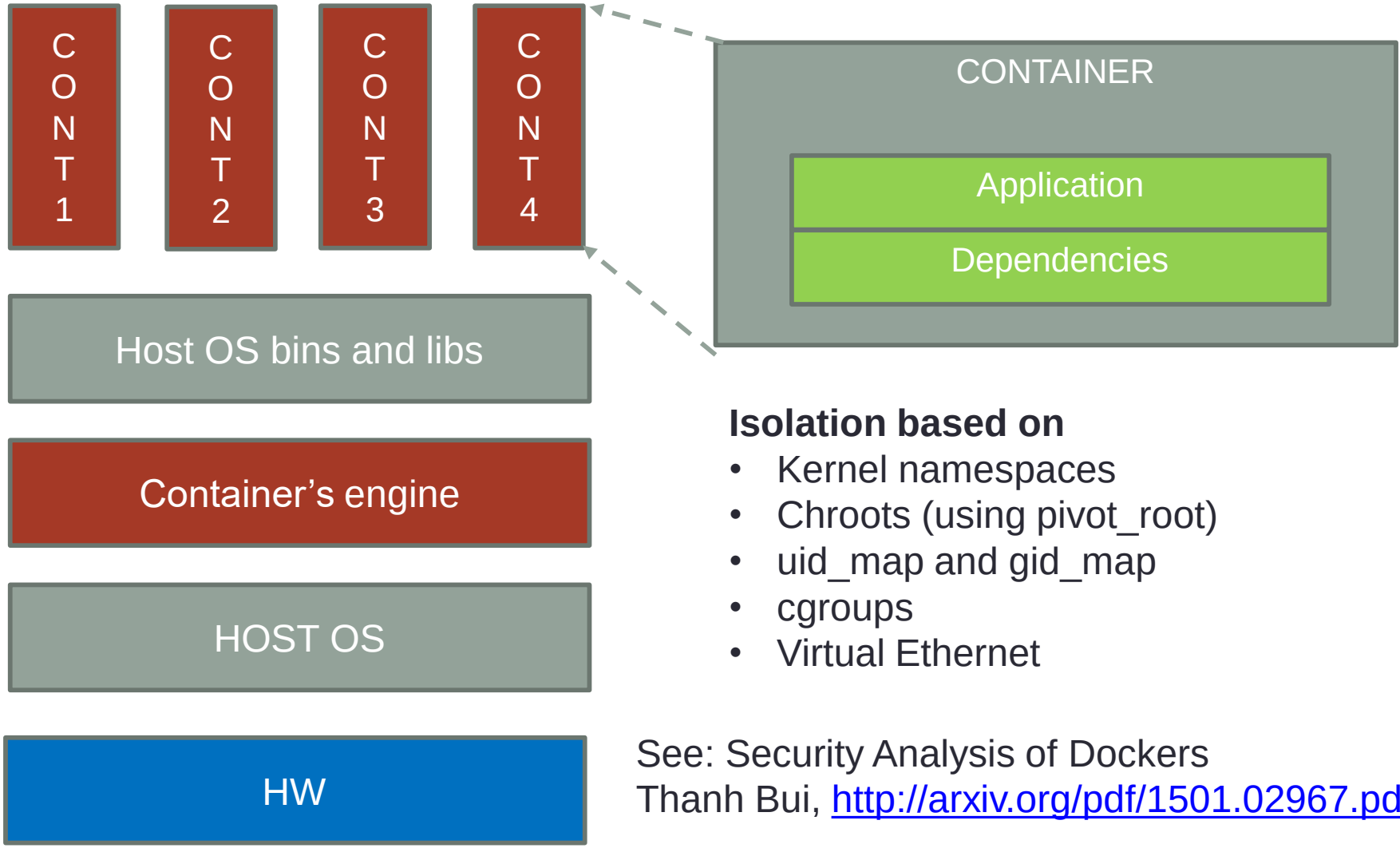
Files checked

From: Using IMA for Integrity Measurement and Attestation, David Safford, Mimi Zohar, Reiner Sailer, IBM, 2009

# UNIX CONTAINERS (LXC)

And OpenVZ

# Container based virtualization




**Isolation based on**

- Kernel namespaces
- Chroots (using pivot_root)
- uid_map and gid_map
- cgroups
- Virtual Ethernet

See: Security Analysis of Dockers
Thanh Bui, http://arxiv.org/pdf/1501.02967.pdf

# TRUSTED COMPUTING IN ANDROID- OS AND IOS?


# Overview

- Security in Android
  - Foundation for security
  - Interaction and access-control in Android
  - Permissions
- iOS: differences (security wise) compared to Android

- Can we trust smartphones?
  - Current views
  - App trust through reputation
  - BYOD
  - App dynamics

# “THE BASICS” IN ANDROID

# Fundament: Android Package (APK) file structure


**/META-INF**

- **MANIFEST.mf** - sha-1 digest for each file of the applicationen
- **HelloWorld.sf** - sha-1 digest of each file (based on info from Manifest.mf)
- **HelloWorld.rsa** - PKCS#7 RSA signature of HelloWorld.sf

**/res**

- **Application's resource files**

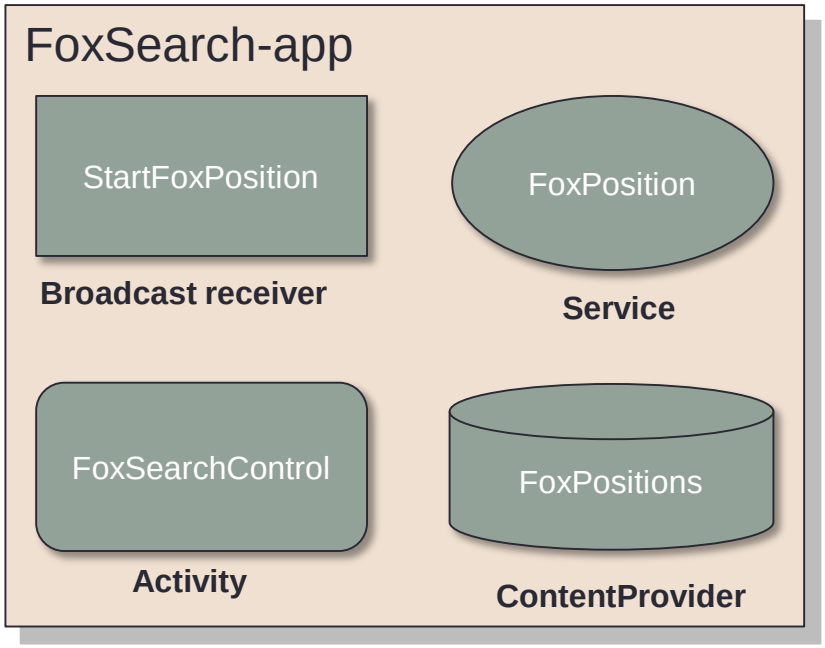
**/**

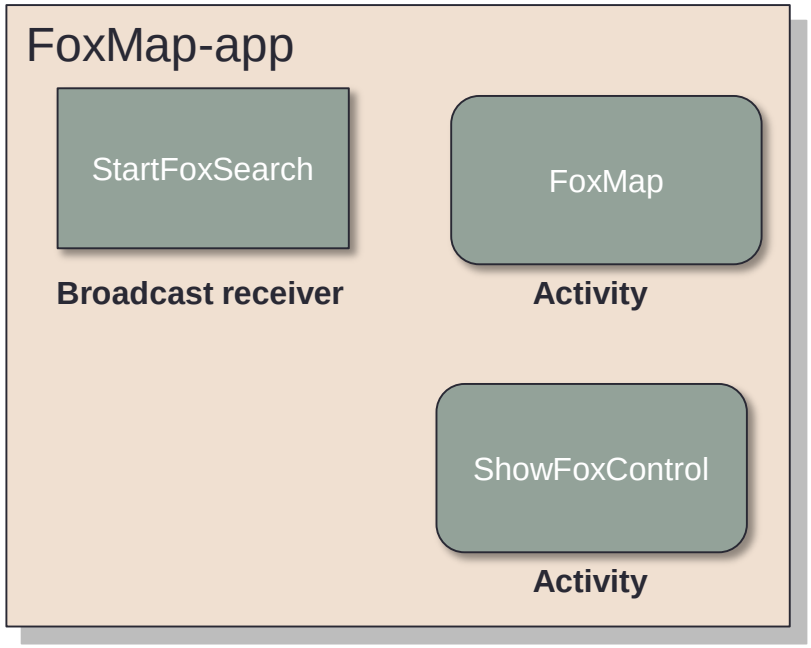
- **AndroidManifest.xml** – app information to Android system
- **classes.dex** - compiled Android Dalvik app
- **resources.arsc** – table on all resources in the application

# Fundament: Android applications are built according a common design pattern

- **Activities** form the presentation layer; one for each screen and Views make the UI for an activity.
- **Intents** specify what is to be done
- **Services** are background processeses with no UI: update  data and trigger events
- **Broadcast receivers** offer message mailboxes from other applications and can trigger intents that start an application
- **Content providers** provide data/resources

# Android Applications - Example




FoxSearchControl : UI for starta/stop searchfunction
FoxPosition: Contacts external localisation server
FoxPositions: Save last positions

# APPLICATION SECURITY - BASIC

Access control

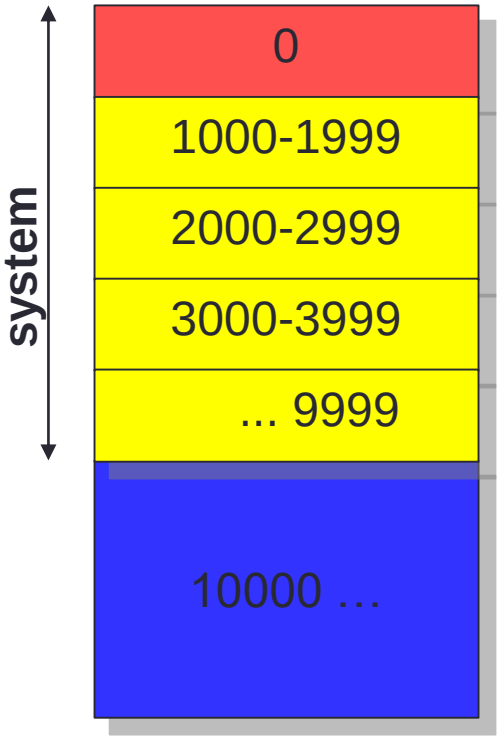
Runtime protection

# Access control in Android

- Android protects applications att
  - (Linux) system level and
  - inter-component communication level: **Permissions**
  - **SE-Linux features**

- **Each application runs as a unique Linux user** under its own identity (user ID) which limits the damage an application program error can inflict on other applications :
  - Linux access control (as usual) keeps apps apart
  - User ID is (locally) assigned during app installation

# User ID Table




Root: Only few root processeses e.g. init, runtime, Zygote,

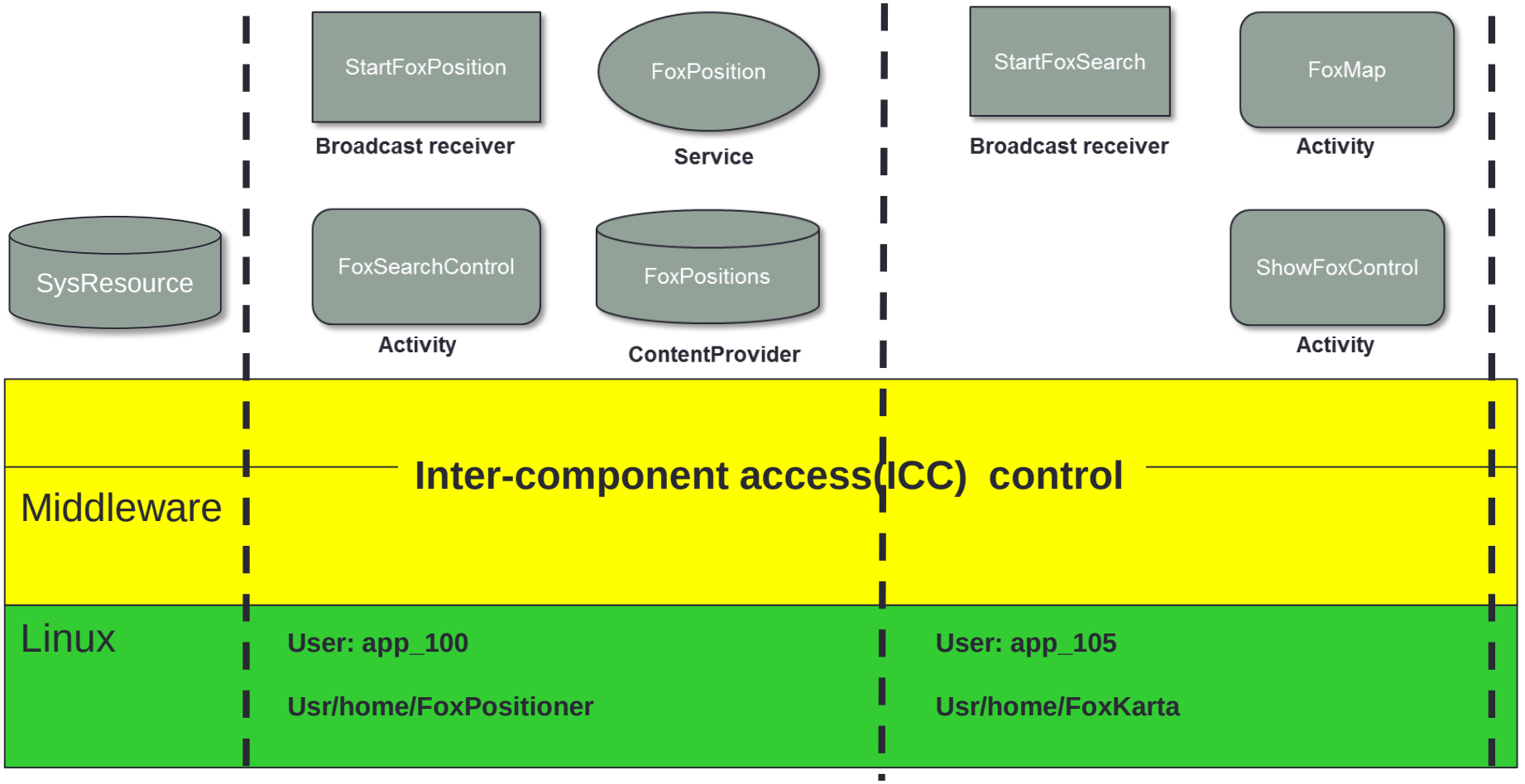
System services

ADB and other debug services

Android services that the kernal is aware

Apps get user ID ≥ 10000

# Access control



# Access control in Android

- Android has implemented a comprehensive mechanism for access control.
- The use of signed applications is an important ingredient in this BUT NOT to assure the (correct) origin of the applications but to assure that the assignment of unique user IDs cannot be spoofed.
- We look at signed applications first

# Signed applications

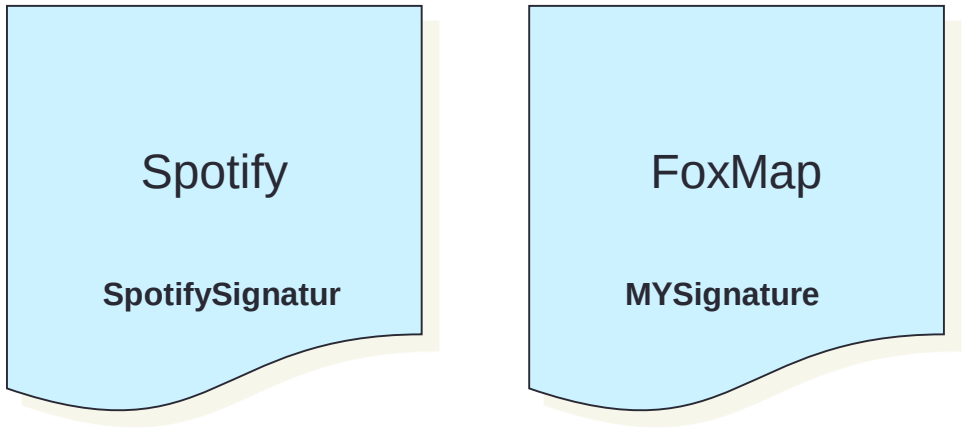
- To distinguish an application. Application signing by self-signed certificates is used.

Note I do not write “identify”

- Androids Distribution does not accept app signed with the default key in the SDK
  - Hence it is a policy issue if known (SDK) keys are accepted for an app.
- Two applications can share the same user ID if they are signed by using the same key/certificate (should normally be avoided)

# PKI structure and access control

**without** signature permissions




- In vanilla Android the signatures have no security value if signature permissions are not used.
- Remember: signature is in this case only used to tell applications apart.

# PKI structure and access control
## with signature permissions

Android distinguishes four core platforms components each having their own key

- **Platform:** a key for packages that are part of the core platform.
- **Shared:** a key for things that are shared in the home/contacts process.
- **Media:** a key for packages that are part of the media/download system.
- **Testkey/releasekey (Application) :** the default key to sign with if not otherwise specified

# Runtime protection

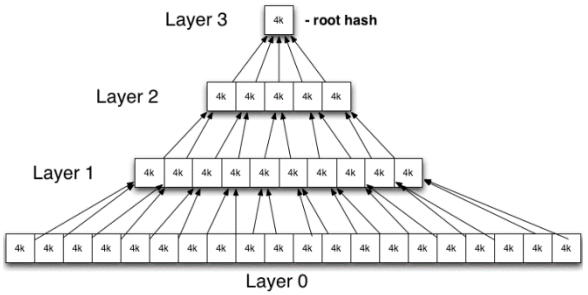
- **Address Space Layout Randomization (ASLR)** is a known technique that aims to reduce the possibility of mounting attacks that exploit ways to get access to desired memory locations through randomization. Attackers are with ASLR in place faced with a much harder problem to find thus address locations. This reduces, for example, the possibility to make a buffer overflow condition into something useful in an attack. ASLR is introduced in Android 4.0

- **ARM NX** Since many Android devices use ARM type CPUs the not execute (NX) feature for data reduces the risk of exploits where stored data is executed. Using the NX bit data can be tagged as not executable until, say, a successful verification has been carried out on which the NX status is removed. Android supports HW based NX since the Ice cream sandwich release.

block oriented storage

# DM-VERITY

- Transparent block-level integrity protection solution for read-only partitions
- dm-verity is a device mapper target
- Uses hash-tree
  - Calculates a hash of every block
  - Stores hashes in the additional block and calculates hash of that block
  - Final hash – root hash – hash of the top level hash-block
  - Root hash is passed as a target parameter and protected by signature. A public key is included on the boot partition, which must be verified externally. That key is used to verify the signature for that hash.
- Update can be done only by overwriting entire partition




- Used in ChromeOS, KitKat 4.4, and some Unix systems to protect read-only partition

http://source.android.com/devices/tech/security/verifiedboot/

# SE-Linux in Android

Each process and object has an associated label, which is also called a context.

Contexts are comprised of a user, a role, a type and an multi-level-security (MLS) portion:

- The type of a process is often referred to as a domain and is defined in the SELinux policy.
- The label of an object is usually decided by the corres-ponding security contexts files.

SELinux policy also contains the rules that state how each domain may access each object.

In Android 4.4 to Android 7.0, SELinux policy files are included in the rootfs image

# IMPLEMENTATION

# Android’s platform security

All Android product vendors do make adoptions to Android when integrating Androiod to their hardware

- **Secure boot process**: which components that are verified and how differ among vendors and products
- **Vendors try to limit access to root**: some suceed better than others, eg Knox by Samsung
- Most products that use a mobile network modem have **modem hardware that is isolated from the application** processor subsystem. The modem is controled via the RIL (Radio Interface Layer) which is in practice a proprietary library with a standard API

# IOS

# iOS – remark on material

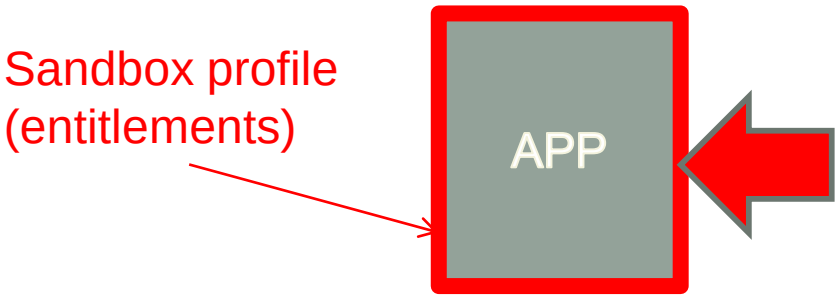
- Since Apple has not revealed all iOS details we have less exact knowledge how things work and in some cases we rely on people that reported on their analysis and reverse engineering work

- Here we treat iOS  through comparison with Android

# iOS sandboxing

- All applications in iOS run as user “mobile”.
  - Therefore iOS applies fine-grained process runtime security policies specifying file and system access restrictions for the applications.

- iOS assigns each installed third party application (incl. preferences and data) a home catalogue1 (or container) of the file system. By default the application may only read and write files within their container, but more specific permissions are detailed in the sandbox profile. When an application is launched, its sandbox profile is determined by so-called profile entitlements.




The XNU Sandbox kernel extension (a.k.a. “Seatbelt”) is implemented as a policy module for the TrustedBSD **Mandatory Access Control** (MAC) framework. The Sandbox extension is a loadable kernel module that installs **MAC policy** hooks for accessing system objects and resources.

[1] /var/mobile/Applications/UUID, where the application Universally Unique IDentifier (UUID) is randomly generated when the application is installed.

# iOS and certificates

- Apple uses X.509 certificates for signature verification. Typically, there is a certificate chain, like:
  - Apple Root CA
  - Some intermediate CA
  - yet another intermediate CA
  - End-Entity

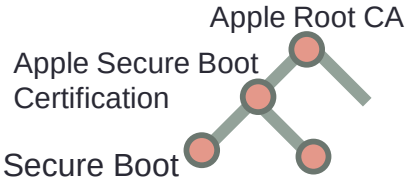
- Disassembling the bootloader (2012) and looking at the X.509 implementation revealed that the code does not check X.509v3  basic constraints. (still true ?, 2014)

# Built-in trusted boot

```
ADC7 US
ADD4 Apple Inc.
ADE9 Apple Certification Authority
AE11 Apple Root CA
AE22 0604252140362
AE31 3502092140362
AE58 Apple Inc.
AE6D Apple Certification Authority
AE95 Apple Root CA
B062 https://www.apple.com/appleca/
B093 Reliance on this certificate by any party assumes
     acceptance of the then applicable standard terms
     and conditions of use, certificate policy and
     certification practice statements.
```

The iPhone products implement a trusted boot scheme. The root of trust is an Apple root certificate stored in the boot ROM. Firmware images are stored encrypted and signed.

The PKI scheme in use has three **levels**




There are three different boot **modes**:

- In the **Normal boot** mode the trusted boot sequence is:
  Boot ROM → LowLevelBootloader (LLB) → Iboot → Kernel.
- **Recovery mode** is a fail-safe option in the Iboot bootloader that allows uploading of a RAM disk and reflashing of the device.
- **DFU mode** is a fall-back option that allows to restore from a given boot state. DFU has a slightly different boot sequence.

DFU and Recovery modes are accessible over the USB interface.

iPhone jailbreaking targets mostly the circumvention of this trusted boot (see, e.g., iPhone Dev Team)

# iOS vs Android

**Similarities**

- Unix/Linux based, ASLR, NX protection
- Use of profiled open source libs
- SDK tools freely available
- APP installation packages must be signed
- APPs and their data are isolated
- Permission model present
- Native browser uses webKit

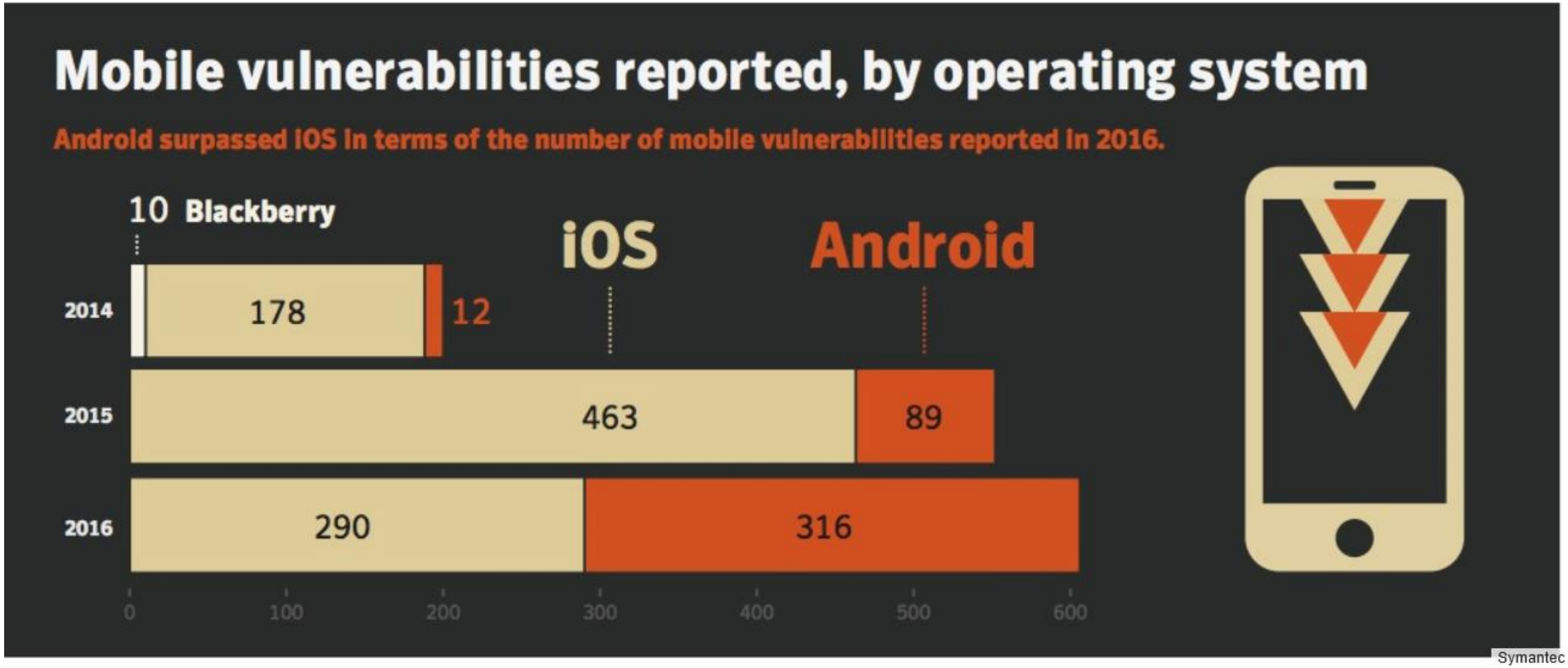
**Differences**

- Kernel protection differences
- Programming languages differ
- Signing is used to check origin of APP installation packages (e.g. limit to come from specific source)
- iOS uses sandboxing for isolation, most apps have same user id (501, mobile), 35 sandbox profiles.
- iOS permission are more handled through fixed policies. Very few by user consent.
- Screening of APPs before releasing for deployment (is not a device issue though)
- iOS has more developed data/file encryption support
- Trusted boot in iPhones which in Android phones is vendor specific
- iPhone APPs are DRM protected during delivery and on the device

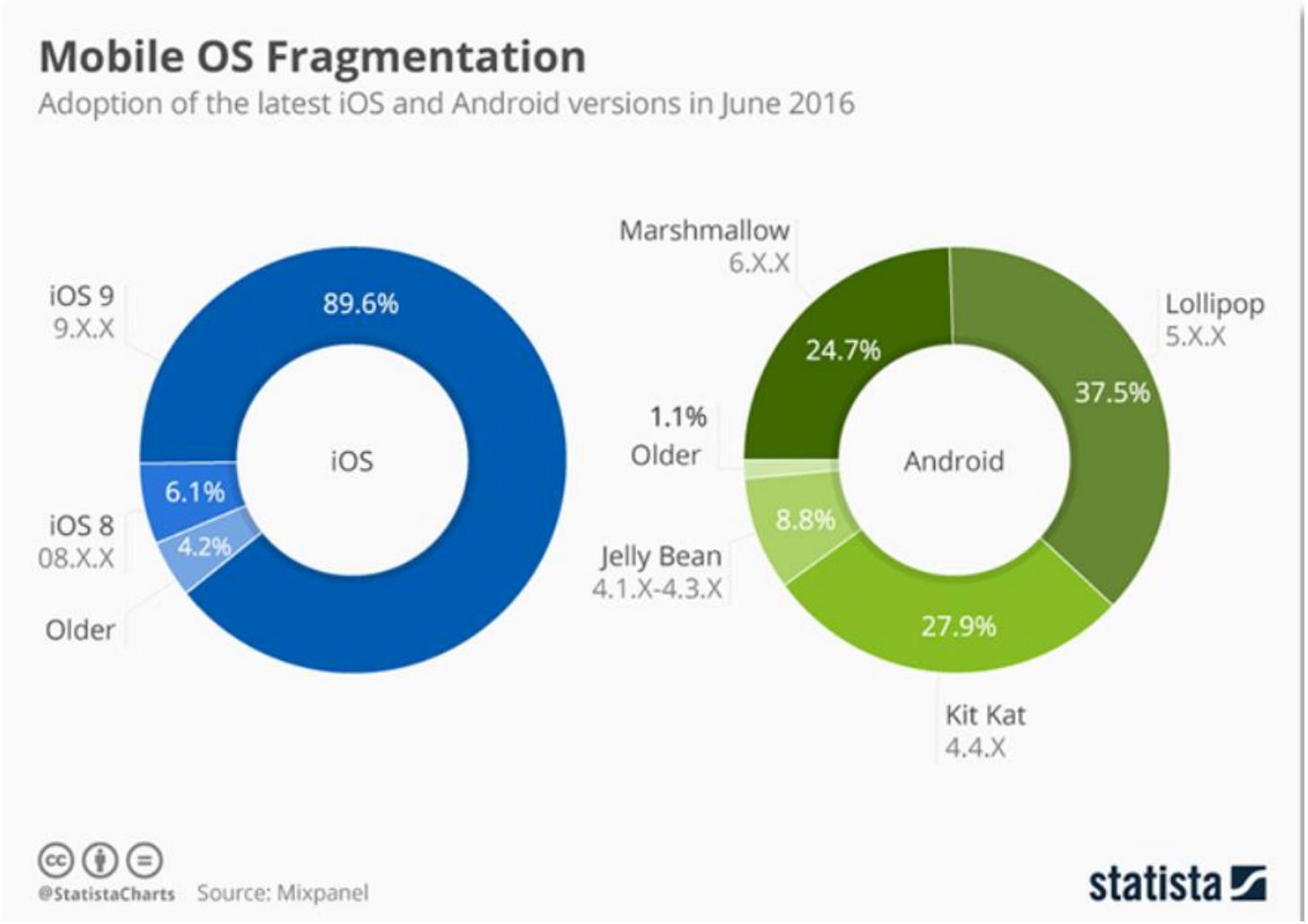
# CAN WE TRUST SMARTPHONES ?

And the result is ….

# iOS vs Android vulnerabilities



# iOS is less fragmented



# References


- Android developer site pages: (too many to list here)
- Understanding Android Security, William Enck, et. al, Security & Privacy January/February 2009
- Developing Secure Mobile Applications For Android, iSEC Partners 2008
- Android Permissions Demystified, Adrienne Porter Felt, et al 2011.
- iOS Security, Apple Inc, May 2012
- https://media.blackhat.com/bh-us 11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf
- http://esec-lab.sogeti.com/dotclear/public/publications/11-hitbamsterdam-iphonedataprotection.pdf

# BYOD….

Corporate use of smartphones

Security perspective

# BYOD trend

- Companies and organizations allow increasing their employees to use their own devices as work platforms.

- Obviously this reduces companies cost of device (PC, phone, etc) hardware
- Mostly people are satisfied with using their own device

BUT

- security wise BYOD is problematic. How to enforce security policies

# Examples:

- Define profiles that govern the behaviour: private or work usage
  - Samsung Knox
  - Android at work
  - Blackberry

# References


- Android developer site pages: (too many to list here)
- Understanding Android Security, William Enck, et. al, Security & Privacy January/February 2009
- Developing Secure Mobile Applications For Android, iSEC Partners 2008
- Android Permissions Demystified, Adrienne Porter Felt, et al 2011.
- iOS Security, Apple Inc, May 2012
- https://media.blackhat.com/bh-us 11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf
- http://esec-lab.sogeti.com/dotclear/public/publications/11-hitbamsterdam-iphonedataprotection.pdf