Sicherheit:
Fragen und Lösungsansätze

im Wintersemester 2012 / 2013

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

Teil 10: Layered Design Including Certificates and Credentials

v. 27.01.2013

Themen der Vorlesung

Part I: Challenges and Basic Approaches

1) Interests, Requirements, Challenges, and Vulnerabilities
2) Key Ideas and Combined Techniques

Part II: Control and Monitoring

3) Fundamentals of Control and Monitoring
4) Case Study: UNIX

Part III: Cryptography

5) Fundamentals of Cryptography
6) Case Studies: PGP and Kerberos
7) Symmetric Encryption
8) Asymmetric Encryption and Digital Signatures with RSA
9) Some Further Cryptographic Protocols

Part IV: Security Architecture

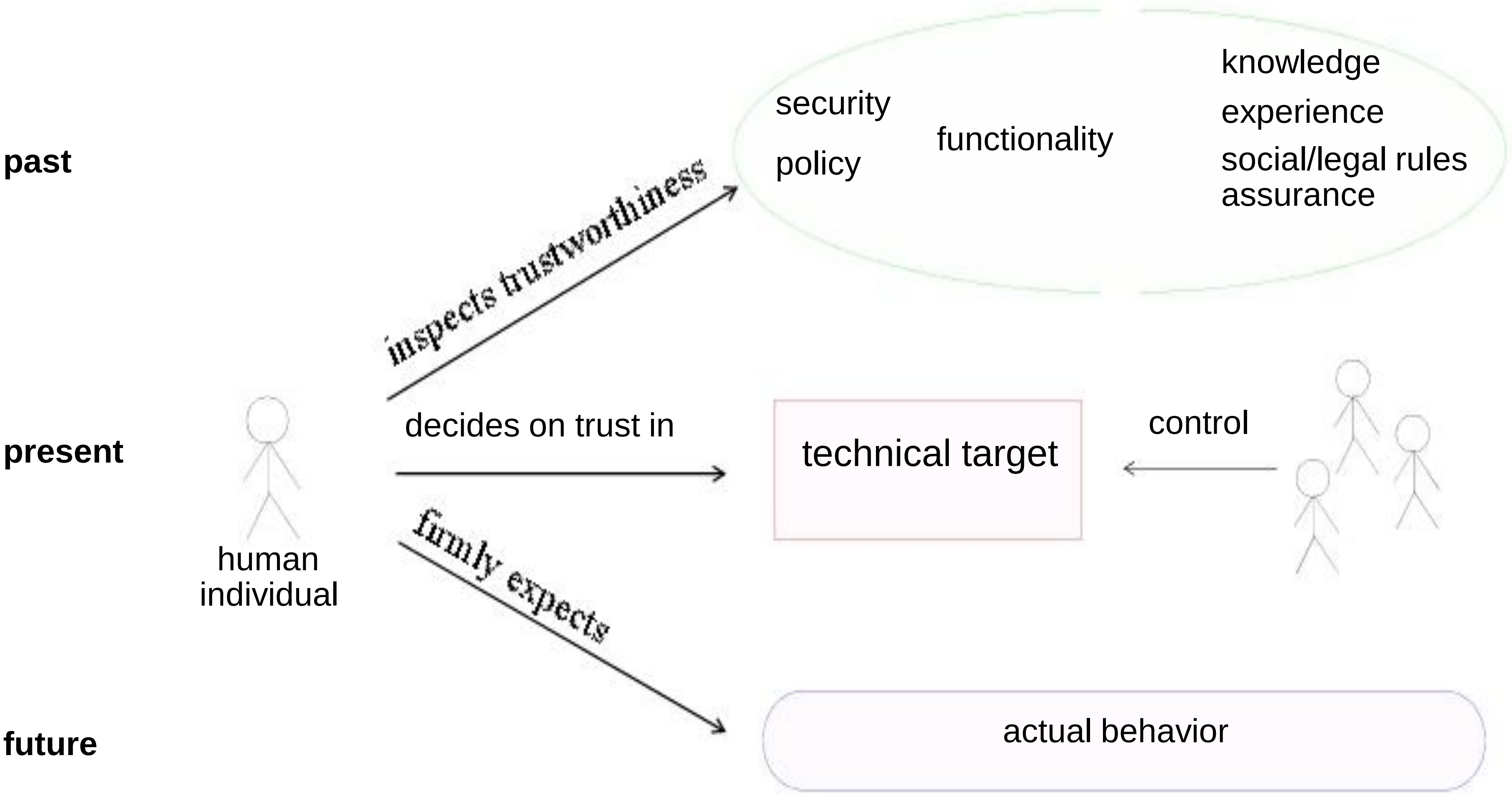
10) **Layered Design Including Certificates and Credentials**

Trust and trustworthiness

- items serving to *found trustworthiness* of a target:
  - a security policy that meets explicitly claimed interests
  - an appropriately designed and reliably implemented functionality
  - verified knowledge
  - justified experience
  - compliance with social and legal rules
  - effective assurances
- an individual (community) may *decide to put trust* in such a target:

  the decider's own behavior
  is firmly grounded on the expectation
  that the target's current or future actual behavior
  - often fully or at least partly hidden and thus only partially observable -
  will match the specified or promised behavior
- trust in the technical target is inseparably combined
with trust in the agents controlling that target

Some aspects of an informational concept of trust



Establishing reasonable trust reductions

- identify small parts of a computing system,
  if possible, preferably under your own and direct control,
  as indispensable targets of trust

- argue that the wanted behavior of the whole system
is a consequence of justified trust
  in only these small components

Trust reductions for control and monitoring

- **starting point:**
  an overall computing system consisting of
  clients, servers, networks and many other components

- **reduction chain:**
  - a distributed application subsystem
  - the underlying operating system installations
  - the operating system kernels
  - the “reference monitors” that implement access control within a kernel

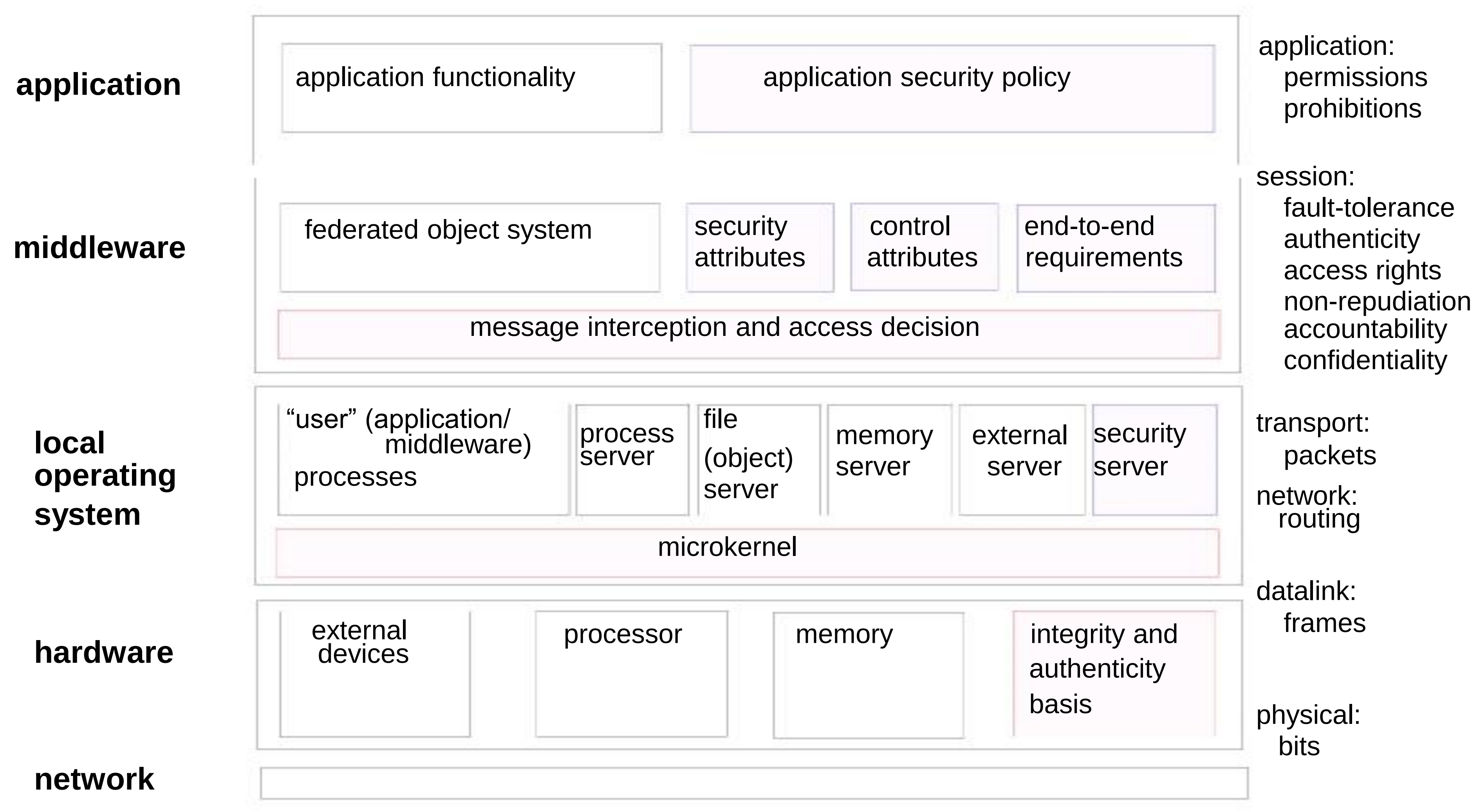
- **extended reduction to hardware support:**
  - “trusted platform modules”
    (enforcing authenticity and integrity)
  - personal computing devices
    (storing and processing cryptographic secret keys)

Trust reductions for cryptography

- **starting point:**
  an overall computing system consisting of
  clients, servers, networks and many other components

- **reduction chain:**
  - cryptographic mechanisms
  - cryptographic key generation and distribution
  - storing and processing secret keys

Layered design: a fictitious architecture



application

application functionality

application security policy

middleware

federated object system

security attributes

control attributes

end-to-end requirements

message interception and access decision

local operating system

“user” (application/ middleware) processes

process server

file (object) server

memory server

external server

security server

microkernel

hardware

external devices

processor

memory

integrity and authenticity basis

network

application:
permissions
prohibitions

session:
fault-tolerance
authenticity
access rights
non-repudiation
accountability
confidentiality

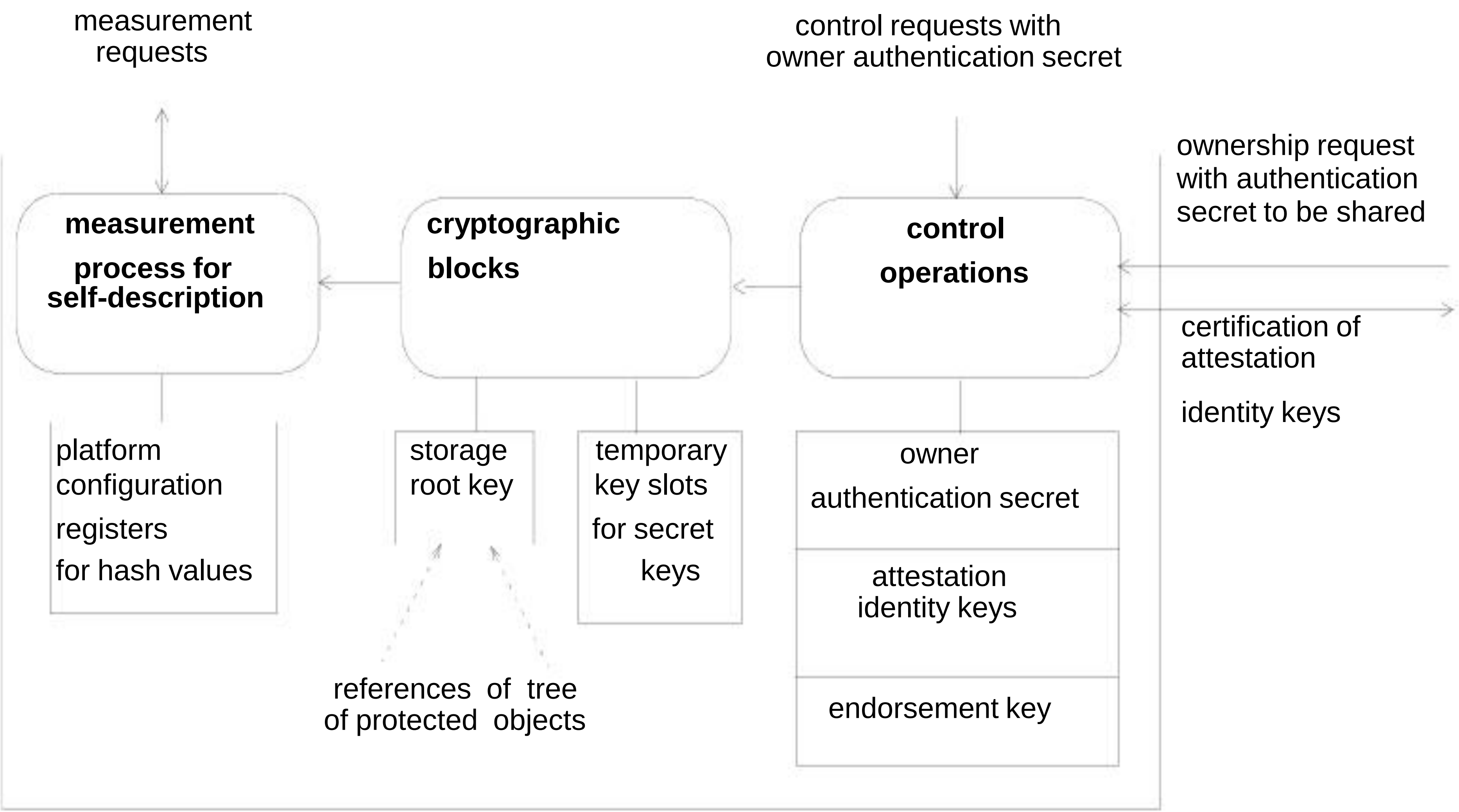
transport:
packets

network:
routing

datalink:
frames

physical:
bits

Integrity and authenticity basis (trusted platform module)



measurement requests

control requests with owner authentication secret

ownership request with authentication secret to be shared

measurement process for self-description

cryptographic blocks

control operations

certification of attestation

identity keys

platform configuration registers for hash values

storage root key

temporary key slots for secret keys

owner authentication secret

attestation identity keys

endorsement key

references of tree of protected objects

Integrity and authenticity basis: main functions of an instance

- **enables the attached system to generate and store a tamper-resistant *self-description* regarding its actual *configuration state*:**
  - represented by a sequence of chained hash values
  - iteratively computed by a *measurement process*
  - stored in protected *platform configuration registers*
  - comparable with a previous or a normative state
- **encapsulates and protects implementations of basic *cryptographic blocks*, including the key generation, storage and employment:**
  - symmetric encryption and decryption for internal data
  - asymmetric decryption for external messages
  - asymmetric authentication (digital signatures) for external messages
  - anonymization by using public (authentication) keys as pseudonyms
  - random sequences for key generation and nonces
  - one-way hash functions for generating the self-descriptions as hash values
  - inspection of timestamps by a built-in timer
- **both *globally identifies* and *personalizes* the attached system:**
  - physically implanted, worldwide unique asymmetric *endorsement key*
    - inserted *authentication secret* shared with the owner

Secure booting and add-on loading: important assumptions

- the overall system, seen as a set of programs, is organized into a hierarchical component structure without loops

- there is one initial component that has *authenticity* and *integrity*, a *bootstrapping* program,
  evaluated at manufacturing time to be trustworthy,
  and securely implanted into the hardware,
  employing a tamper-resistant read-only memory

- each noninitial component (program) originates from a responsible source, which can be verified in a *proof of authenticity*;
  such a proof is enabled by a certificate referring to the component
  and digitally signed by the pertinent source

Secure booting and add-on loading: important assumptions

- each noninitial component has a well-documented state that can be measured; such a state is represented as a *hash value*;
  the expected state, as specified by the source,
  is documented in the *certificate* for the component

- each component, or some dedicated mechanism acting on behalf of it, can perform an *authenticity and integrity check* of another component, by measuring the actual state of the other component and comparing the measured value with the expected value

- the *hardware* parts involved are authentic and possess integrity, too,
  which is ensured by additional mechanisms or supposed by assigning trust

- the *certificates* for the components are authentic and possess integrity

Basic booting and loading procedure

```
load initial component;
repeat
[invariant: all components loaded so far are authentic and possess integrity]

    after having been completely loaded, a component
    -  first checks a successor component for authenticity and integrity
    -  then, depending on the returned result,
         either lets the whole procedure fail
         or
         loads the checked successor component

until all components are loaded
```

Some extensions and variants

- **recovery from failures**
  the procedure automatically searches
  for an uncorrupted copy of the expected component
- **chaining**
  hash values are chained, superimposing the next value on the previous value,
  for producing a hash value of a sequence of components
- **data with "integrity semantics"**
  the procedure also inspect further data relevant to the overall integrity,
  such as separately stored installation parameters
- **integrity measurement**
  the procedure recomputes the hash value of the component actually loaded and
  stores this value into dedicated storage for reporting
- **reporting**
  the recomputed and stored hash values are reported
  to external participants as the current self-description

Middleware: functional and security services

- managing the local fractions of the static and dynamic aspects of the system, including *local control and monitoring*

- enabling interoperability across the participating sites, and also contributing to *global control and monitoring* by regarding incoming and outgoing messages as *access requests*

- establishing virtual *end-to-end connections* to remote sites (the *session layer* according to the ISO/OSI model), dealing in particular with
  - *fault tolerance*
  - *authenticity*
  - *access rights*
  - *non-repudiation*
  - *accountability*
  - *confidentiality*

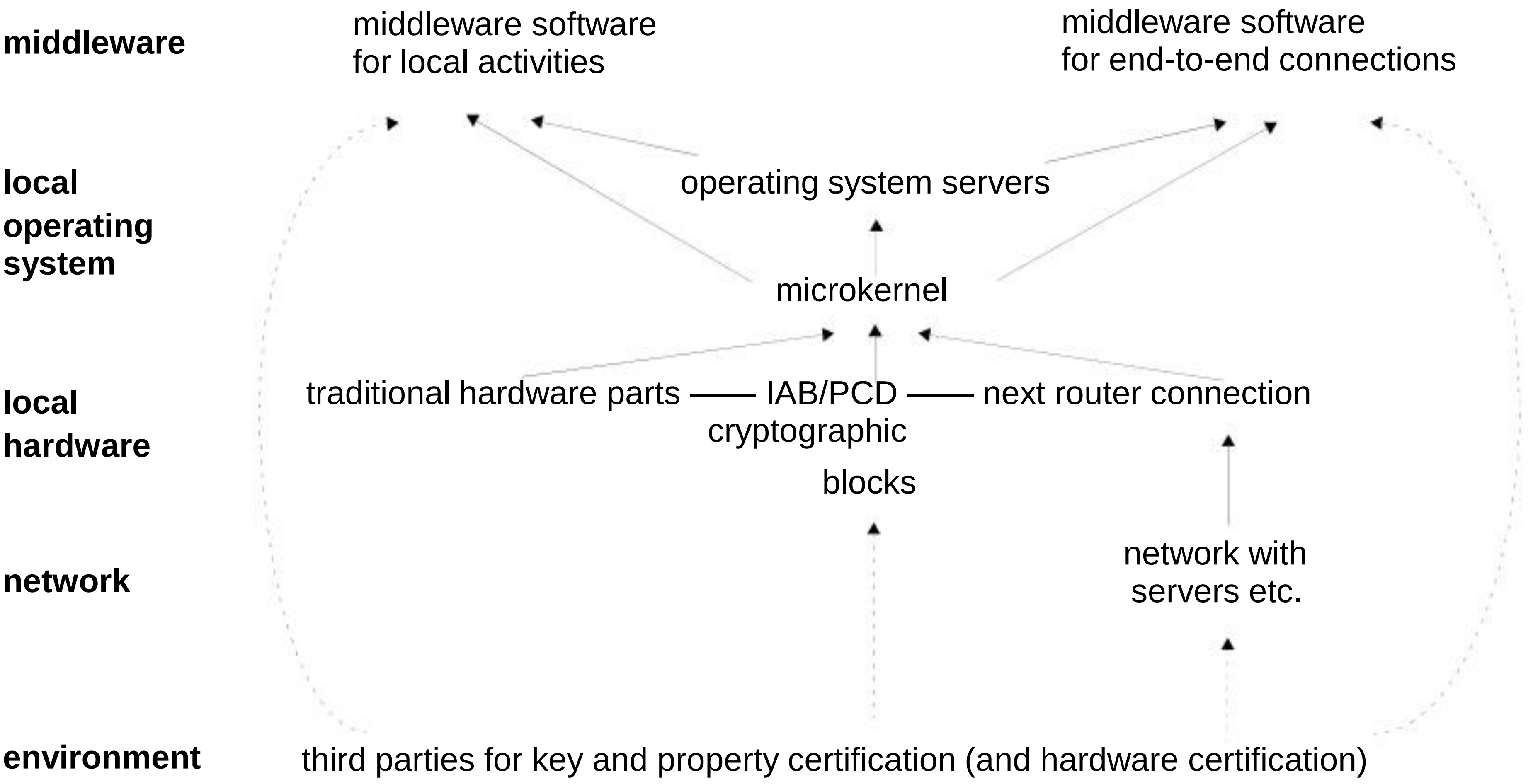
Informational infrastructure and organizational environment

- with regard to sites (i.e., their extended operating systems), enabling *mutual authentication*
  using *certificates* for the public parts of *asymmetric key pairs*,
  and generating and distributing symmetric *session keys*

- with regard to “user processes”,
  enabling autonomous *tunneling*:
  wrapping data by encryption and authentication
  under the mastership of the *endusers*
  (as proposed for *Virtual Private Networks*, *VPNs*)

- enabling *anonymity*,
  by employing (the public parts of) asymmetric key pairs as *pseudonyms*,
  and by dedicated *MIX servers* with *onion routing*

Middleware: support by underlying layers and global infrastructure



middleware

middleware software
for local activities

middleware software
for end-to-end connections

local operating system

operating system servers

microkernel

local hardware

traditional hardware parts —— IAB/PCD —— next router connection

cryptographic

blocks

network

network with
servers etc.

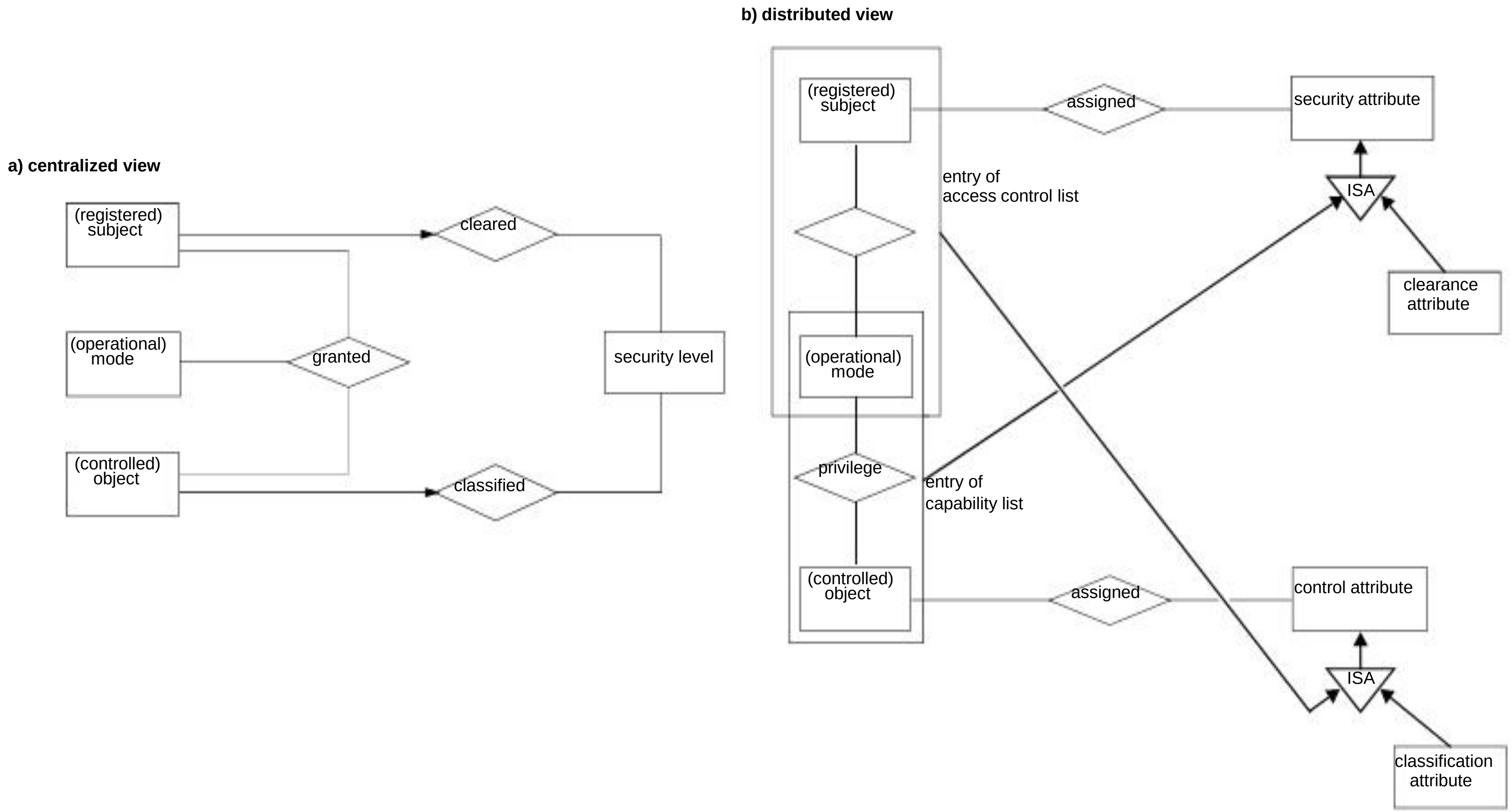
environment

third parties for key and property certification (and hardware certification)

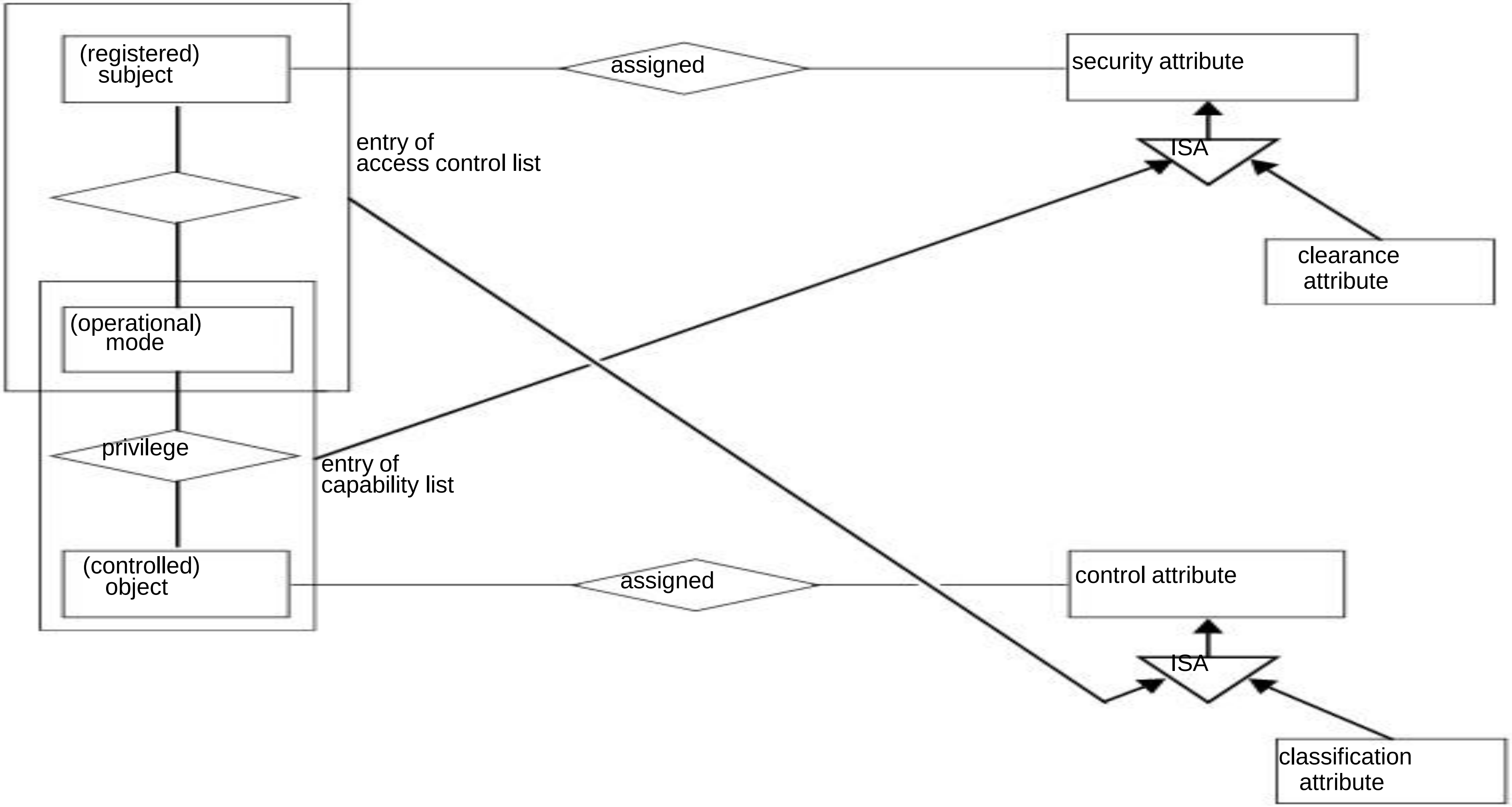
Middleware instantiation of control and monitoring

- for a *distributed computing system*, the *isolation* of participating subjects and controlled objects is split into two parts
- at a subject's site, a *subject*, acting as a *client*, is confined concerning *sending* (messages containing) *access requests*
- at an object's site, a target *object*, acting as a *server*, is shielded concerning *receiving* such (messages containing) *access requests* and then actually interpreting them
- the fundamental *permissions* (and *prohibitions*) relationships between subjects and objects are represented by two complementary views
- a ternary *discretionary* `granted` relationship (*s*, *o*, *m*) is split into
  - a *privilege* (or *capability*) [*o*, *m*] for the subject *s*
  - an entry [*s*, *m*] for the *access control list* of the object *o*
- a subject can be assigned *security attributes* (e.g., a privilege [*o*, *m*]); an object can be assigned *control attributes* (e.g., an entry [*s*, *m*])
- similarly, *clearances* of subjects and *classifications* of objects are assigned

ER models of fundamental relationship classes for permissions



Fundamental relationship classes for permissions: distributed view



Programming languages: enforcing compile time features

- *object-orientation* contributes a specific kind of *encapsulation*: an instance object is accessible only by the *methods* declared in the pertinent class
- explicit commands for the *lifespan* of instance objects assist in keeping track of the current object population,
  for example by *generating* (`new`) an instance object with explicit parameters and *releasing* (`delete`) it after finishing its usage,
  possibly together with *erasing* the previously allocated memory
- *modularization* of programs,
  together with strong *visibility* (*scope*) rules for declarations,
  crucially supports confinement
- strong *typing* of objects and designators,
  including typed references (disabling “pointer arithmetic”)
  together with disciplined *type embeddings* (*coercions*),
  prevent unintended usage

Programming languages: enforcing compile time features

- *explicit interfaces* of modules, procedures and other fragments, requiring full parameter passing and prohibiting global variables, shared memory or a related implicit supply of resources, avoid unexpected *side effects*

- explicit *exception handling*
  forces all relevant cases to be handled appropriately

- for *parallel computing*,
  (full) *interleaving* semantics and explicit *synchronization*
  help to make parallel executions understandable and verifiable

- for supporting *inference control*,
  built-in declarations of *permitted information flows* are helpful

- if *self-modification* of programs is offered,
  it should be used only carefully, where favorable for strong reasons

Programming languages: controlling runtime features

- runtime checks for *array bounds*
- runtime checks for *types*,
  in particular for the proper *actual parameters* of procedure calls
- actual enforcement of *atomicity* (no intervening operations),
  if supplied by the programming language
- *dynamic monitoring* of compliance with permitted information flows
- space allocation in *virtual memory* only:
  physical-memory accesses must be mediated
  by the (micro)kernel of the operating system
- allocation of carefully *separated memory spaces*
  (with dedicated *granting* of access rights) for
  - the program (only *execute* rights)
  - its own static data (if possible, only *read* rights)
  - the *runtime stack* and the *heap*

Software engineering: helpful recommendations

- explicitly *guarding* external input values and output values
- explicitly *guarding* values passed
  for the expected range, well-definedness or related properties
- elaborating a complete *case distinction* for guarded commands
- carefully considering *visibility* and naming conventions
- handling *error conditions* wherever appropriate
- restoring a safe execution state and immediately terminating after a security-critical failure has been detected
- explicitly stating *preconditions*, *invariants* and *postconditions*
- *verifying* the implementation with respect to a specification
- inspecting *executable code* as well, in particular, capturing all interleavings for parallel constructs
- *certifying* and *digitally signing* executable code, possibly providing a hash value for *measurements*
- *statically verifying* the compliance with declarations of *permitted information flows*