Security in Computing Systems

Challenges, Approaches and Solutions

Lecture Notes extracted from a monograph published by Springer-Verlag

Joachim Biskup
ISSI - Informationssysteme und Sicherheit
Technische Universität Dortmund

Original source of these lecture notes

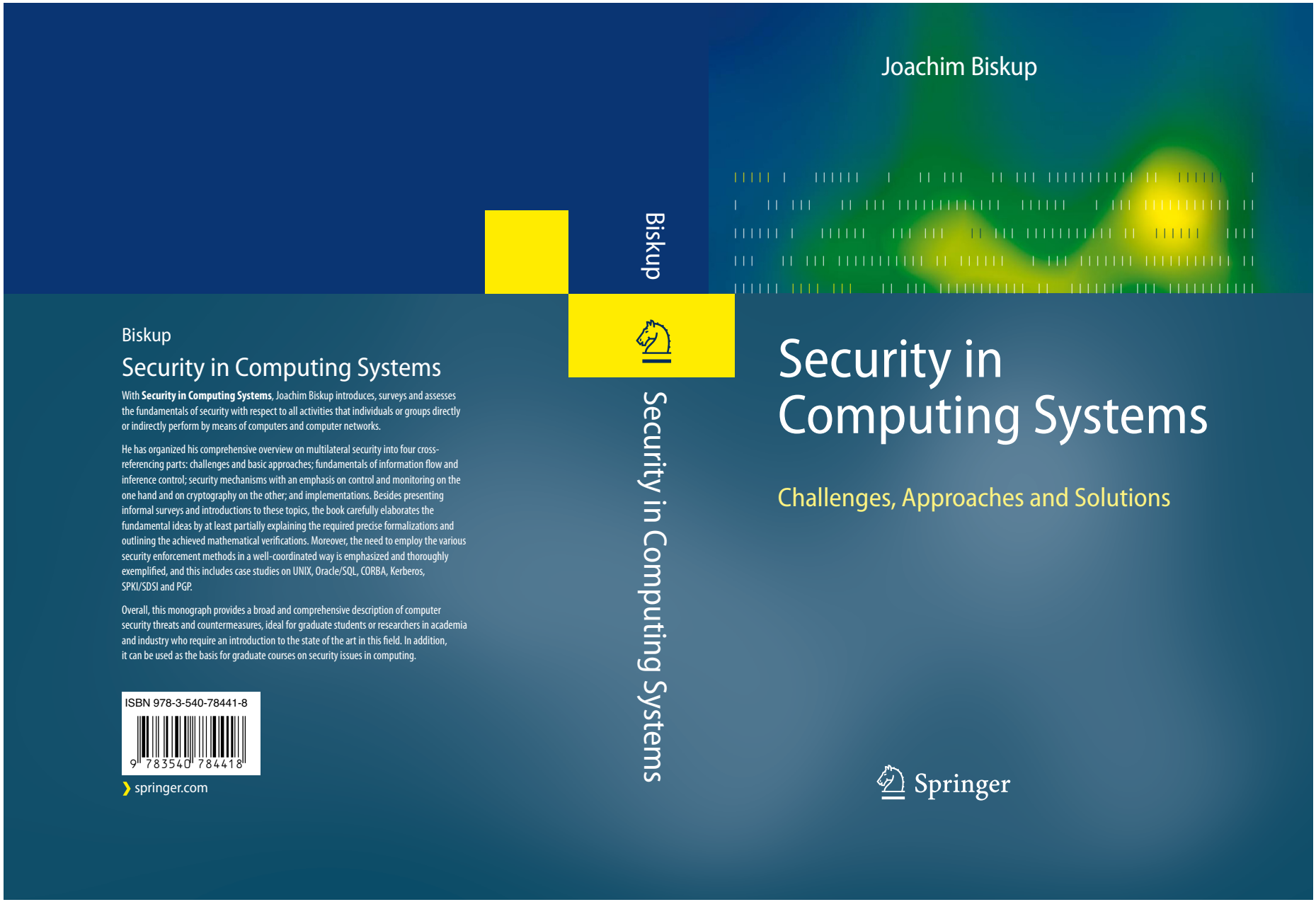


Table of Contents

Part I

Challenges and Basic Approaches

1 Interests, Requirements, Challenges, and Vulnerabilities

A notion of security

a computing system is *secure*

iff

it satisfies the intended purposes
without violating relevant informational (or other) rights

Basic security interests

- *availability* of data and activities
- *confidentiality* of information and actions
- *integrity* of the computing system
- *authenticity* of actors
- *non-repudiation* of their actions

Fundamental aspects of security

- *security* is a a comprehensive property
- security design reflects the *interests* of *participants*
- *conflicts* must be balanced
- *security requirements* identify *informational activities* and their *threats*
- *security interests* comprise
  – *availability*
  – *confidentiality*
  – *integrity*
  – *authenticity*
  – *non-repudiation*
- *security mechanisms* aim at
  – *preventing* security violations
  – *limiting* the damage caused by violations
  – *compensating* their consequences

Security evaluation

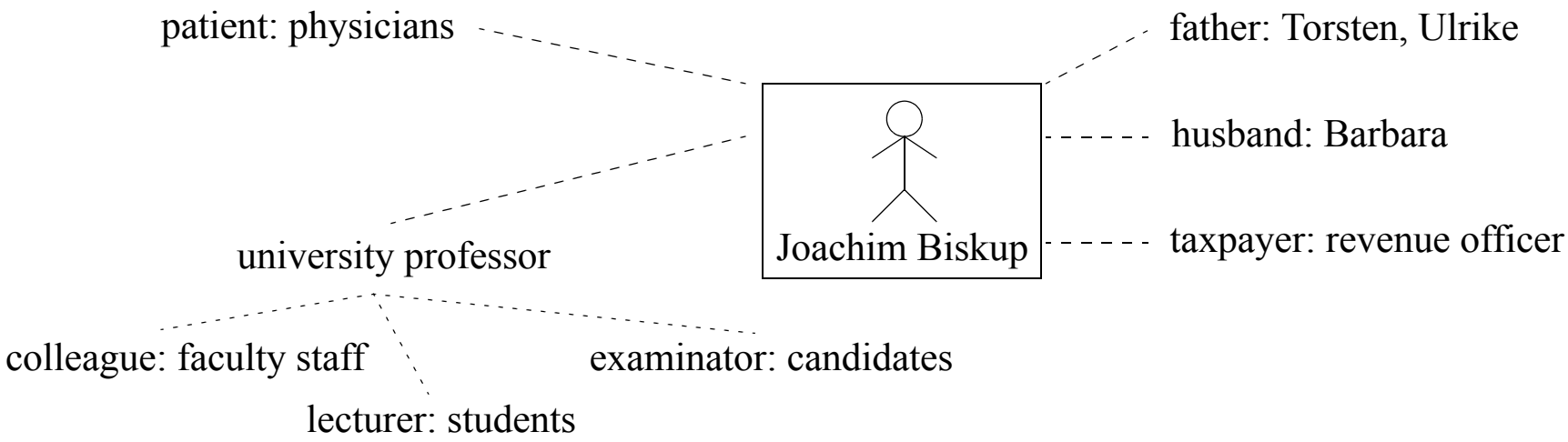
- whether, or to what extent,
  do security mechanisms *satisfy* the security requirements?
- which *assumptions* are underlying the evaluation?
- which kind of *trust* is assigned to participants or system components?
- do the *risks* recognized justify
  the *expenditure* for the security mechanisms selected?

Requirements by legislation: important examples

- *privacy acts detailing the principles of informational self-determination*
  first declare a general and protecting forbiddance,
  and then allow the processing of personal data under specific conditions
- *telecommunication and services acts*
  enable the public and commercial exploitation of informational activities, and
  lay foundations for legally binding transactions
  in public administration and private commerce
- *intellectual property acts*
  support and extend the traditional concept of
  authors' (or their publishers') copyright in texts or images
  to all kinds of electronic multimedia objects
- *criminal acts*
  identify definitely offending behavior within computing systems

Privacy and informational self-determination

- an individual determines by himself
  which personal *information* he is willing to *share* with group members
  in a specific social *role*



- an individual *selects* his social *roles* under his own responsibility
- other agents respect the intended *separation of roles*,
  refraining from unauthorized information flows between different roles

Protection rules for personal data

- *based on permission*:
  personal data should be processed only by permission,
  expressed in a law or with the explicit consent of the person concerned
- *need-to-know*:
  processing personal data should be restricted to actual needs,
  preferably by avoiding the collection of personal data at all or
  by converting it into nonpersonal data by anonymization
- *collected from the source*:
  personal data should be collected from the person concerned
- *bound to original purpose*:
  personal data should be processed only for the well-defined purpose
  for which it was originally collected

- *subject to inspection*:
  a person concerned should be informed about the kind of processing that employs his personal data
- *under ongoing control*:
  “wrong” personal data should be corrected;
  “no longer needed” personal data should be deleted
- *with active support*:
  agents processing personal data are obliged to actively pursue the privacy of the persons concerned

Requirements by security evaluation criteria

- *Trusted Computer System Evaluation Criteria* (TCSEC),
  known as the *Orange Book*,
  issued by the US Department of Defense

- *Information Technology Security Evaluation Criteria* (ITSEC),
  jointly published by some European countries

- *Common Criteria for Information Technology Security Evaluation* (CC),
  a version of which has also become an *ISO standard*

Common Criteria: security functionality

- *Audit*, as the basis of monitoring and analyzing the behavior of participants
- *Communication*, with an emphasis on providing evidence for sending and receiving of messages
- *User Data Protection*, with an emphasis on enforcing availability, integrity and confidentiality of the users' objects
- *Identification and Authentication*, for enforcing authenticity with non-repudiation and accountability
- *Privacy*, including non-observability, anonymity, pseudonymity and unlinkability
- *Protection of the Trusted Security Functions*, which deals with the installation, administration and operation of security mechanisms, i.e., how security mechanisms are securely protected in turn
- *Resource Utilization*, including fault tolerance, priorization and scheduling
- *Target of Evaluation Access*, including log-in procedures
- *Trusted Path/Channel*, dealing with the physical link between a (human) participant and the (processor of the) technical device employed

Common Criteria: evaluation assurance levels

- EAL1: functionally tested
- EAL2: structurally tested
- EAL3: methodically tested and checked
- EAL4: methodically designed, tested and reviewed
- EAL5: semiformally designed and tested
- EAL6: semiformally verified design and tested
- EAL7: formally verified design and tested

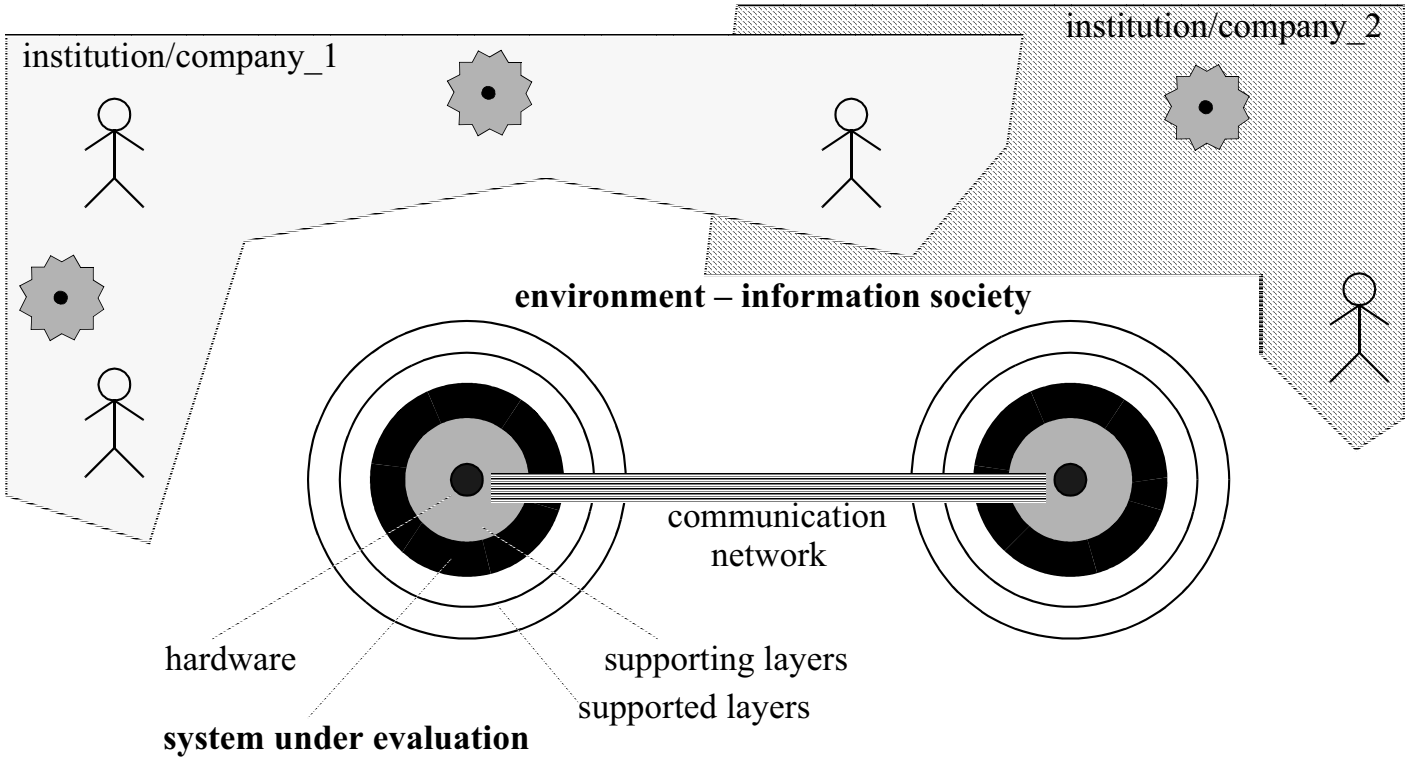
Common Criteria: top-level assurance classes

- Configuration Management
- Delivery and Operation
- Development
- Guidance Documents
- Life Cycle Support
- Tests
- Vulnerabilities

for each of the subclasses of the assurance classes,
appropriate assurance levels are required

A practical checklist for evaluations

- a comprehensive view of the circumstances



- answers to the following questions:
  - on what other *components*, in what layers, is the system based?
  - in what *environment* is the system embedded?
  - in what *institution* or *company* is the system used?

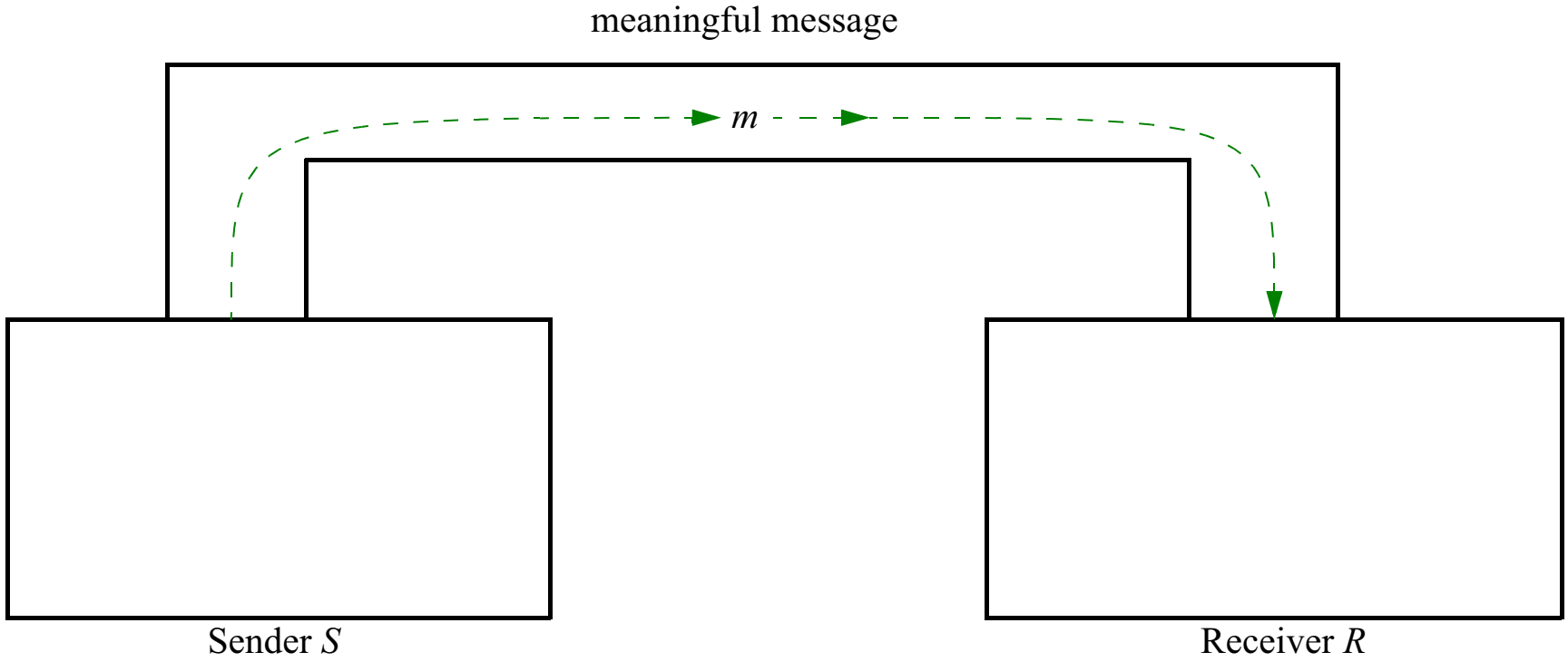
Issues for the actual version, configuration and circumstances

- ***security policy*:**
  are the security requirements explicitly expressed?
- ***authorization*:**
  is every access (execution of an operation by a subject on an object),
  preceded by an explicit permission
  (granting a corresponding access right/a suitable cryptographic key)?
- ***control*:**
  is such a permission controlled before execution,
  (by checking access rights/by the need for a suitable cryptographic key)?
- ***authenticity*:**
  is the authenticity of all items checked before the execution?
- ***monitoring*:**
  can intrusions be detected, though potentially only afterwards, and
  can any resulting damage be limited or compensated?
- ***total overage*:**
  do the security mechanisms cover all accesses and messages?

Construction principles

- ***open design*:**
  the design and the actual implementation of security mechanisms
  may or even must be made public (“no security by obscurity”)
- ***fail-safe defaults*:**
  any informational activity within a computing system is forbidden
  unless it has been explicitly permitted
- ***fine granularity*:**
  elementary, independent activity classes are defined as units of control
- ***need-to-know*/*need-to-act*:**
  permissions are granted only if they are strictly needed
- ***complete mediation*:**
  permissions are granted to well-defined *single* activity executions
- ***economy of mechanisms*:**
  the main burden of security enforcement is put on technical mechanisms
- ***complexity reduction*:**
  the security mechanisms are appropriately concentrated

Message transmission: a basic abstraction for challenges

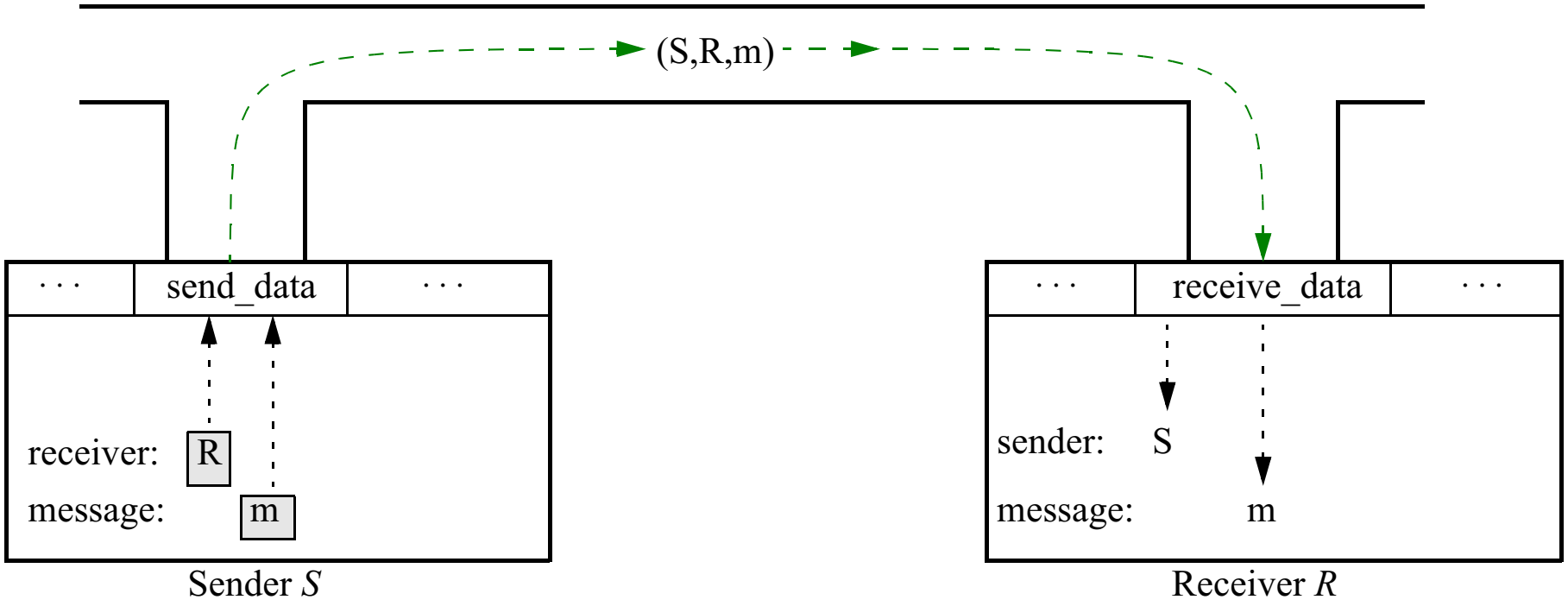


- captured by an *assignment* statement of the form `R:=S`
- the content *m* of the memory part denoted by *S*
  is transmitted to the memory part denoted by *R*
- *S writes* into *R*, or
  R reads from *S*, or
  some mechanism *pushes* the transmission

Transmission control in distributed computing systems: example

sender::send_data(receiver,message)

receiver::receive_data(sender,message)



Information flow

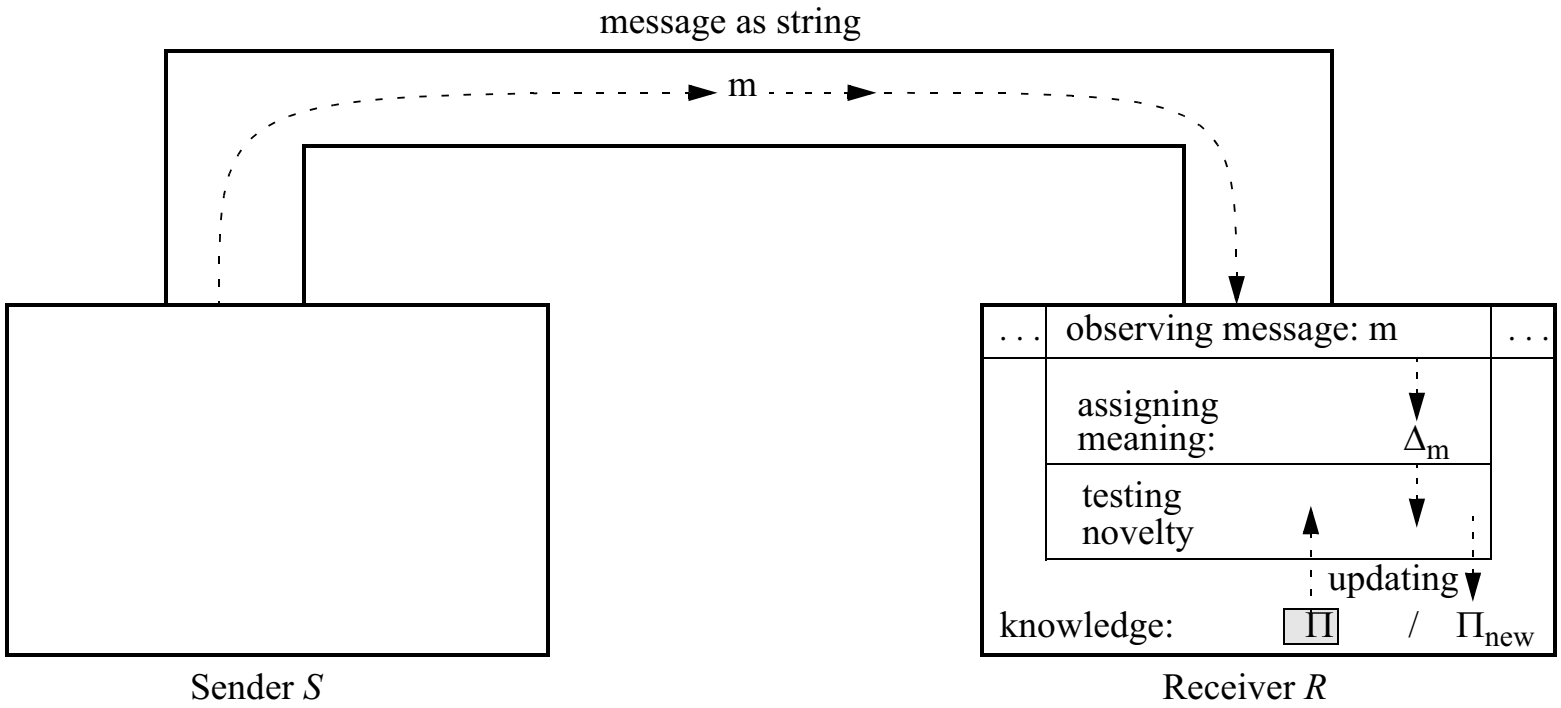
- a transmitted message,
  seen as a string (of letters and, ultimately, of 0's and 1's),
  is not necessarily *meaningful* concerning content
  for a receiver or any other *observer*

- it may happen and can even be sensible that
  an observed string appears random and without information:
  from the point of view of the observer,
  the message transmission has *not* caused an information flow

- in other cases, an observer succeeds in
  assigning a meaning to the observed string,
  roughly in the following sense:
  he determines an assertion
  expressing the truth of some aspect of his considerations;
  if, additionally, the observer has newly learnt this truth,
  then the message transmission has caused
  an *information flow* from the observer's point of view

Information flow based on message transmission

1. observing a message: consider a string m
2. assigning meaning: determine a sentence Δ_m
3. expressing knowledge: form presupposition Π as a collection of sentences
   testing novelty: infer whether Π implies Δ_m
   updating the knowledge: if novel (not implied), add Δ_m to Π and reorganize,
4. resulting in Π_{new}.



Information flow and message transmission

- a message transmission does not necessarily cause an information flow for any observer
- sometimes an observer has to infer implications in order to let a message transmission appear as an information flow from his point of view
- for such an inference, the observer can exploit a priori knowledge such as a previously acquired key
- for an actual inference, the observer needs appropriate computational means

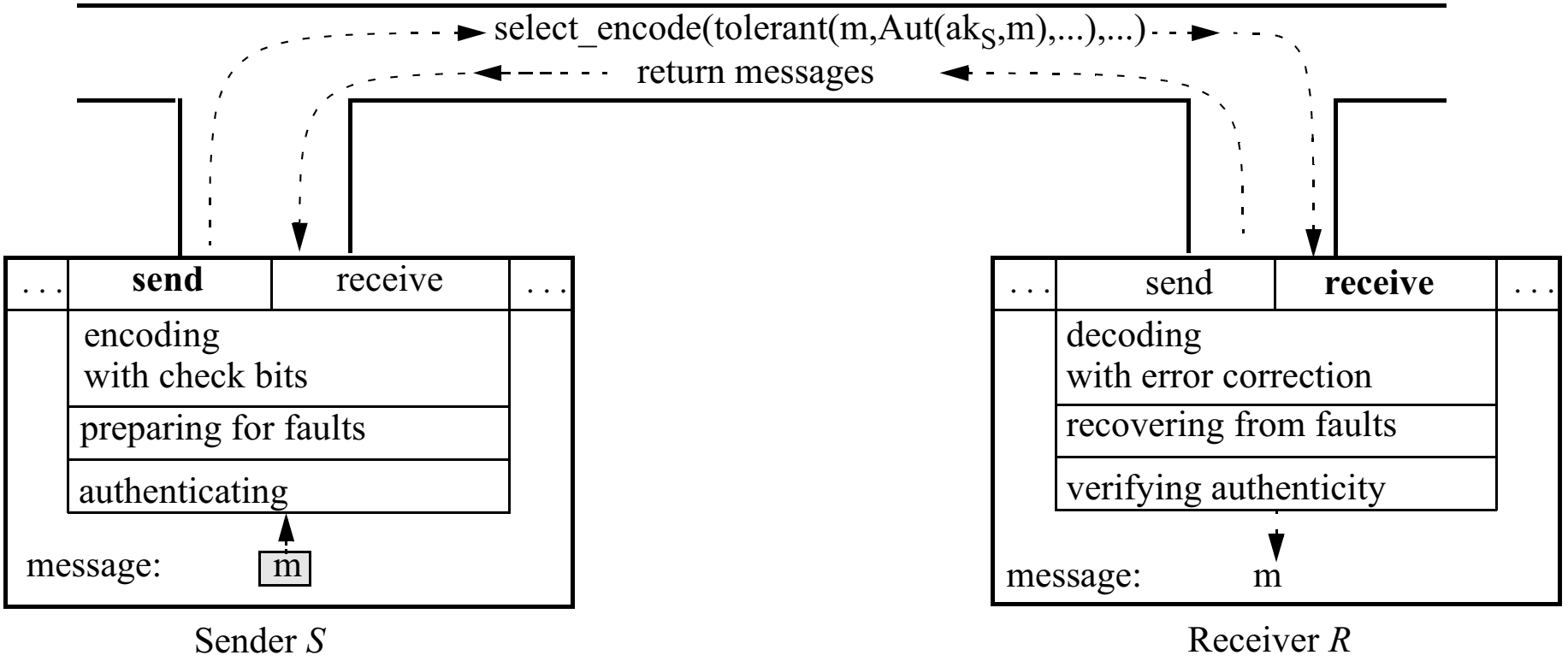
Inspection and exception handling: basic approach

- a message transmission can be accidently disturbed or deliberately distorted, with the effect that the receiver observes a modified or even forged message

- as a provision against such unfortunate events,
  senders generate redundancy in the form of *auxiliary objects*, in particular:
  - additional (check) bits for encoding
  - copies for fault-tolerant computing
  - cryptographic exhibits for authentication

- participants agree on protocols to exploit the redundancy, in particular:
  - to detect and correct errors for decoding
  - to detect and recover from faults for fault-tolerant computing
  - to detect forgeries for authenticity verification

Inspection and exception handling: summary



Security interests reconsidered in terms of message transmission/information flow

- each participant should express his *interests* with respect to the *service* considered (here: message transmission /information flow)

- some interests mainly expect *reliable correctness*, i.e., correct execution of the specified service even in the presence of threats, and maybe also additional evidence for actual executions

- other interests mainly require *confinement*, i.e., that nobody can misuse the service for unwanted effects

Threats: originators and causes

originators

- the interest holder himself
- participants directly involved in the service
- participants who have implemented the service
- other participants who are authorized to share the computing system
- intruders from outside
- manufacturers, vendors and administrators

originators might threat the service

- harmlessly and accidently
- maliciously and deliberately

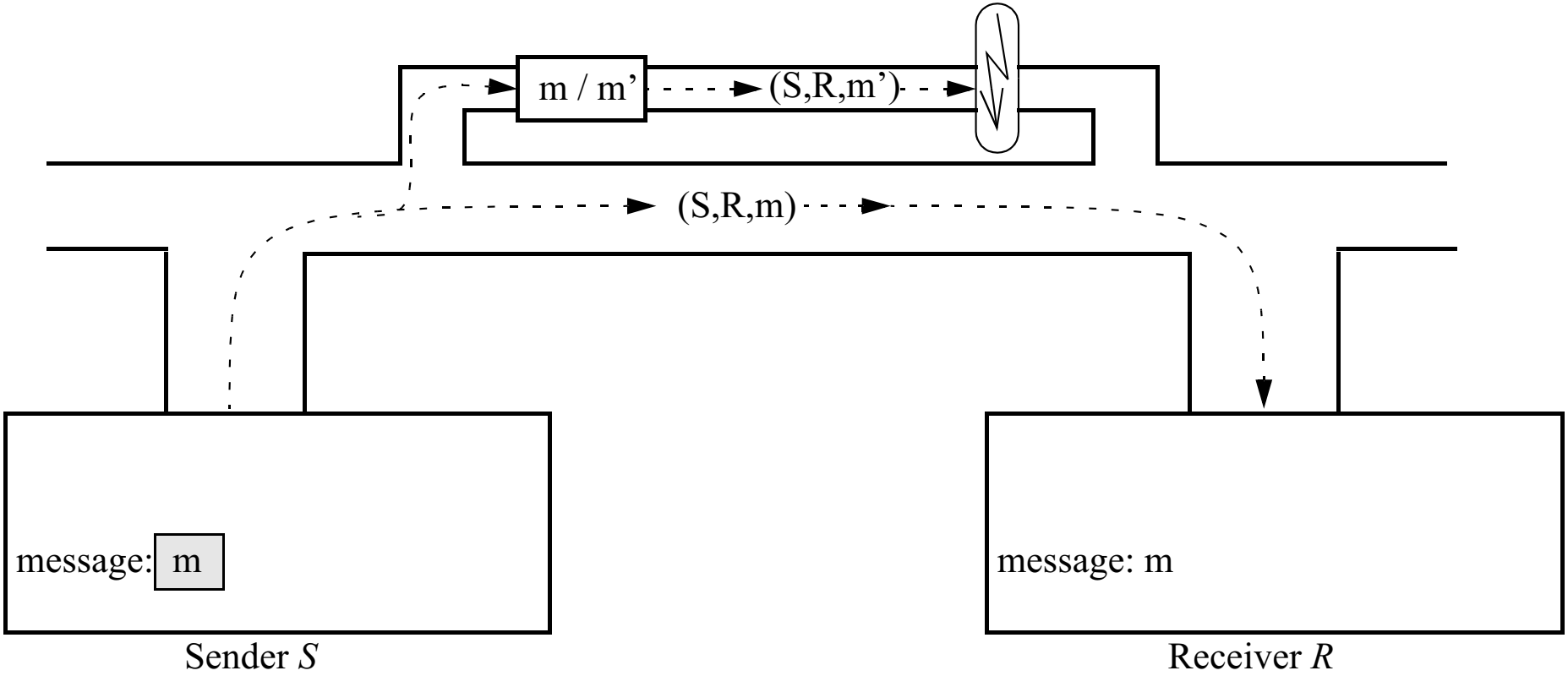
causes might range from

- improper requirements, through
- faulty implementations or
- wrong administration, to
- unfortunate external events

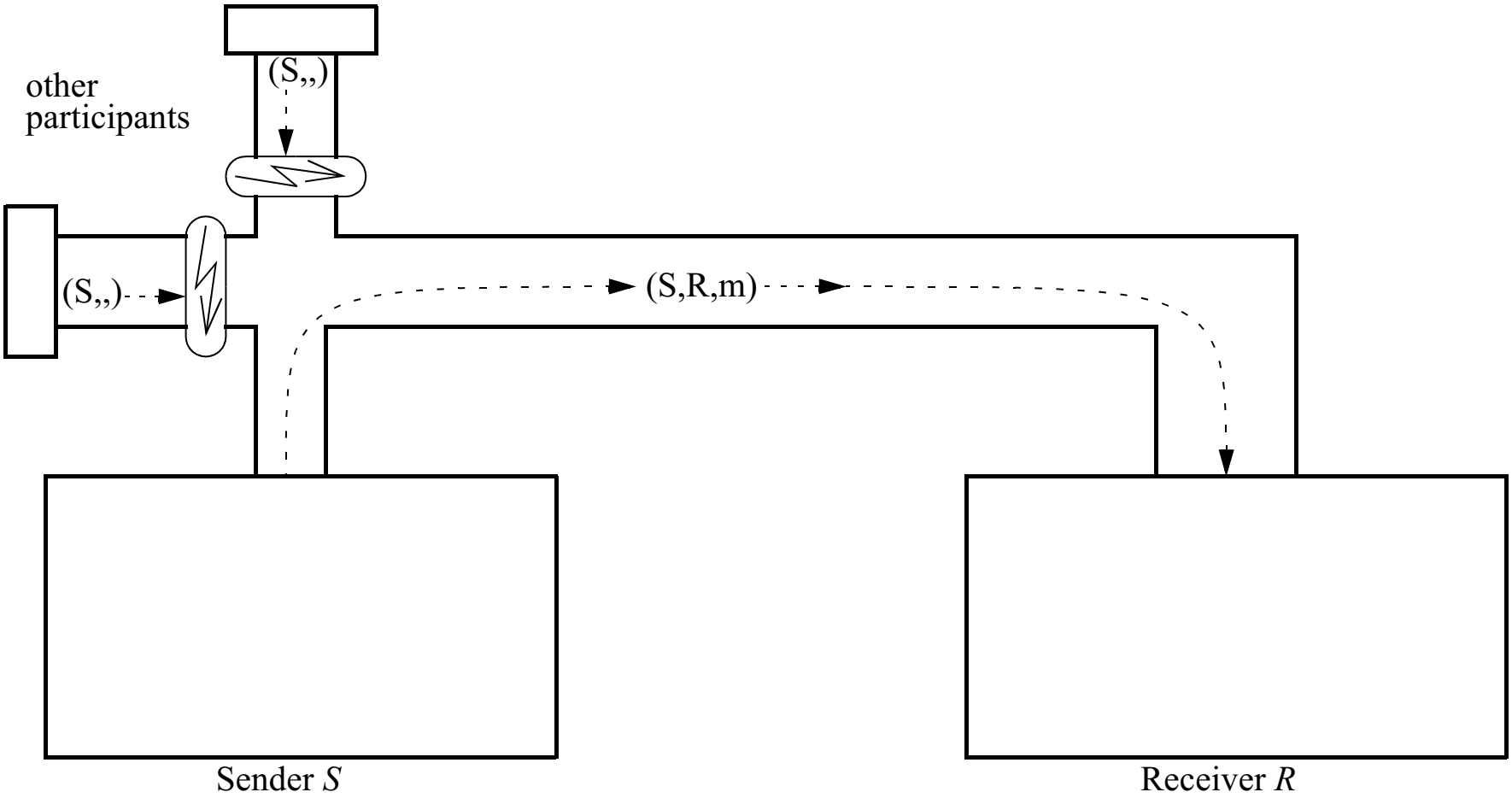
Security interests: an expanded list

- availability
- integrity: correct content
- integrity: unmodified state
- integrity: detection of modification
- authenticity
- non-repudiation
- confidentiality
- non-observability
- anonymity
- accountability
- evidence
- integrity: temporal correctness
- separation of roles
- covert obligations
- fair exchange
- monitoring and eavesdropping

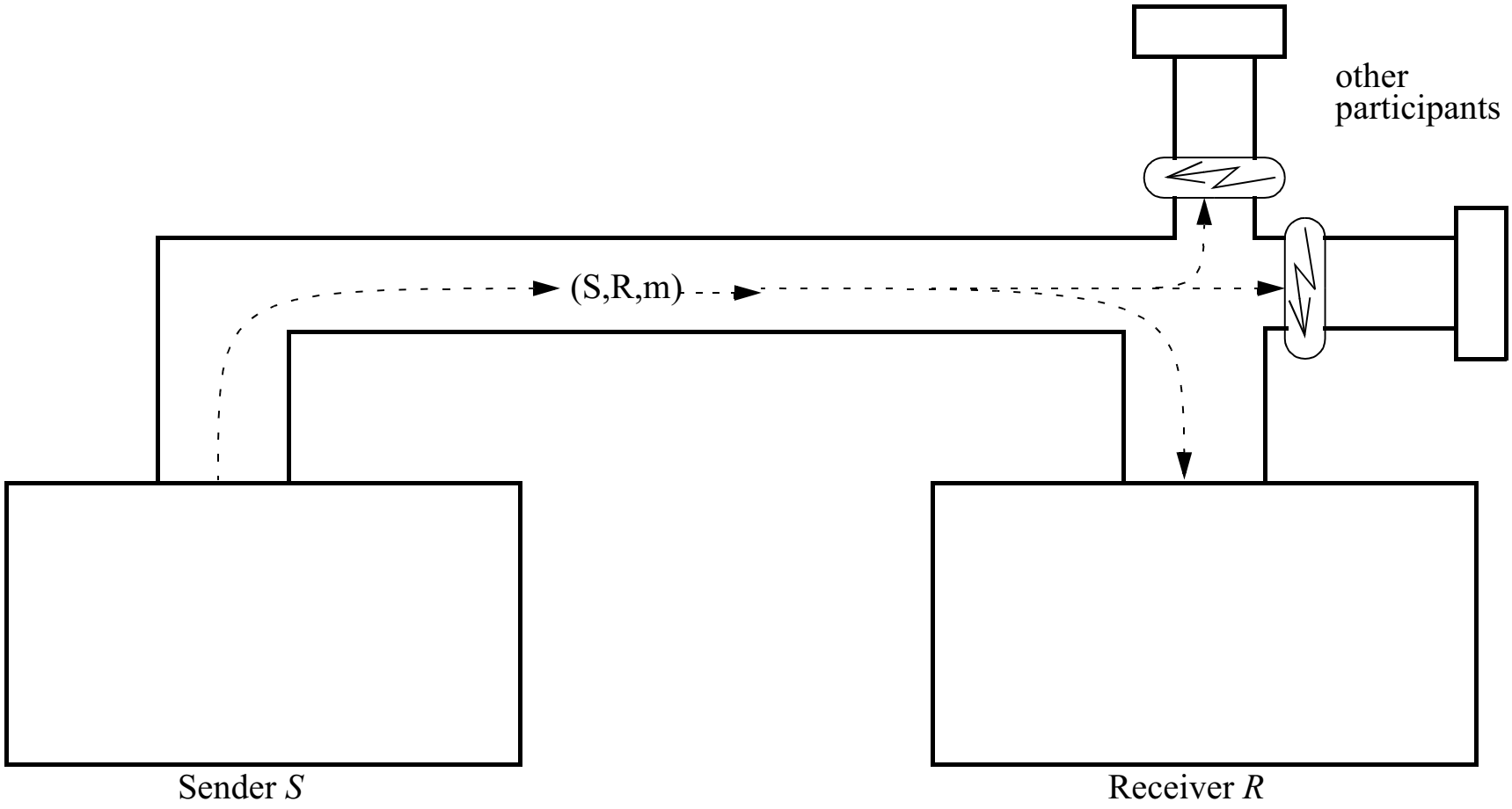
Integrity: unmodified state



Authenticity



Confidentiality



Autonomy and cooperation: a classification of security interests

Interest	Autonomy	Cooperation
availability	•,+	+,•
integrity: correct content	•	•
integrity: unmodified state	•	•
integrity: detection of modification	+	•
authenticity	+	•
non-repudiation	•,+	+
confidentiality	+	•
non-observability	+	•
anonymity	+	•
accountability	–,•	+
evidence	•,+	+
integrity: temporal correctness	+	•
separation of roles	+	•
covert obligations	+	•
fair exchange	+	•
monitoring and eavesdropping	–	+

Trust and threats

- while interacting, one participant might see another one
  both as a wanted partner
  and as a potentially *threatening* opponent

- at least some limited *trust* has to be assigned to some participants involved

- components of a computing system might fail,
  but a user has to trust at least some components

Crucial points of multilateral security

- the trust needed should be minimized
  while simultaneously maximizing the achievable functionality,
  thereby facing the potential threat from the untrusted parts

- each participant should autonomously assign trust at their own discretion

- as far as possible,
  assigned trust should be justified, and

  the assigning participant should have the power
  to verify the trustworthiness and
  to control the actual behavior of the trusted realm

Confident and optimistic approach

the administrator chooses relatively weak security mechanisms, roughly expecting the following:

- at relatively low cost,
- only slightly affecting the standard operations,
- most of the anticipated threats are effectively covered,
- but exceptional violations (hopefully rare) might still be possible;
- such violations will, hopefully, manageable or acceptable,
- though potentially at high cost

Provisional and pessimistic approach

the administrator selects relatively strong security mechanisms, roughly expecting the following:

- at relatively high cost,
- greatly affecting the standard operations,
- all anticipated threats are effectively covered

Optimistic approach versus pessimistic approach

- cheap *versus* expensive
- basically unaffected standard operations *versus* an essential security overhead
- approximate *versus* complete coverage of threats
- toleration *versus* strict avoidance of exceptional violations

example: access control

optimistic: we audit all activities and,
taking random samples or in cases of suspicion,
analyze the audit trail for violations only afterwards

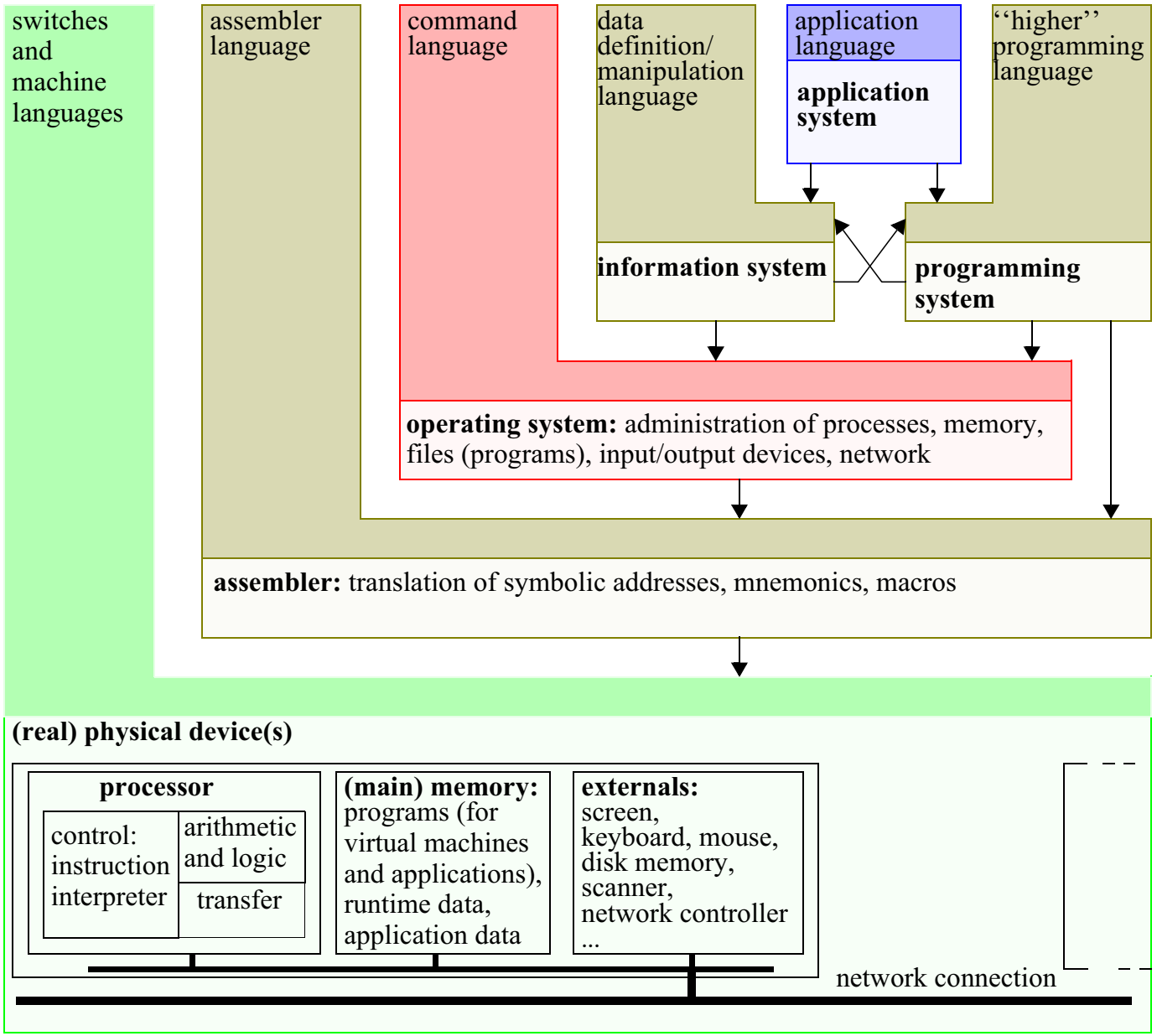
pessimistic: we fully control all requests for activities and
decide them in advance

example: trading

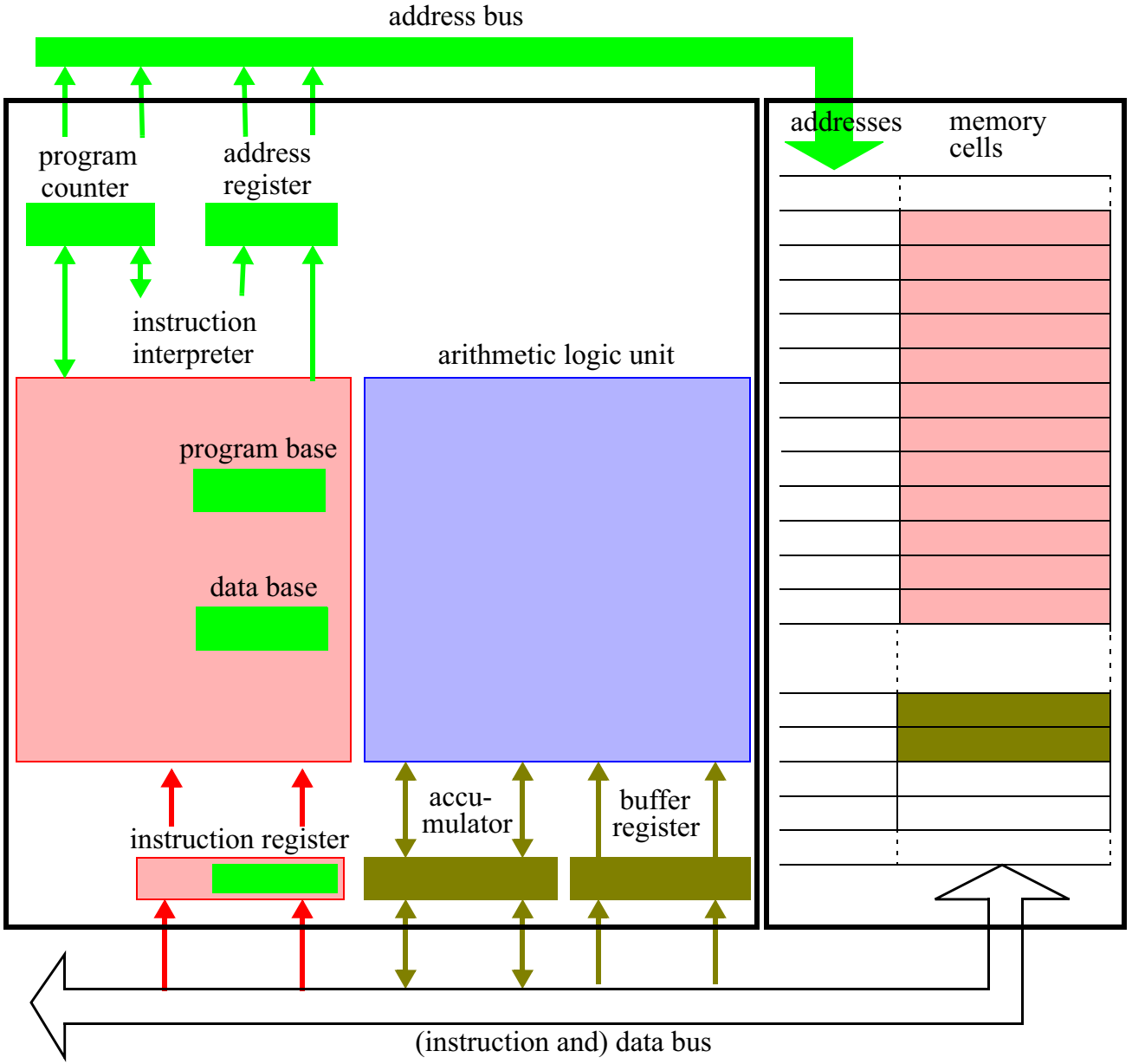
optimistic: cooperating participants issue exhibits by themselves,
which are subject to later evaluation by a trusted third party
only in the case of disputes

pessimistic: every trade is mediated and supervised by a trusted notary

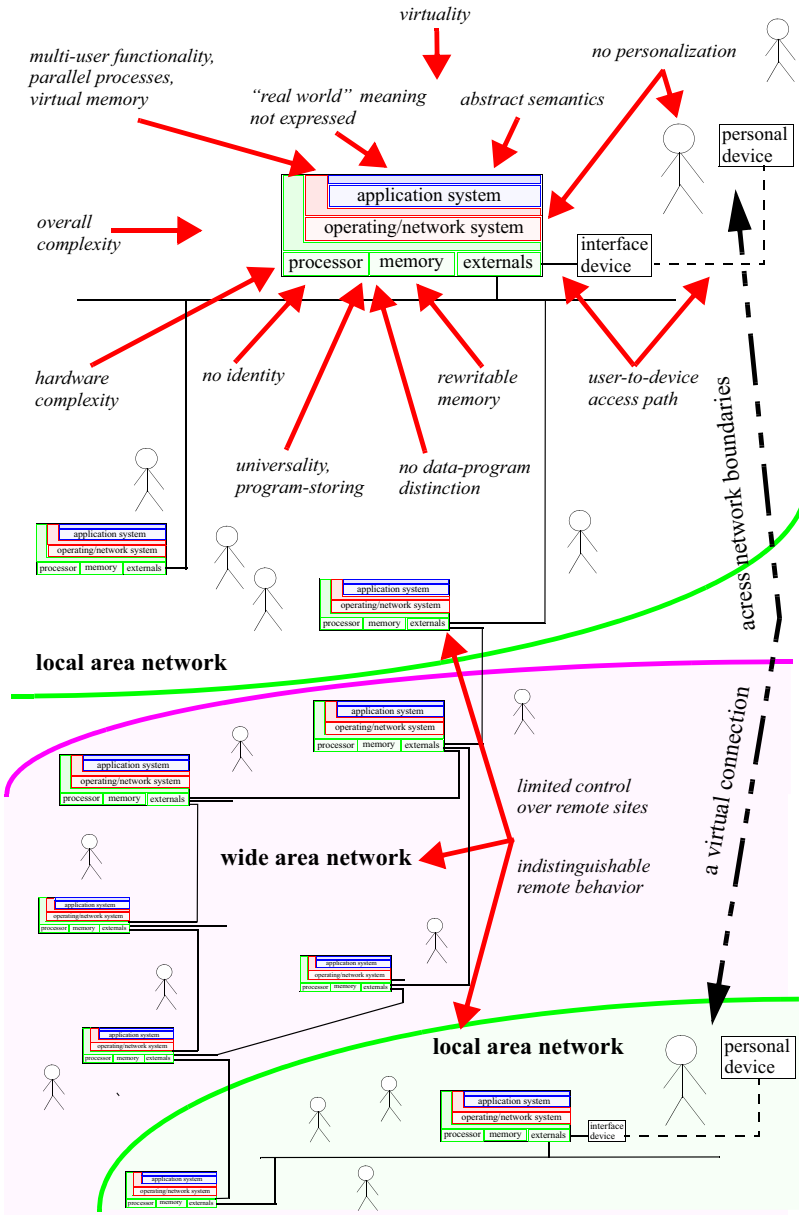
Computing system: layered design



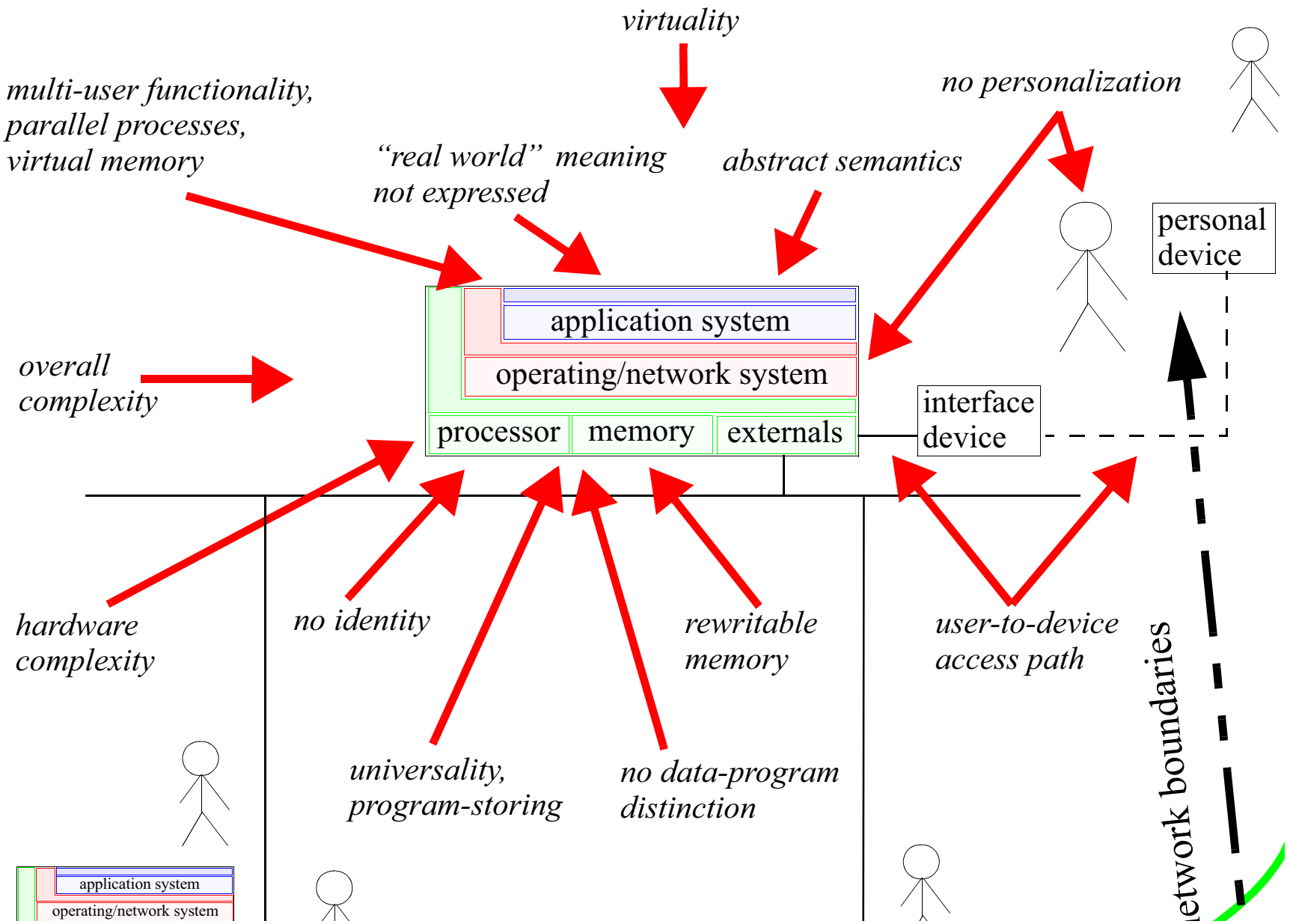
Internal structure of a processor and its memory



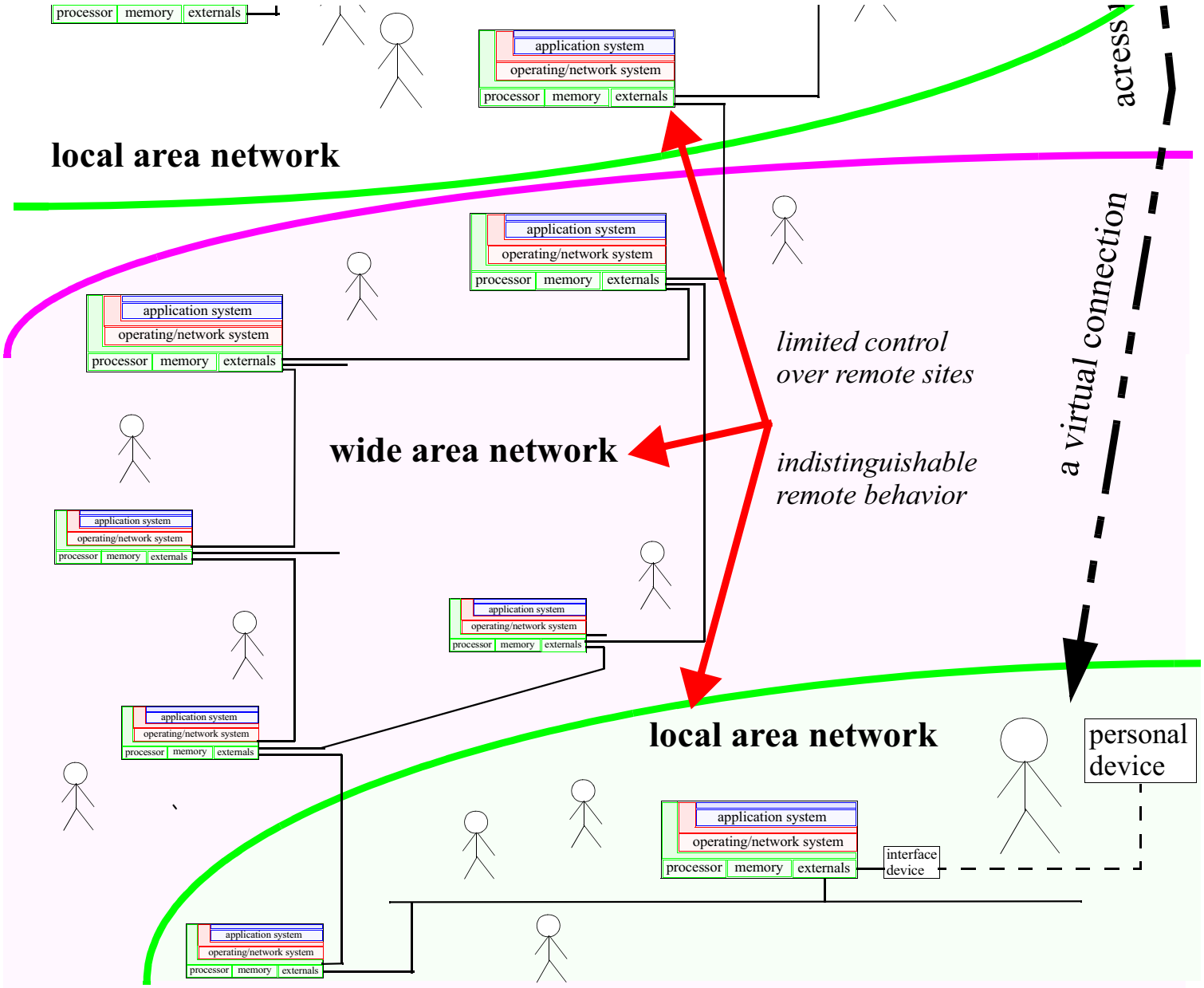
Features of computing and basic vulnerabilities: overview



Features of computing and basic vulnerabilities: one component



Features of computing and basic vulnerabilities: networks



Features and vulnerabilities

virtuality	“virtual security” corrupted or circumvented in supporting layers
overall complexity	no global, complete understanding; unexpected interferences
universality, program-storing	imposed (malicious) “computable will”
processors without identity	masquerades
devices without personalization	masquerades, repudiated human–device binding
no data–program distinction	program (self-)modification (buffer overflow attacks)
rewritable memory	program and data modification
hardware complexity	hidden functionality, covert channels
user-to-device access path	exposed attack target
multi-user functionality, parallel processes and virtual memory	unintended interferences by resource sharing
abstract semantics of virtual layers	incorrect translation, non-captured but security-relevant aspects
“real-world” meaning not expressed	unperceived attack possibilities
seemingly restricted functionality	universality by simulation
(identifiable) virtual digital objects represented by bit string	unauthorized copying (double spending of coins)
limited control over remote sites	remote activities only derivable by inferences
indistinguishable remote behavior	eavesdropping, message manipulation and forgery, (malicious) message production

2 Key Ideas and Combined Techniques

Key ideas for technical security enforcement mechanisms

- ***redundancy***
  enables one
  to infer needed information,
  to detect failures and attacks and
  to recover from such unfortunate events

- ***isolation***
  prevents unwanted information flows or interference

- ***indistinguishability***
  makes maliciously planned observations
  appear random or uniformly expected
  and thus useless

Redundancy: important examples

- spare equipment and emergency power
- recovery copies for data and programs
- deposit of secrets
- switching networks with multiple connections

- fault-tolerant protocols:
  - *infer* a hidden original state from observations and auxiliary redundancy and *reconstruct* it accordingly
  - *abort* a failing operation and *restart* it from a saved or reconstructed previous state, or even to *redo* a completed operation
  - take a *majority vote* regarding the actual outputs of computations performed independently and in parallel

- error-detecting and error-correcting codes

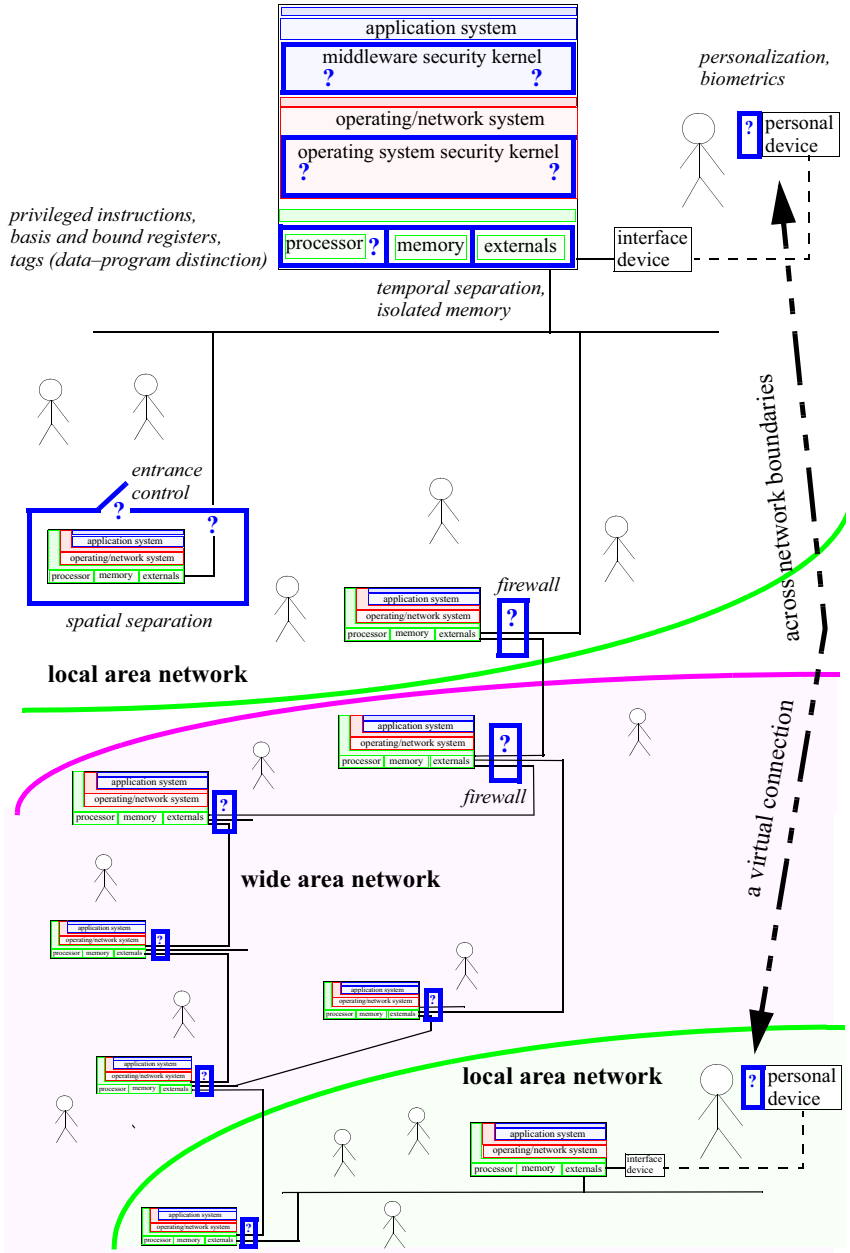
- cryptographic pieces of evidence

Isolation

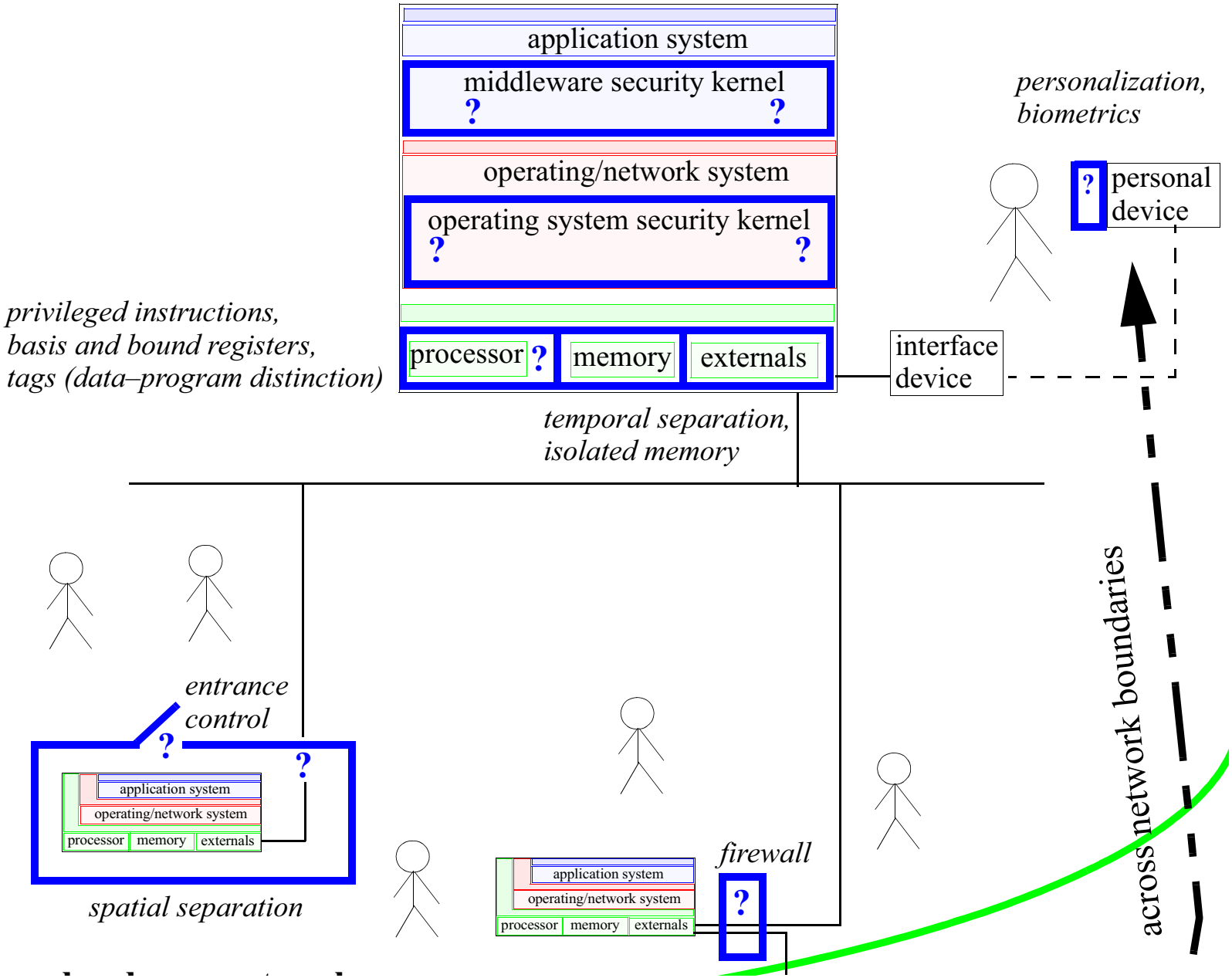
- **physical/programming-based** isolations
  requiring explicit *access decisions* at runtime,
  in order to enable the restricted usage of the isolated components
  according to declared *permissions*

- **virtual cryptographic** isolations
  employing more implicit access decisions
  based on the distribution of secret keys

Physical/programming-based isolations: a global view



Physical/programming-based isolations: a local view



Spatial separation and entrance control

- *spatially separate* an autonomously operated, *stand-alone* computing system in a dedicated closed room with locked doors (and windows)

- operate an effective *entrance control* enabling only *authorized individuals* to enter and then to (unrestrictedly) use the system

- may suffer from serious threats:
  - authorized individuals might not match the interests, owing to organizational weaknesses or unresolved conflicts
  - two or more authorized individuals might (unrestrictedly) interfere and collaborate
  - an (unrestrictedly) authorized individual might misuse the trust for unexpected and unwanted goals
  - the entrance control might fail, and some unauthorized individual might then (unrestrictedly) exploit the system

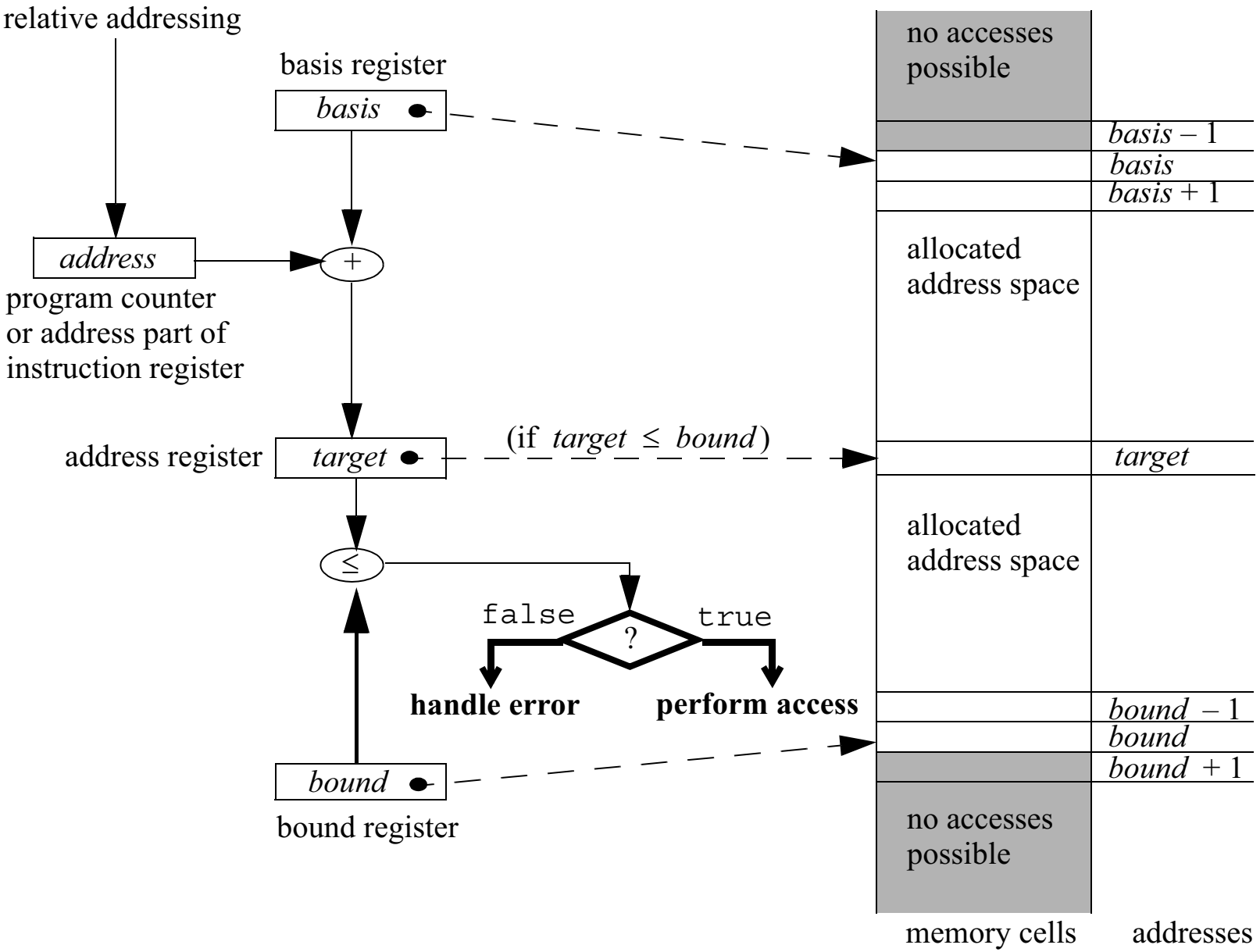
Temporal separation and isolated memory

- several participants can share a computing system
  either *strictly in sequence* or *overlapping in time*
- the participants might then interfere,
  when the processes executed on behalf of them access common memory
- if sharing is done strictly in sequence, after finishing a job,
  completely *erase* all memory contents,
  i.e., reestablish an agreed *normal state*,
  maintained as an *invariant* of any usage of the computing system
- if sharing is done so that there is overlapping in time,
  adapt the notion of a normal state and take additional measures:
  - ensure that the allocated *process spaces*
    (containing programs to be executed, runtime stacks, heaps, etc.)
    always remain strictly isolated:
    one process can never access memory locations
    currently reserved for a different process
  - ground these measures on *physical tamper-resistant* mechanisms

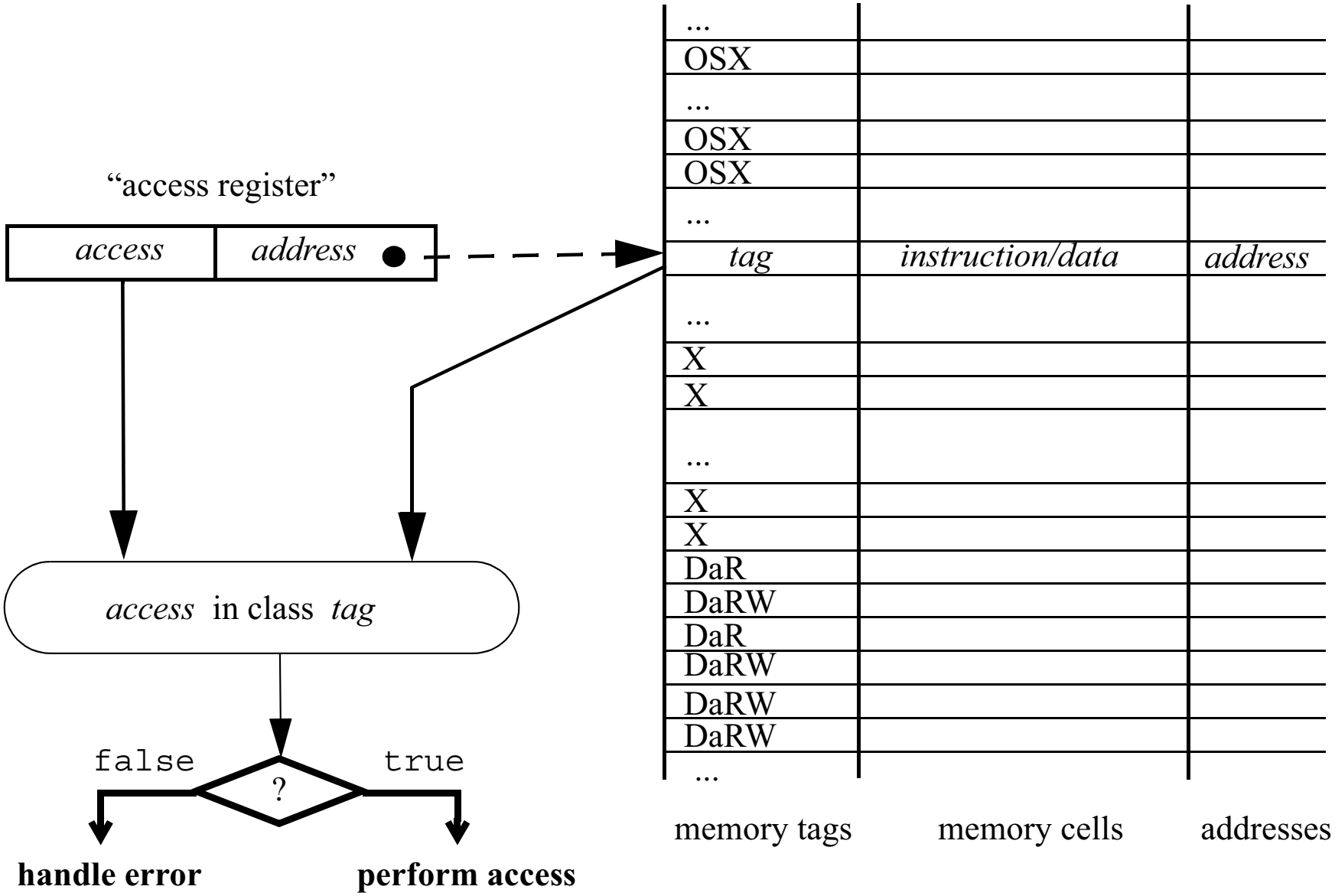
Memory protection and privileged instructions

- *memory protection* physically restrict memory accesses with respect to
  - addresses and
  - the mode of the operation requested
- ensured behavior of the processor's instruction interpreter:
  if the next instruction must be fetched from a memory location *address* or
  a machine instruction of the kind *instr* = [*operation* , *address*] is considered,
  then the request is actually executed
  iff
  a specific *protection condition* is satisfied
- a protection condition might depend on
  - the process,
  - the activity requested and
  - the address referred to

Basis register and bound register



Memory tags


...
OSX
...
OSX
OSX
...
tag
instruction/data
address
...
X
X
...
X
X
DaR
DaRW
DaR
DaRW
DaRW
DaRW
...
memory tags
memory cells
addresses
"access register"
access
address
access in class tag
false
?
true
handle error
perform access

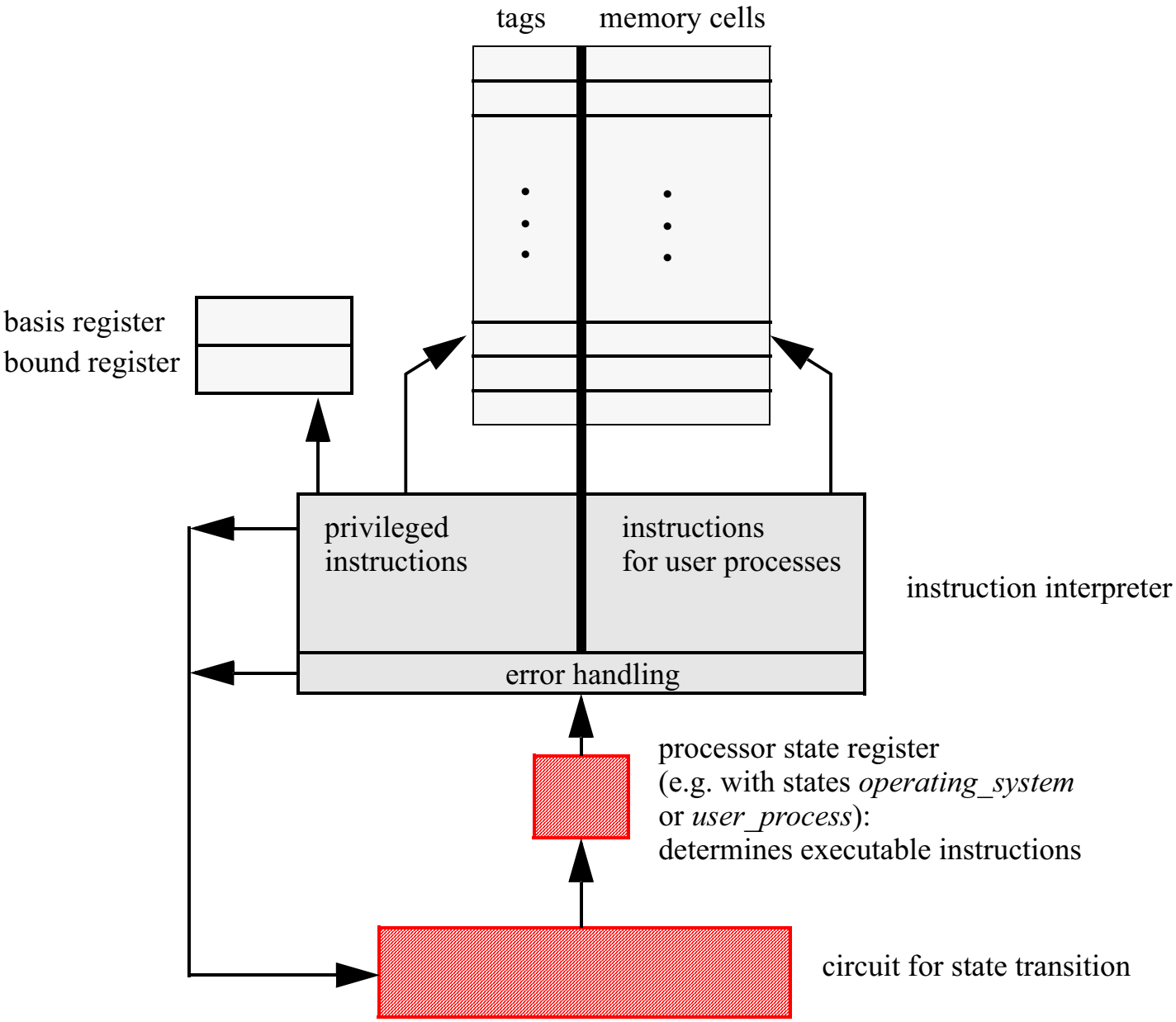
Tags as usage classes: examples

- *read* access to an executable *instruction*
  (fetching into the instruction register)
  by any *user process* or by special *operating system processes*
- *read* access to *arbitrary data*
  (loading into a data register)
  by any *user process* or by special *operating system processes*
- *write* access with *arbitrary data*
  (storing from a data register)
  by any *user process* or by *special operating system processes*
- *read* access to *data of a specific type*
  (e.g., integer, string, address or pointer),
  which has to be suitably recognized by the context or other means
- *write* access with *data of a specific type*
  (e.g., integer, string, address or pointer),
  which has to be suitably recognized by the context or other means.

Basis register and bound register versus memory tags

	Basis register and bound register	**Memory tags**
Extra memory	2 registers	linear in the size of memory
Operational overhead	assigning the registers; calculating and comparing addresses during memory accesses	assigning the memory tags; checking conformance during memory accesses
Abstraction layer of separated items	dynamically allocated address spaces	instances of types known to the processor
Granularity	more coarse (according to the memory requirements of dynamically generated, active items)	more fine (according to the size of instances of static types)
Protection goal primarily achieved	isolation of active items for avoiding unintended sharing of memory	isolation of instances of types for avoiding unintended usage
Coordination with higher layers	relative addressing, as usually employed	mapping of more application-oriented types to usage classes denoted by tags
Deployment	widespread, mostly together with other mechanisms of indirect addressing	seldom, mostly only in a simple variation

Privileged instructions



Further isolation mechanisms

- separate process spaces
- object-oriented encapsulation
- security kernels
- stand-alone systems
- separate transmission lines
- security services in middleware
- firewalls
- cryptographic isolation

Indistinguishability

- blurs specific informational activities
  by making them *indistinguishable* from random or uniformly expected events

- thus prevents an unauthorized observer to infer
  the details or even the occurrence of a specific activity

- might be achieved by employing
  - *randomness* or
  - *standardized behavior*

Indistinguishability by randomness

some explicit *randomness* is generated,

and then the specific activity considered
has this randomness *superimposed* on it

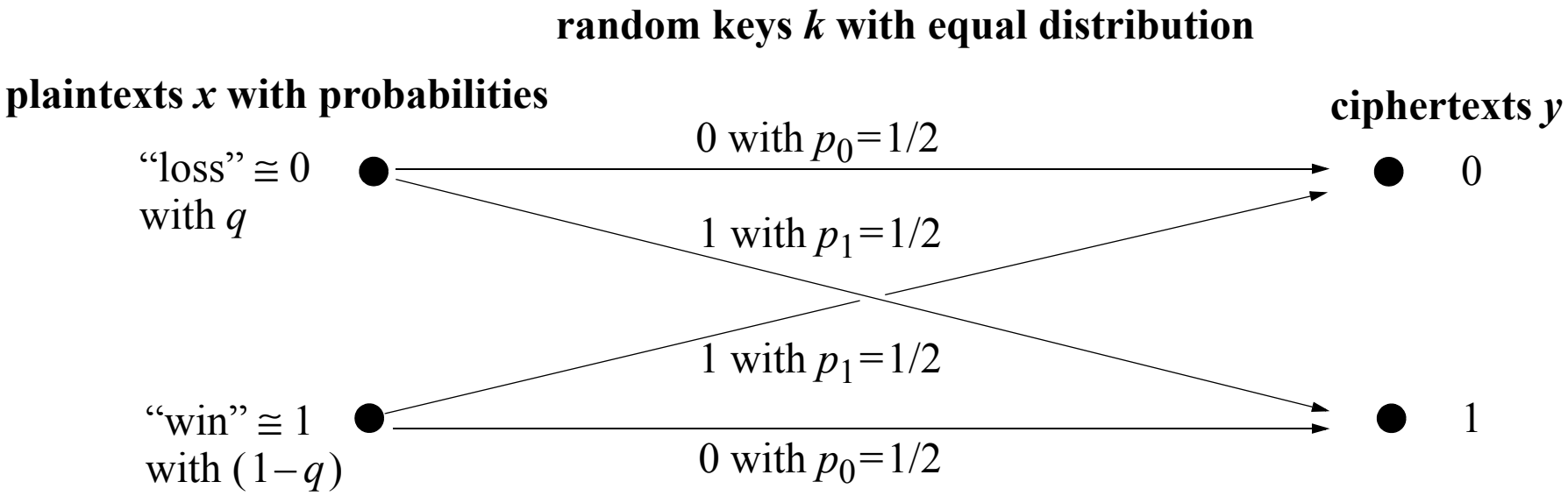
such that the activity appears (sufficiently) random itself

used in cryptography: the *secret key* is randomly selected
from a very large number of possibilities,

and the randomness of the secret key is transformed
into (some sufficient degree of) randomness
of the activity to be protected

Example for superimposing randomness: encryption

- two possible plaintexts: 0 with probability q
  1 with probability $1-q$
- source of *randomness*: two equally distributed keys, 0 and 1,
  with probability 1/2, independently of the plaintext
- the *randomness* of the keys is then *superimposed* on the plaintexts:



$$Prob[y=0] = Prob[x=0]\cdot Prob[k=0] + Prob[x=1]\cdot Prob[k=1]$$
$$= (q+(1-q))\cdot\frac{1}{2} = \frac{1}{2}$$

Encryption: indistinguishability of plaintexts

described in terms of a mental experiment:

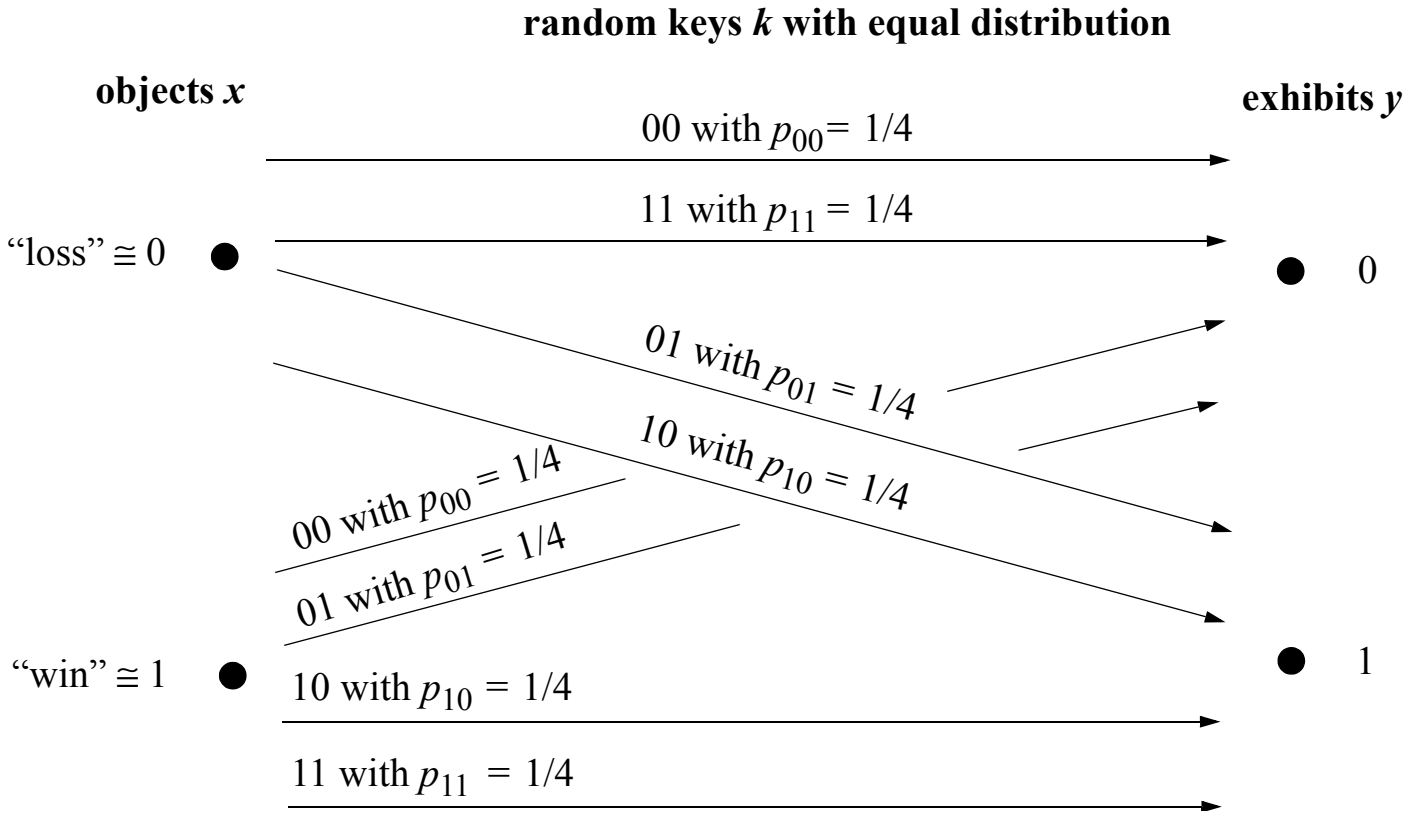
- attempt: construct an efficient *accepting device*
  that discriminates (hidden) plaintexts
  on the basis of observing (visible) ciphertexts

- insight: such a device *cannot* exist:
  an observed ciphertext does not contain
  any information about the underlying plaintext,

  thus this plaintext and the alternative one
  remain completely *indistinguishable*

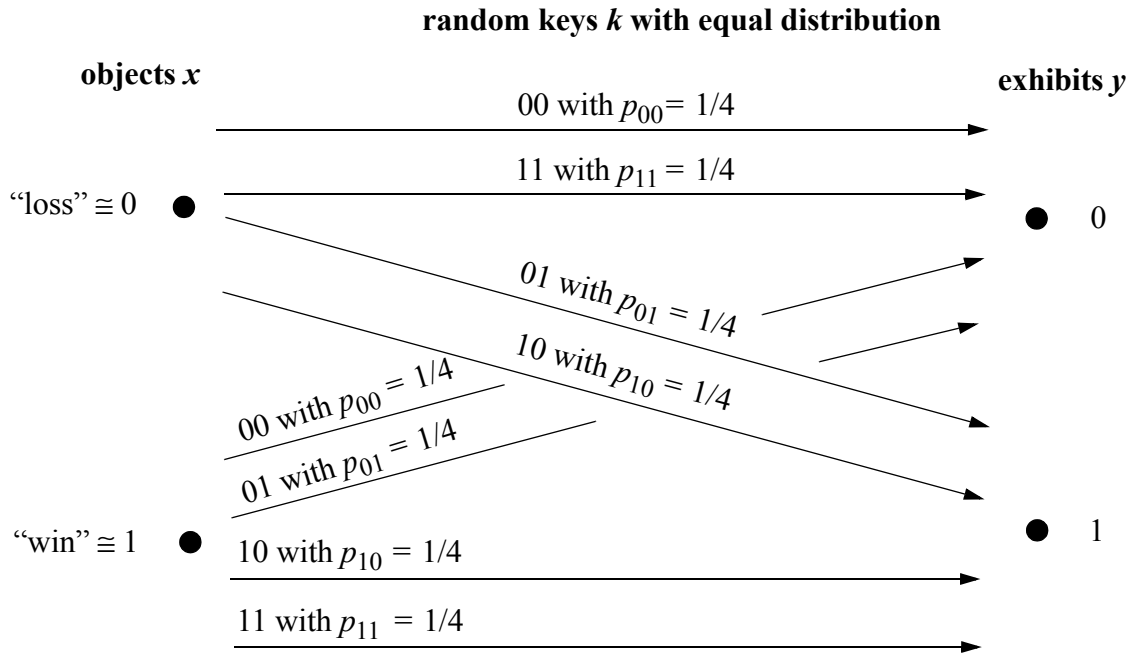
Example for superimposing randomness: authentication

- two possible objects, 0 and 1
- source of *randomness*: four equally distributed keys, 00, 01, 10 and 11, each of which is used with probability 1/4, independently of the object
- the *randomness* of the keys is then *superimposed* on the objects:



Authentication: indistinguishability of exhibits

- suppose the exhibit 0 for the event “loss” is known
- then, either key 00 or key 11 has been secretly used:
  these keys still map the event “win” onto either exhibit, 0 or 1,
  which are thus *indistinguishable*
  regarding their acceptance on the basis of the pertinent secret key



Indistinguishability by standardized behavior

a suitably designed *standardized behavior*,
possibly consisting just of dummy activities, is foreseeably produced,

and then the specific activity considered is hidden
among the foreseeable behavior,
for instance by replacing one of the dummy activities

Hiding among standardized behavior: examples

- **non-observable activities**
  hiding the points in time of *sending* a message
  by pretending to be *uniformly active*:
  - participant actually communicate with some partner:
    prepares a corresponding document,
    appropriately adds the final destination of the communication,
    pads the document with some additional material until it has the expected length,
    envelops all data,
    waits for the next agreed point in time, and
    then sends the final message to the intermediate address used as a postbox
  - participant wants no "real activity":
    just sends a *dummy message* of the expected length

- **brokers and blackboards**
  employing a sort of fixed intermediate postbox to hide
  the sources and the final destinations of communications

- **group activities**
  authorizing group members to act on behalf of the community
  but without revealing the actor's *identity* to observers outside the group

Combined techniques: overview

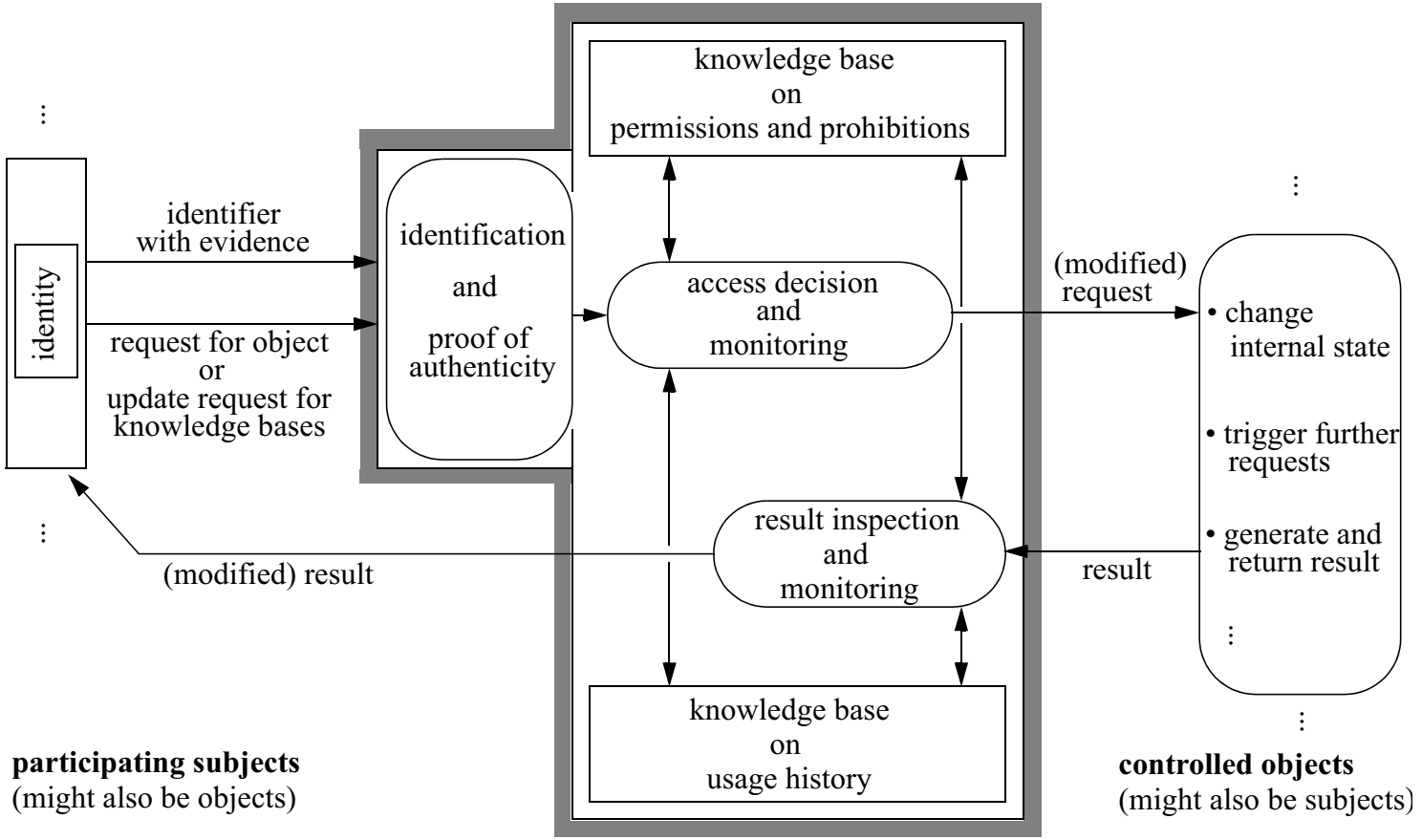
- ***control* and *monitoring*:**
  identifiable agents can have *access rights* granted and revoked, and
  access requests of authenticated agents are intercepted by *control components*
  that decide on allowing or denying an actual access

- ***cryptography*:**
  secrets are generated and kept by agents:
  the secrets are exploited as cryptographic *keys*,
  distinguishing the key holder so that
  that agent is enabled to execute a specific operation in a meaningful way,
  in contrast to all other agents

- ***certificates* and *credentials*:**
  digitally signed digital documents (*digital legitimations*),
  conceptually bind *properties* that are relevant for access decisions
  to specific agents, which are denoted only by *public keys*

  (here, a public key is understood as a suitable reference to
  a *private* (*secret*) cryptographic *key* held by the agent considered)

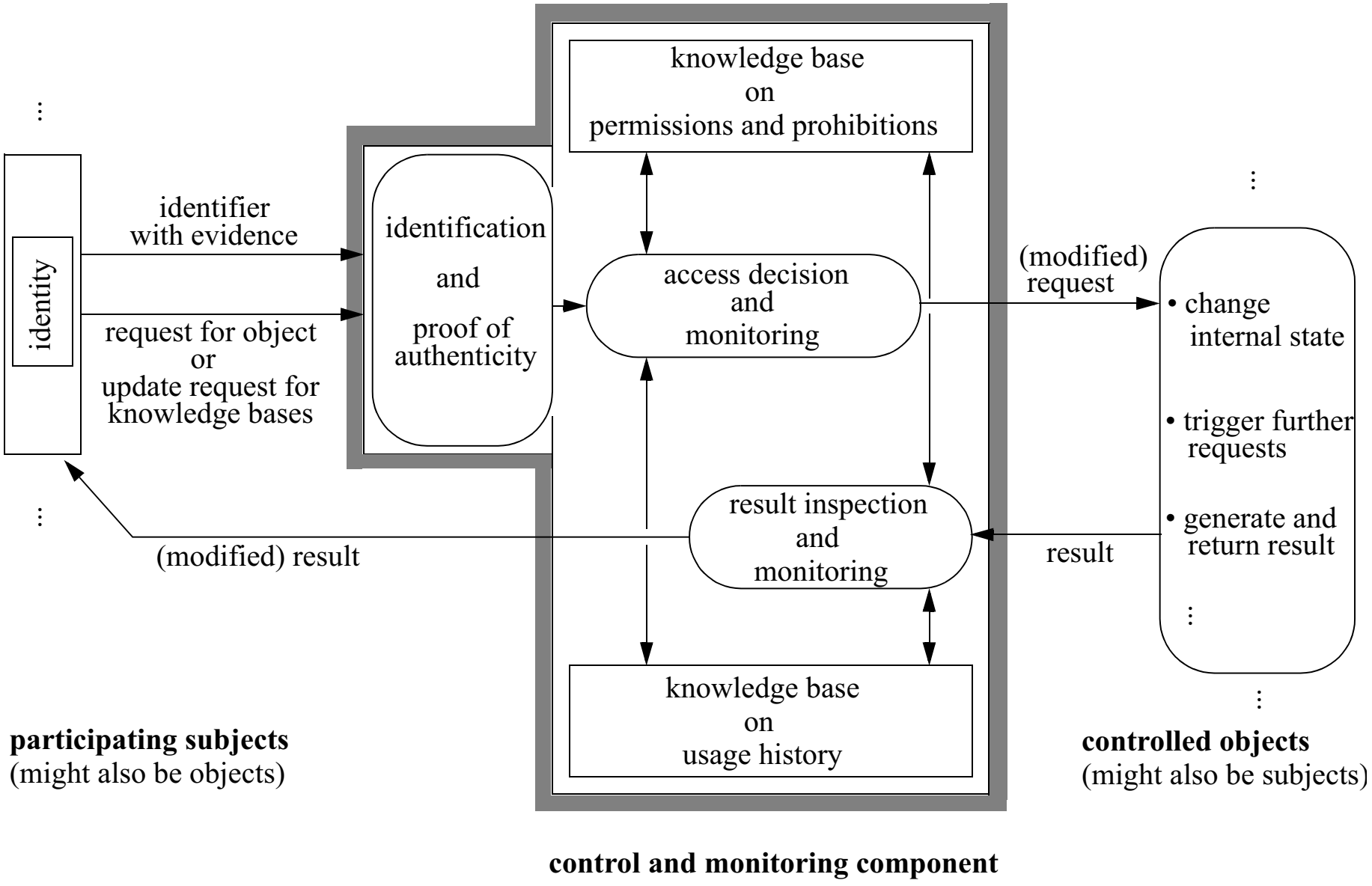
Local control and monitoring



control and monitoring component

- cannot be bypassed
- (virtually) isolates participating subjects from controlled objects
- is based on physical isolation (indicated by the gray frame)
- decides on requests and results and possibly modifies them

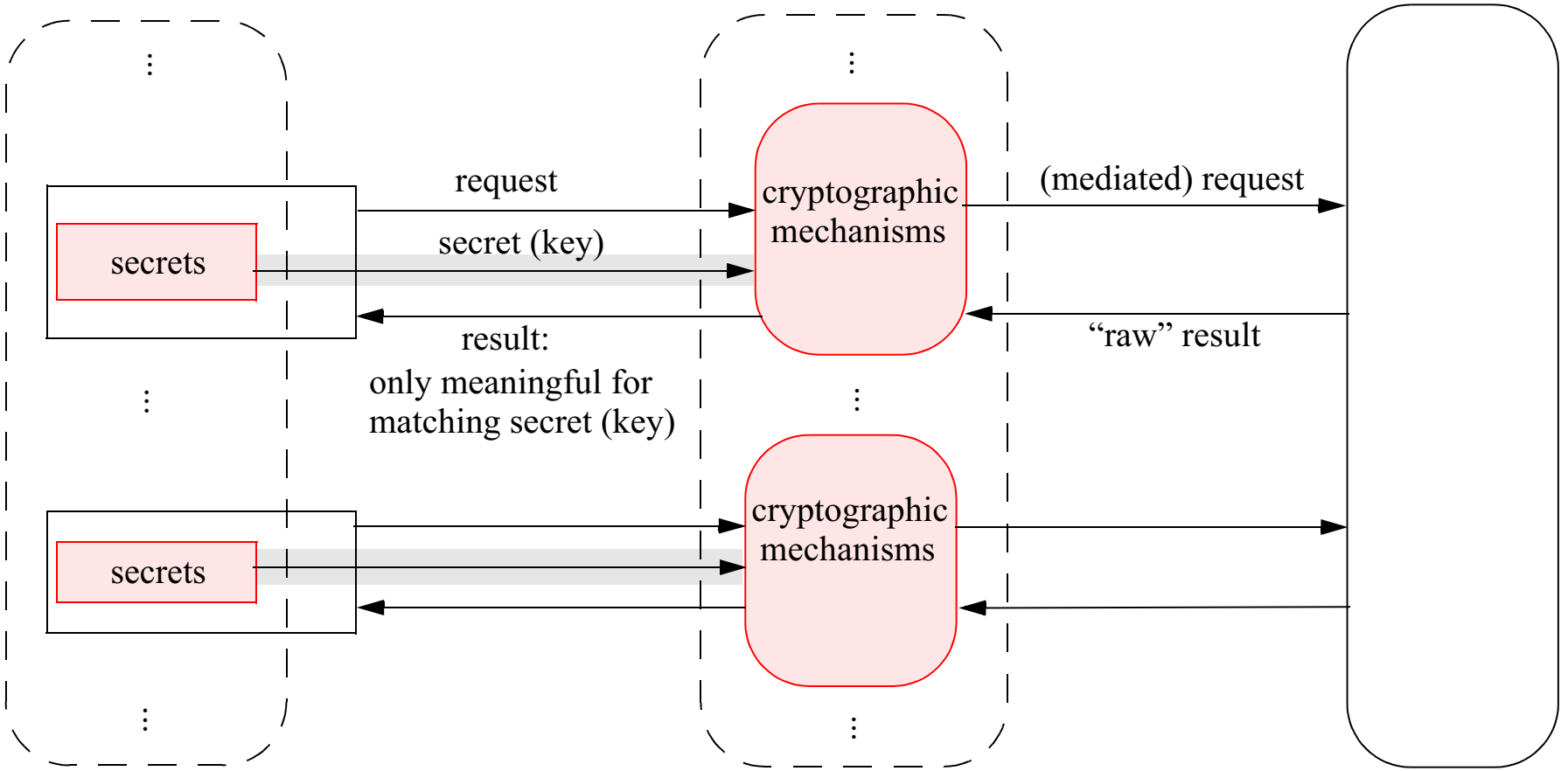
Local control and monitoring



Cryptography

conceptually: (cryptographic)
knowledge base on permissions
(and prohibitions)

conceptually: (cryptographic)
control component

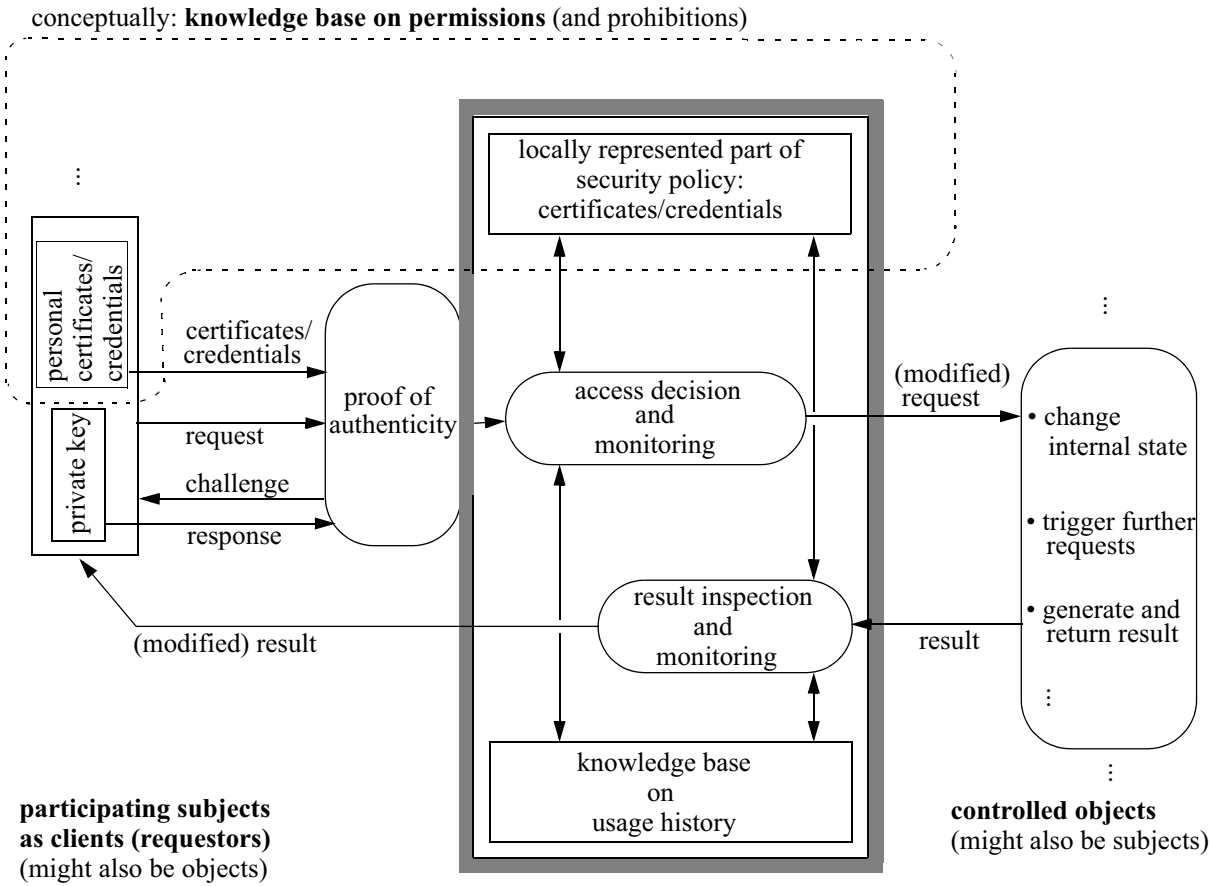


participating subjects

- generate, store and employ secrets
- exploit physical isolation
  (indicated by the gray areas)

controlled objects

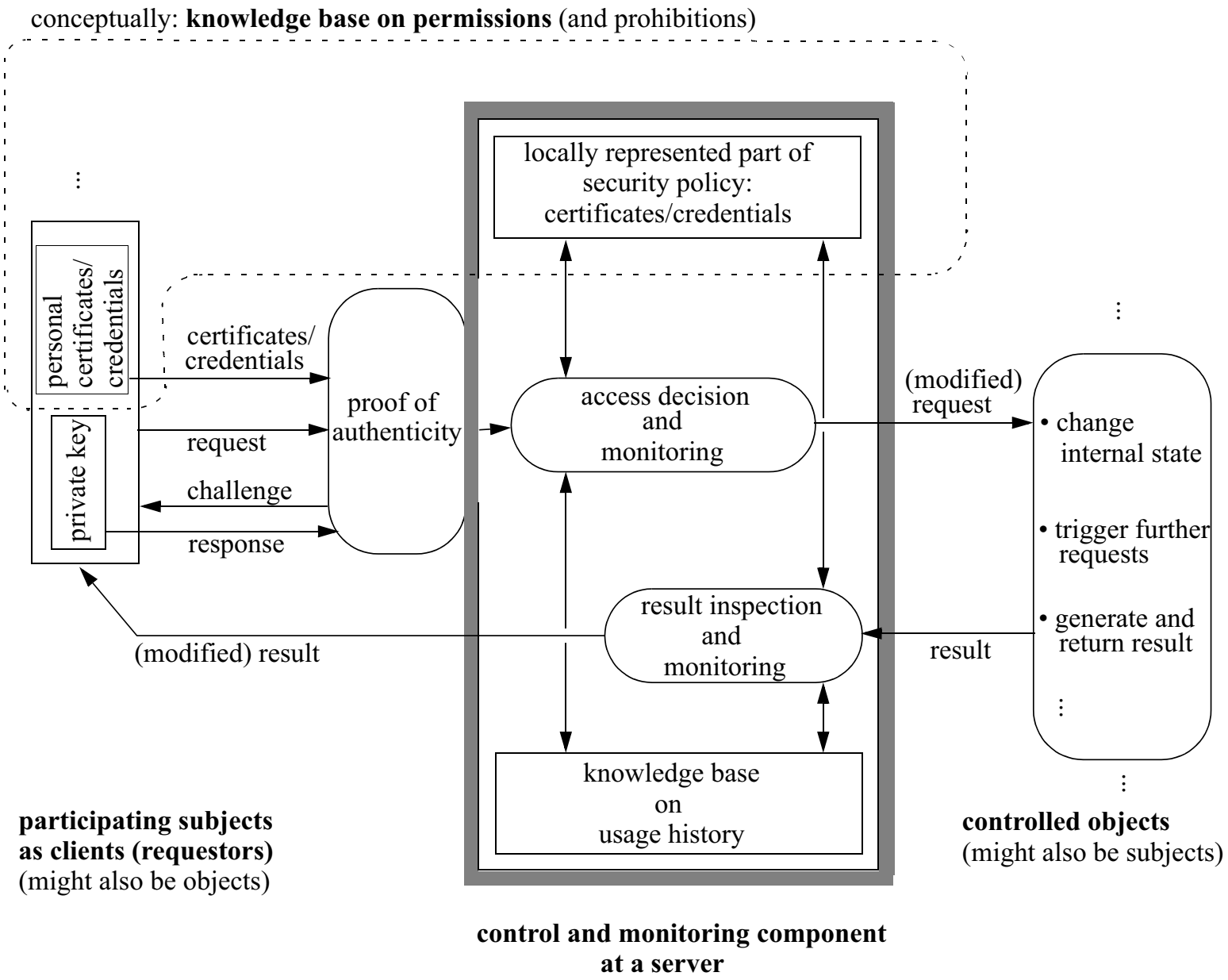
Certificates and credentials



control and monitoring component at a server

- cannot be bypassed
- (virtually) isolates participating subjects from controlled objects
- is based on physical isolation (indicated by the gray frame)
- decides on requests and results and possibly modifies them

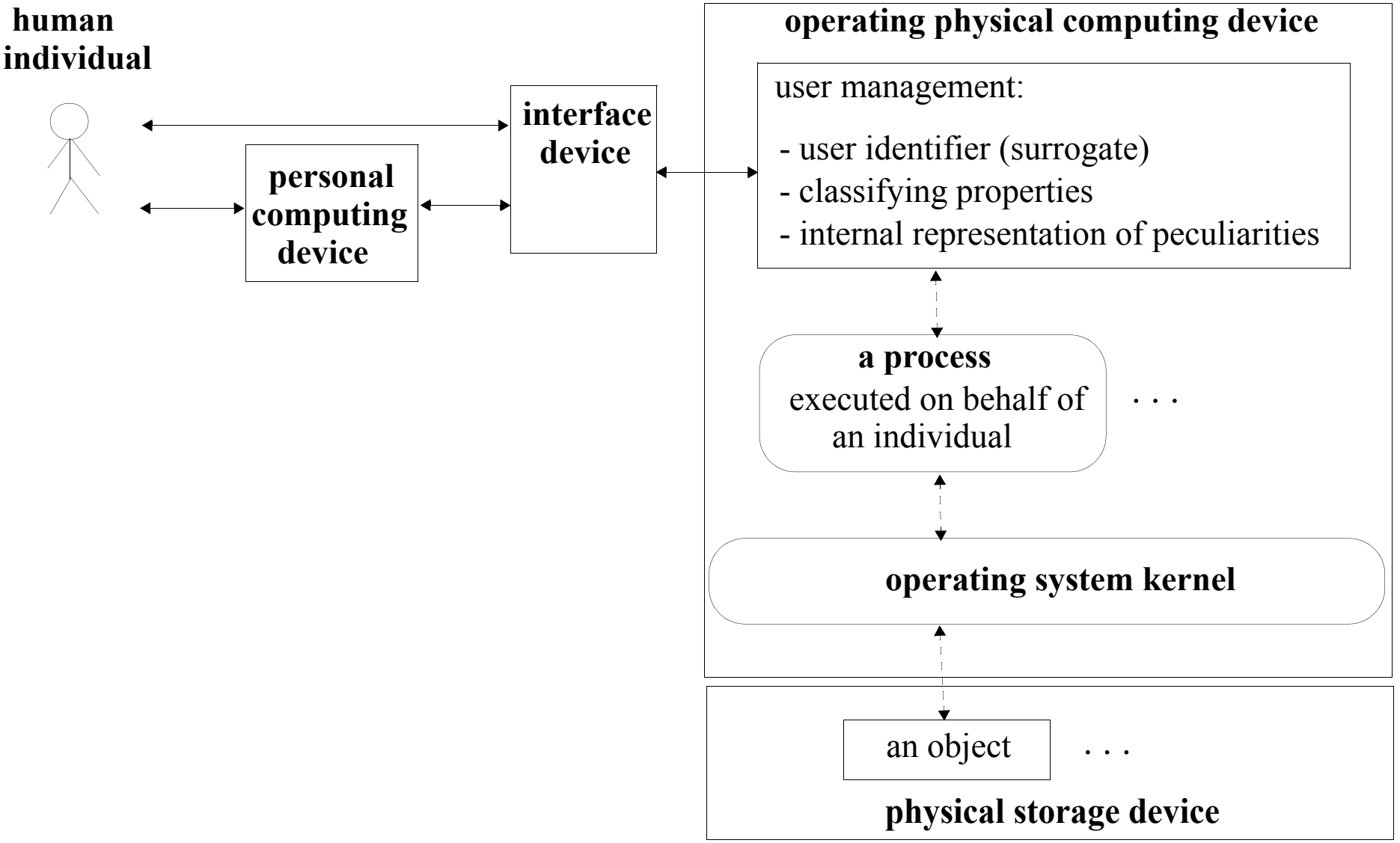
Certificates and credentials



Participants and objects involved

- a *human individual*
- a (physical) *personal computing device*
- a (physical) *interface device*
- a *physical computing device*
  (with a *processor* as its main component,
  and running an *operating system* and other *system software*)
- a *process*
- an *operating system kernel*
- a (physical) s*torage device*
- a (virtual application) *object*

Local identifiers: participants and their local connections



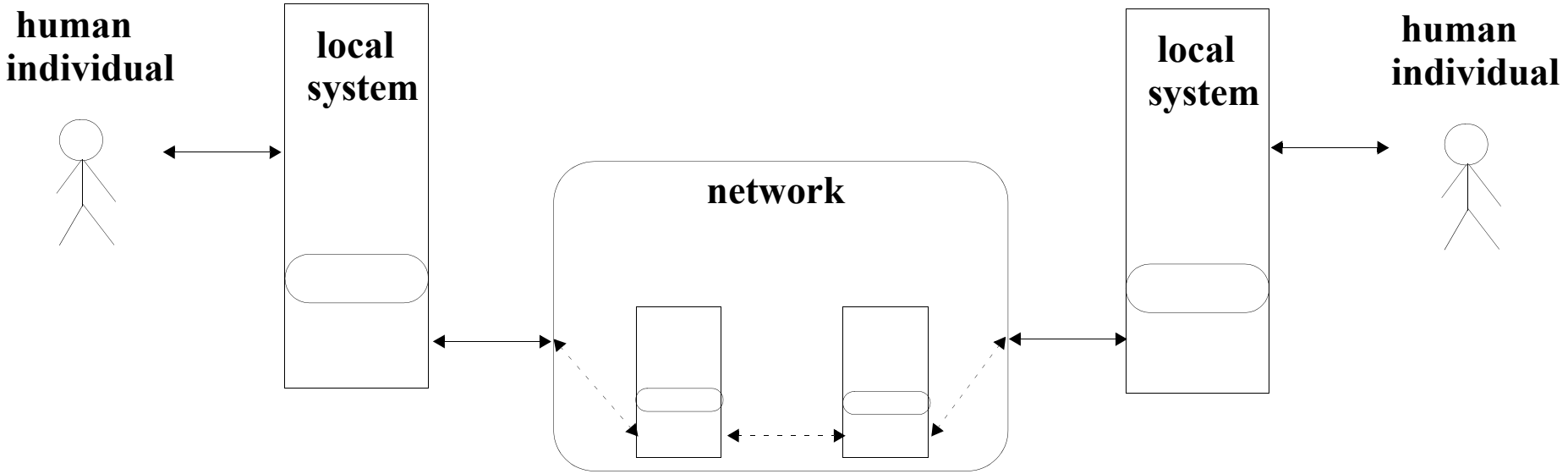
The fiction of an overall “connection”

conceptual perception:
an *individual* is permitted (or prohibited) to perform an *action* on an *object*

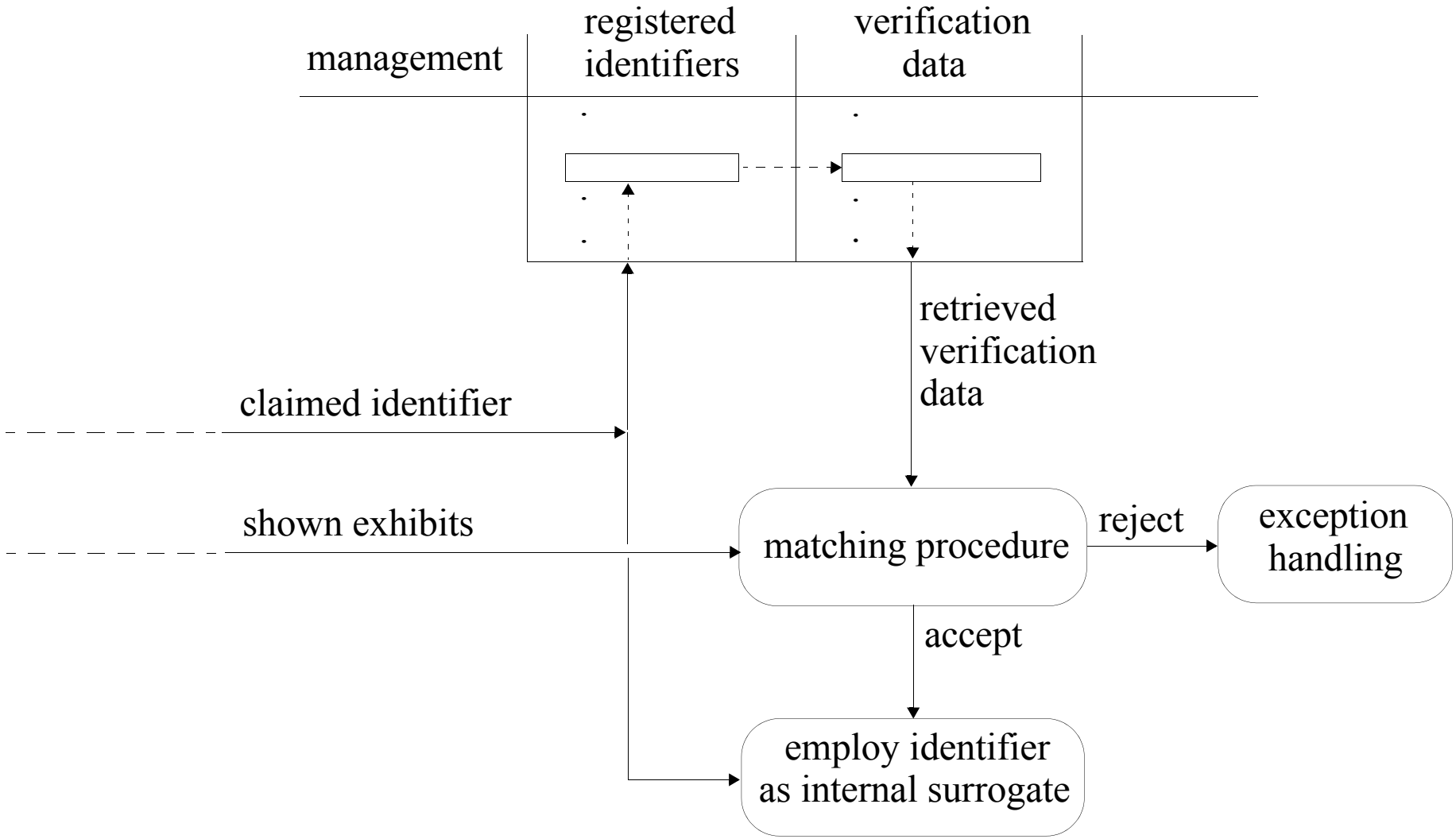
actual requirement:
the “natural identity” of a human individual
must be appropriately reflected along the chain of local connections,
ensuring that the *messages* involved are directed as expected, in particular:

- between the human individual and the interface device:
  either directly or with the help of a secure *personal computing device*
- between the interface device and the physical computing device:
  a secure *physical access path*
- between one process and another local process:
  secure *process communication*
- between a process and the local storage:
  a secure *operating system kernel*

Global identifiers: virtual end-to-end connections



Provisions for authentication and proof of authenticity



Peculiarities of human individuals: examples

- **individual knowledge:**
  - password, passphrase
  - PIN (personal identification number)
  - personal data
  - historic data
  - (discretionarily selected) cryptographic key
  - random number (nonce)
- **physical possession:**
  - smartcard
  - personal(ized) computing device
- **biological characteristics (biometrics):**
  - fingerprints
  - eye pattern,
  - genetic code
  - speech sound
- **individual (reproducible) behavior:**
  - pattern of keyboard striking

Peculiarities of physical devices: examples

- tamper-resistant, physically implanted serial number
- tamper-resistant, physically implanted cryptographic key
- discretionarily selected cryptographic key
- random number

Properties of verification data: informal version

- **(strong) correctness:**
  an exhibit presented is accepted
  iff
  it is authentic for the claimed identifier

- **(extended) unforgeability:**
  knowing the verification data alone
  should *not* enable one to produce any matching exhibits

Some contributions of cryptography

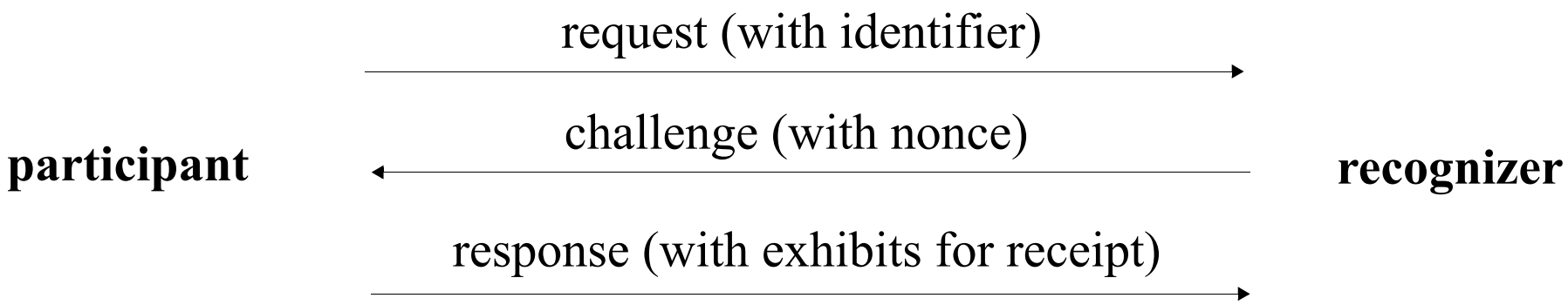
- by applying ***encryption***,
  any verification data can be persistently stored in encrypted form,
  such that only the recognizing system can exploit the verification data

- by applying asymmetric ***cryptographic authentication***,
  a participant's given peculiarity can be made
  to consist of a private (secret) *authentication* or s*ignature key*,
  and the corresponding public *test key* serves as the verification data

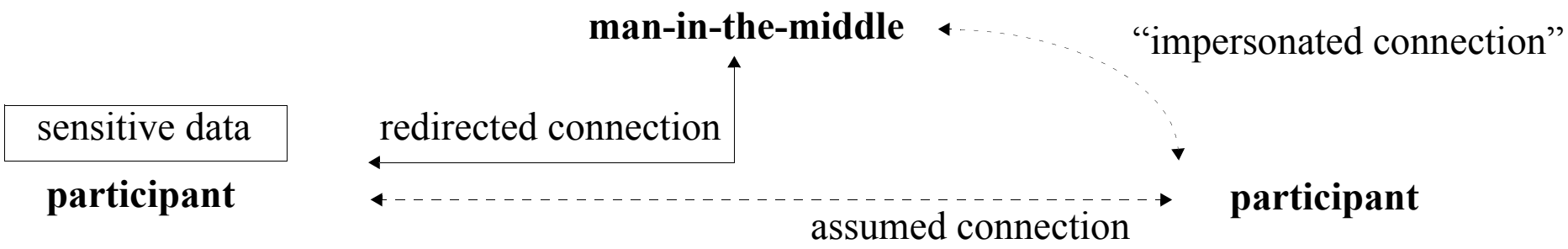
- by applying a collision-resistant ***one-way hash function***,
  a (digital encoding of any) peculiarity is mapped to a *hash value*
  serving as stored verification data;
  later on, the peculiarity can be shown as an exhibit,
  whose hash value is recomputed and compared with the stored value

Issue of authentic verification data: trusted authorities

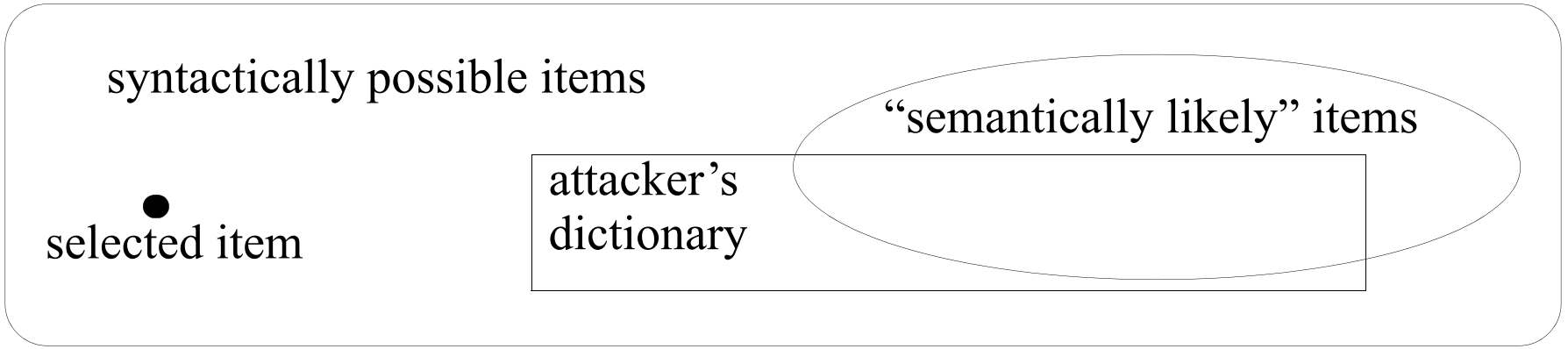
Issue of freshness: challenge–response procedures



Issue of malicious redirection by man-in-the-middle



Issue of malicious guessing or probing: carefully chosen exhibits



Permissions and prohibitions: the need for a layered approach

- participants by themselves, or
  some distinguished participants acting on behalf of the others,
  specify and *declare* the wanted permissions and prohibitions

- declarations are then (hopefully) appropriately *represented*
  by the means of the computing system and inside it

- representations are (hopefully) efficiently *managed* there,
  both for *decisions* on actual requests for an operational option and for *updates*

- decisions are effectively *enforced*, i.e.,
  (hopefully) exactly those requests are successfully executed
  that have been declared permitted, and,
  accordingly, none of those that have been declared prohibited

Specification of permissions and prohibitions: some guidelines

- alignment with the environment
- least privileges according to need-to-know or need-to-act
- separation of roles
- purpose binding
- separation of privileges

Requirements and mechanisms reconsidered

security interests:

- *availability*: requested data/action returned/executed in a timely manner
- *integrity*: an item's state unmodified, or its modification detectable
- *authenticity*: claimed origin of data or action recognized as correct
- *non-repudiation*: correct origin of data or action provable to third parties
- *confidentiality*: information kept secret from unauthorized participants
- *non-observability and anonymity*: activities kept secret
- *accountability*: activities traceable to correct origin

key ideas for security mechanisms:

- *redundancy*: adding additional data or resources
  to enable needed inferences, detect failures and attacks,
  or recover from them
- *isolation*: separating items
  to disable information flows and interferences
- *indistinguishability*: hiding data or activities by letting them appear to be
  random samples of a large collection or uniformly expected

Combined techniques reconsidered

- **local control and monitoring:**
  - *identity*-based
  - identification and proof of authenticity
  - permissions as access rights
  - control of intercepted requests and results
  - monitoring of overall behavior
- **cryptography:**
  - *secret*-based
  - encryption, (cryptographic) authentication including digital signatures, anonymization, randomness, one-way hash functions, timestamps
  - more advanced protocols built from these blocks
- **certificates and credentials:**
  - *property*-based
  - features of local control and monitoring applied to requests that are accompanied by digitally signed assignments of security-relevant properties to public keys

Interests and enforcing mechanisms: summary (part 1)

Interest	Redundancy	Isolation	Indistinguishability	Control and monitoring	Cryptography	Certificates and credentials
Availability	provisionally multiplying (sub)objects or generating auxiliary objects to reconstruct lost or corrupted objects	attributing distinguishing identifiers or characterizing properties confining threatening operations in the context of integrity		granting access rights for enabling permitted operations (and confining them as far as they are threatening) detecting and reconstructing losses and corruptions while intercepting requests and results	generating and distributing secrets (keys) for enabling permitted operations	issuing documents about properties for enabling permitted operations (and confining them as far as they are threatening)
Integrity	provisionally generating auxiliary objects to detect modifications	confining operations on objects to dedicated purposes generating distinguishing secrets	making exhibits appear randomly selected for preventing forgeries	specifying prohibitions for rejecting or confining threatening operations	detecting unwanted modifications of objects	specifying prohibitions for rejecting or confining threatening operations
Authenticity	adding exhibits derived from a distinguishing secret	attributing distinguishing identifiers generating distinguishing secrets	making exhibits appear randomly selected for preventing forgeries	recognizing a requestor by identification and proof of authenticity	recognizing a requestor or actor by verifying cryptographic exhibits	challenging a requestor and verifying cryptographic exhibits in responses

Interests and enforcing mechanisms: summary (part 2)

Interest	Redundancy	Isolation	Indistinguishability	Control and monitoring	Cryptography	Certificates and credentials
Non-repudiation	adding cryptographic exhibits in the form of digital signatures derived from a distinguishing secret	generating distinguishing secrets	making exhibits appear randomly selected for preventing forgeries		proving an actor responsible by verifying cryptographic exhibits in the form of digital signatures	assigning provable responsibility to issuers of documents by verifying cryptographic exhibits in the form of digital signatures
Confidentiality		confining operations on objects to dedicated purposes	making data appear randomly selected from a large collection of possibilities	specifying prohibitions for rejecting or confining threatening operations	prohibiting gain of information by encrypting data	specifying prohibitions for rejecting or confining threatening operations
Non-observability/ anonymity			hiding activities in a large collection of possibilities	untraceably mediating requests and results	superimposing randomness	issuing documents about properties referring to public keys (rather than identities)
Accountability	adding cryptographic exhibits in the form of digital signatures or similar means derived from a distinguishing secret	attributing distinguishing identities generating distinguishing secrets		logging and analyzing intercepted requests and results	proving an actor responsible by verifying cryptographic exhibits in the form of digital signatures or similar means	logging and analyzing intercepted requests and results

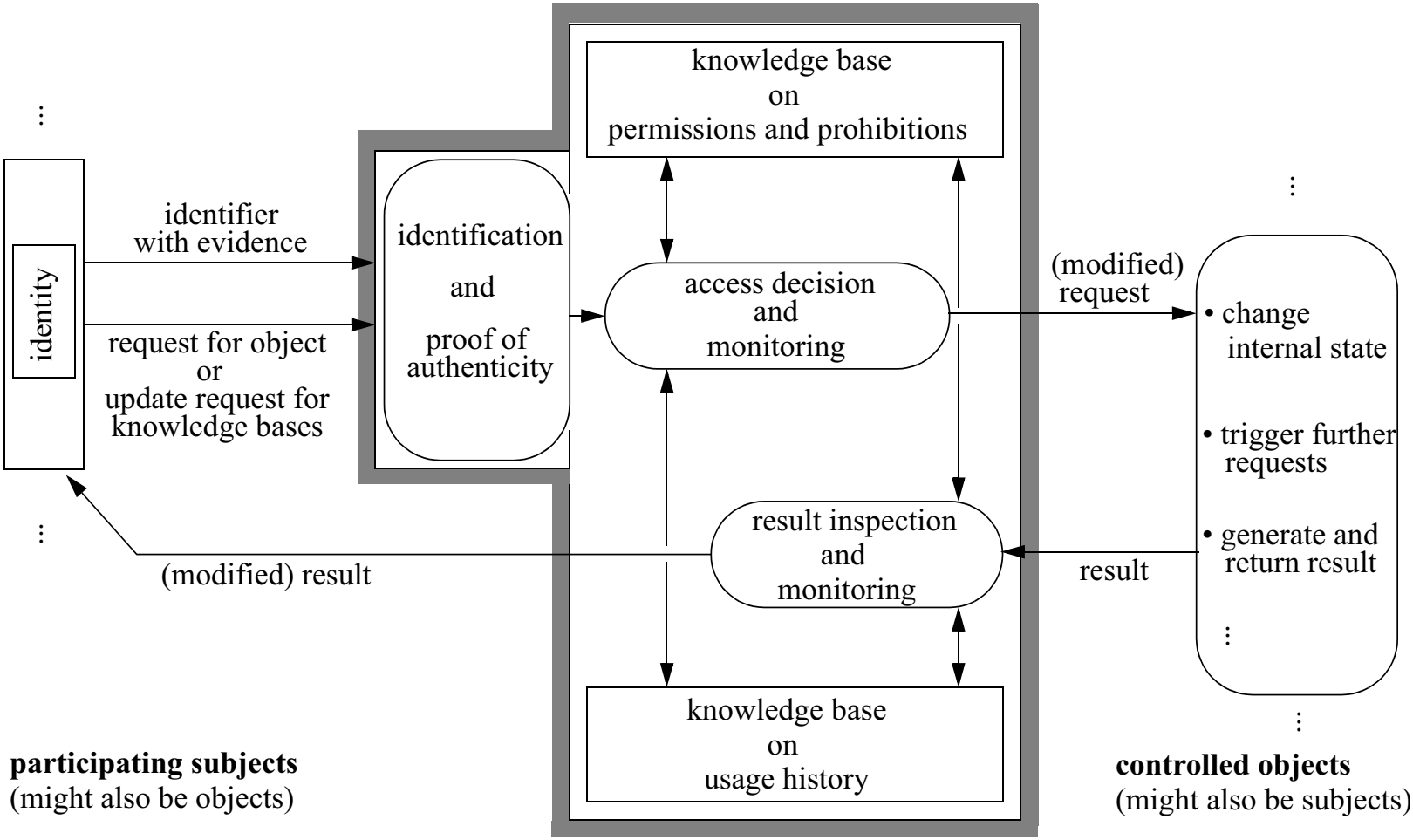
Part II

Control and Monitoring

3 Fundamentals of Control and Monitoring

Control and monitoring

- identifiable agents can have *access rights* granted and revoked
- access requests of authenticated agents are intercepted by *control components*
- control components decide on *allowing* or *denying* an actual access



Essential parts

- declaration of permissions and prohibitions
- control operations
- isolation, interception and mediation of messages
- proof of authenticity
- access decisions
- monitoring

Declarations: subjects, objects and kinds of access

- conceptualize and denote the ***subjects***: carriers of permissions and prohibitions
- where appropriate, treat *collectives* of subjects in a uniform way

- conceptualize and denote the ***objects***: targets of permissions and prohibitions
- where appropriate, collect objects into
  classes, *domains* or related *aggregates* for uniform treatment

- conceptualize and denote the ***kinds of access*** offered:
  from generic *reading* and *writing*
  to application-specific *methods*
- where appropriate, abstract from concrete accesses and
  instead refer to their (*operational*) *modes*

Declarations: expressive means

- a permission or a prohibition can be *directly* expressed
  by *explicitly* naming the respective subject, object and operational mode
- preferably, the needed items are expressed in a more *indirect* way,
  employing a wide range of techniques of computer science
  (programming languages, knowledge engineering, ...):
- in particular, *syntactic* means for
  – collectives of subjects (e.g., *hierarchies*),
  – aggregates of objects (e.g., *complex compositions*)
  – modes of access (e.g., further *method invocations*)
  must be suitably handled at the *semantic* level
- in general, techniques for deriving
  implicit properties of the items considered from explicit properties
  might be exploited
  (e.g., *inheritance* rules, *first-order logic reasoning*, ...)

Declarations: positive, negative and mixed approach

- positive approach:
  only **explicit permissions** expressible,
  and, by default, prohibitions defined as the absence of a permission

- negative approach:
  only **explicit prohibitions** expressible,
  and, by default, permissions defined as the absence of a prohibition

- mixed approach:
  both **explicit permissions and explicit prohibitions** expressible,
  with a need for the *resolution of conflicts* and for *completions*

Required completeness property for declarations

for any request

of a subject *s*

to access an object *o*

in an operational mode *m*,

the declared permissions and prohibitions entail

a unique and definite access decision

Control operations

- first level:
  permissions and prohibitions for the *functionality* of a system, i.e., for *functional operations*

- second level:
  permissions and prohibitions for the ***control operations***
  that manipulate the first-level functional permissions and prohibitions,
  including ***granting*** and ***revoking*** of *functional* permissions and prohibitions;

  more advanced control operations deal with, e.g.,
  transferring or ***delegating*** permissions and prohibitions
  to declare functional permissions and prohibitions

- further levels:
  possible, but rarely employed

Grantors and owners

- need to define which subjects may grant permissions and prohibitions
  - initially
  - by means of some special qualifications

- example:
  a (nearly) omnipotent *administrator*,
  known as *root* or *superuser*, is permitted
  - to manage *any kind* of permissions and prohibitions
  - to assign each subject that generates a new object
    the *ownership* of the creation, coupled with the permission
    to manage the permissions and prohibitions *for that creation*

Control states

- the granting of permissions should be done with great care
- an administrator or owner planning some control operations has to *analyze* the potential consequences regarding which subjects can *eventually* acquire which permissions
- more generally, for any *control state* resulting from control operations, such an analysis should be performed (unfortunately, in general computationally infeasible or even impossible)

Required analysis property for control operations

for any control state resulting from control operations,

the analysis problem regarding

which subjects can eventually acquire which permissions

should be computationally feasible

or at least admit a computational approximation

Isolation, interception and mediation of messages

- effective enforcement of declared permissions and prohibitions
  relies on an appropriate system *architecture*:
  - it strictly *isolates* subjects from objects
  - it considers that some entity might act *both* as a subject *and* as an object
- a subject should *not* be able to *directly* access any object
- a subject can send a message containing an *access request*
  that will be *intercepted* by a separating *control and monitoring component*
- the control and monitoring component mediates the request,
  basically in three steps:
  - *identification* and *proof of authenticity*
  - *access decision* and forwarding
  - further *monitoring*

Required complete mediation property

each request of a subject

to access an object

is intercepted and mediated

by a control and monitoring component

Proof of authenticity

- declared permissions and prohibitions refer to well-conceptualized *subjects*
- the control and monitoring component must relate
  the sender of any request message, an actual *requestor*, to a pertinent subject
- given a request message,
  the control and monitoring component must *recognize* the requestor
  as one of the conceptualized subjects,
  being aware of the possibility of a maliciously cheating agent
- the requesting agent must provide some further *evidence* regarding itself;
  the control and monitoring component can then base a *proof of authenticity* on
  - the *freshly* communicated *evidence*
  - suitably maintained permanent *verification data*

Required authenticity property

any mediation of an access request

is based on a proof of authenticity

of the requestor and,

as far as needed,

of the target object as well

Access decisions

- once a requestor has been recognized as a conceptualized subject,
  the control and monitoring component takes an *access decision*
  by evaluating the request with respect to
  the previously declared permissions and prohibitions
- the declarations constitute a *knowledge base on permissions and prohibitions*,
  from which the access decision is derived as a logical consequence
- such *derivations* might vary
  from simple lookup procedures to highly sophisticated reasoning
- such *reasoning* might additionally consider
  the *dynamic evolution* of the controlled system,
  as conceptually represented by a *knowledge base on the usage history*
- such a knowledge base must be appropriately maintained
  by *logging* all relevant events

Requirement for architecture of control

the control and monitoring component

maintains suitably isolated knowledge bases

on permissions and prohibitions and

on the usage history

Monitoring: inspecting results

- an accepted and forwarded request might produce some *results* that should be *inspected* afterwards
- if the *results* are to be *returned* to the original requestor: the inspection might retain all or some parts of them:
  - totally *block* the forwarding to the requestor, or
  - suitably *modify* the results before forwarding
- if an *internal state* of an accessed object might have been changed or *further requests* to other objects might have been triggered:
  the options for undoing such effects depend strongly on additional mechanisms such as *transactions*, seen as atomic actions that can be finally either completely *committed* or *aborted*
- in case of an *abort*, the effect should be (largely) *indistinguishable* from the situation where the access has not occurred at all

Monitoring: auditing and intrusion detection

- complementary to access decisions and result inspection,
  the control and monitoring component can analyze
  all messages and possibly further audit data
  regarding an *intrusion defense policy*

- such a policy assists in classifying the activities actually occurring as
  either *semantically acceptable* or *violating*

- the notions of *permissions* and *prohibitions* should be semantically related
  to the notions of *acceptable behavior* and *violating behavior*, respectively

- in general, however, these notions will not fully coincide because of
  - inevitable shortcomings of the preventive access control mechanisms
  - efficiency considerations (leading to an *optimistic* approach)

Requirement for architecture of monitoring

complementarily to access decision and result inspection,

the control and monitoring component

audits and analyzes all activities regarding potential violations

defined by an intrusion defence policy

Imagined ideal and real world

- **ideal world:**
  - all subjects behave as expected
  - all informational devices actually operate as completely specified
  - correct and complete knowledge is available whenever needed

- **real world:**
  - such an imaginary scenario is not met with at all
  - security aims at managing the imperfections, including:
    - potentially maliciously behaving subjects,
    - failing implementations of inadequate designs,
    - decision making regarding remote subjects

Root of trust

- there always remains the need
  to base at least small parts of an overall computing system on *trust*

- trust in a *technical part* usually means,
  or at least includes the requirement,
  that the *participant* controlling that part is trusted

- as security is a *multilateral* property
  that respects potentially conflicting *interests*,
  trust is essentially *context-dependent*, i.e.,
  subjectively assigned by one participant
  but refused by another one

Issues of trust raised when the following problems are investigated

- does the *control and monitoring component* actually work as expected, intercepting and suitably mediating each access request?

- does it support *availability* by accepting permitted requests, and does it preserve *integrity* and *confidentiality* by denying prohibited accesses?

- do participants permitted to execute *control operations* behave appropriately and honestly when granting, revoking, transferring or delegating permissions?

- do shown *evidence* and maintained *verification data* reflect the actual *peculiarities* of remote communication partners?

4 Case Study: UNIX

Some basic features of UNIX

- UNIX supports participants in
  – using their own workstation for their specific application tasks
  – cooperating with colleagues in server-based local networks for joint projects
- a participant can manage his own computing resources at his discretion,
  – either keeping them private
  – or making them available to other particular participants or to everybody
- security mechanisms
  – enforce the virtual isolation of identified, previously registered users
  – enable the deliberate sharing of resources
- the mechanisms are closely intertwined with the basic functional concepts of files and processes, which are managed by the UNIX kernel
- the kernel acts as controller and monitor of all security-relevant accesses

Basic blocks of control and monitoring (and cryptography)

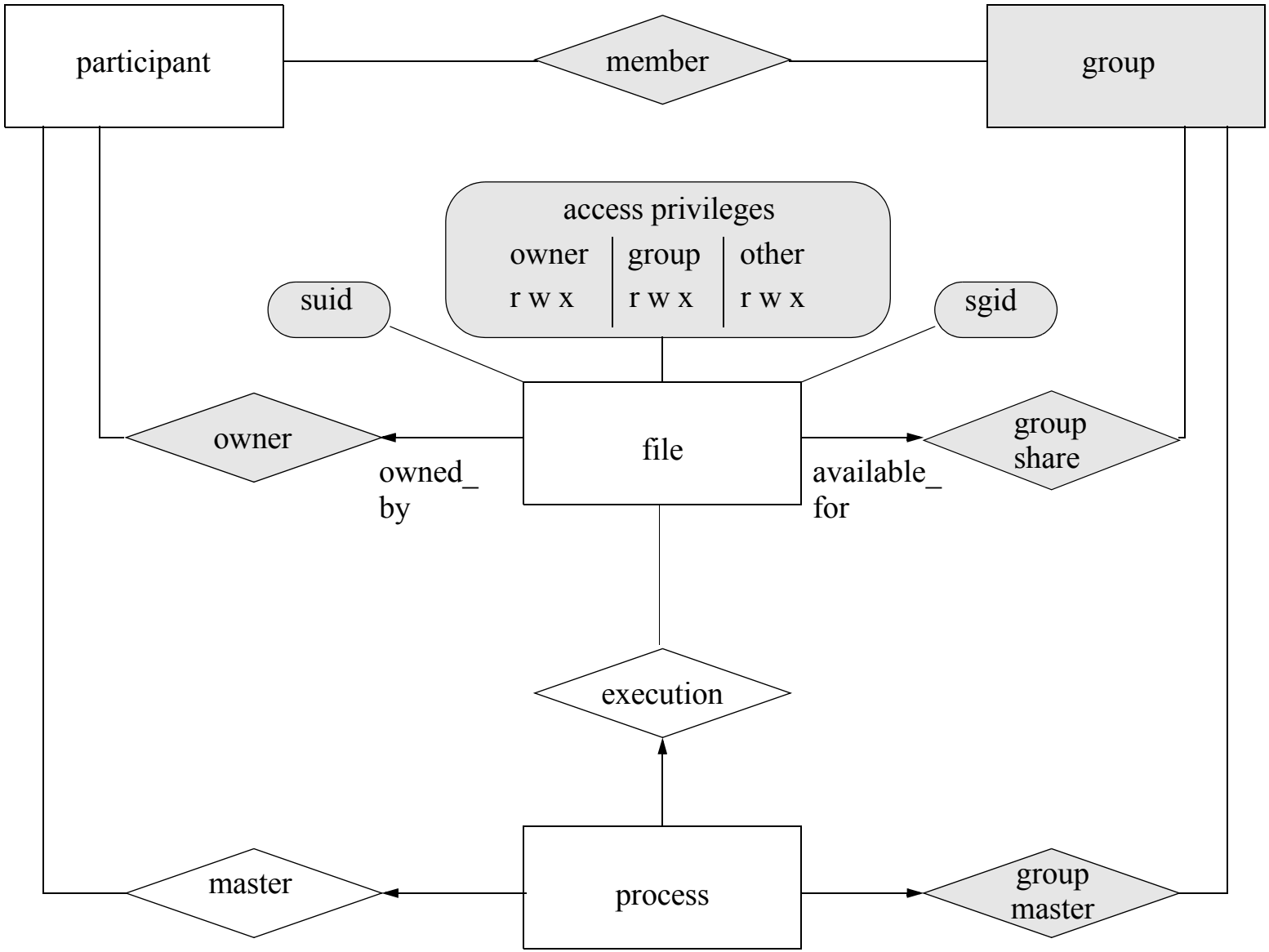
- *identification* of registered users as participants
- *passwords* for user *authentication* at login time
- *a one-way hash function* for storing password data
- *discretionary access rights* concerning *files* as basic objects and three fundamental *operational modes*, read, write and execute
- *owners*, as autonomous grantors of access rights
- owners, *groups* and the full community of all users, as kinds of grantees
- *right amplification* for temporarily increasing the operational options of a user
- a *superuser*, capable of overriding the specifications of owners
- *access control* concerning the commands and the corresponding system calls
- *monitoring* of the functionality
- *kernel*-based implementation of control and monitoring

Conceptual design of the operating system functionality

- UNIX provides a *virtual machine*
  that offers an external *command* interface
  with the following fundamental features:
  - identified *participants* can
  - *master processes* that
  - *execute* programs
  - stored in *files*

- the processes, in turn, can operate on files,
  in particular for *reading* and *writing*

ER model of fundamental functional features and security concepts



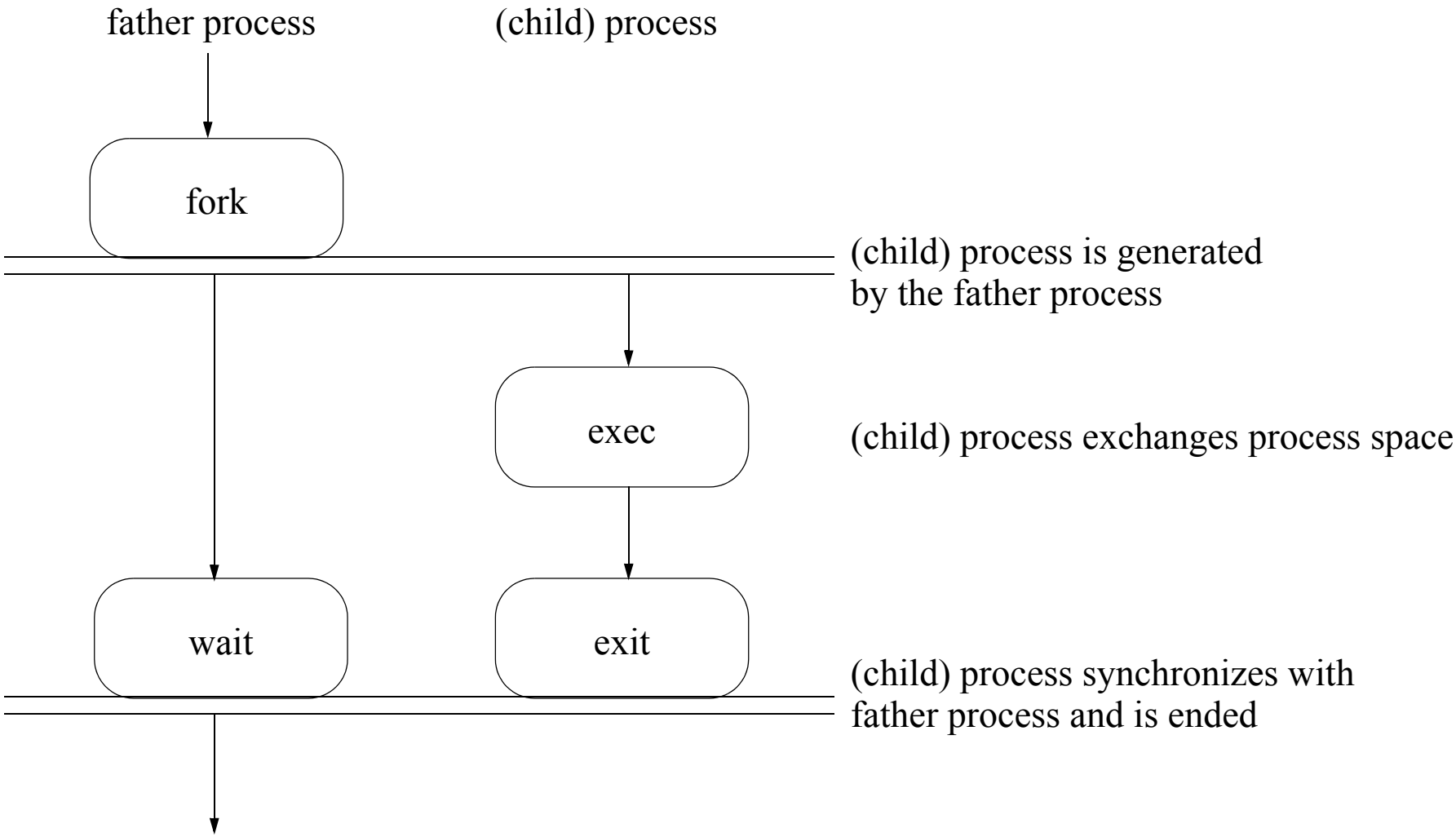
Participants, sessions and system calls

- a previously *registered participant*
  can start a *session* by means of the `login` command
- thereby the *system*
  - assigns a *physical device* for input and output data to him
  - starts a *command interpreter*
    as the first process mastered by that participant
- afterwards, the participant can issue *commands*,
  which may possibly generate additional processes
  that are also mastered by him
- the commands invoke *system calls* that serve for
  - process management
  - signaling
  - file management
  - directory and file system management
  - protection
  - time management

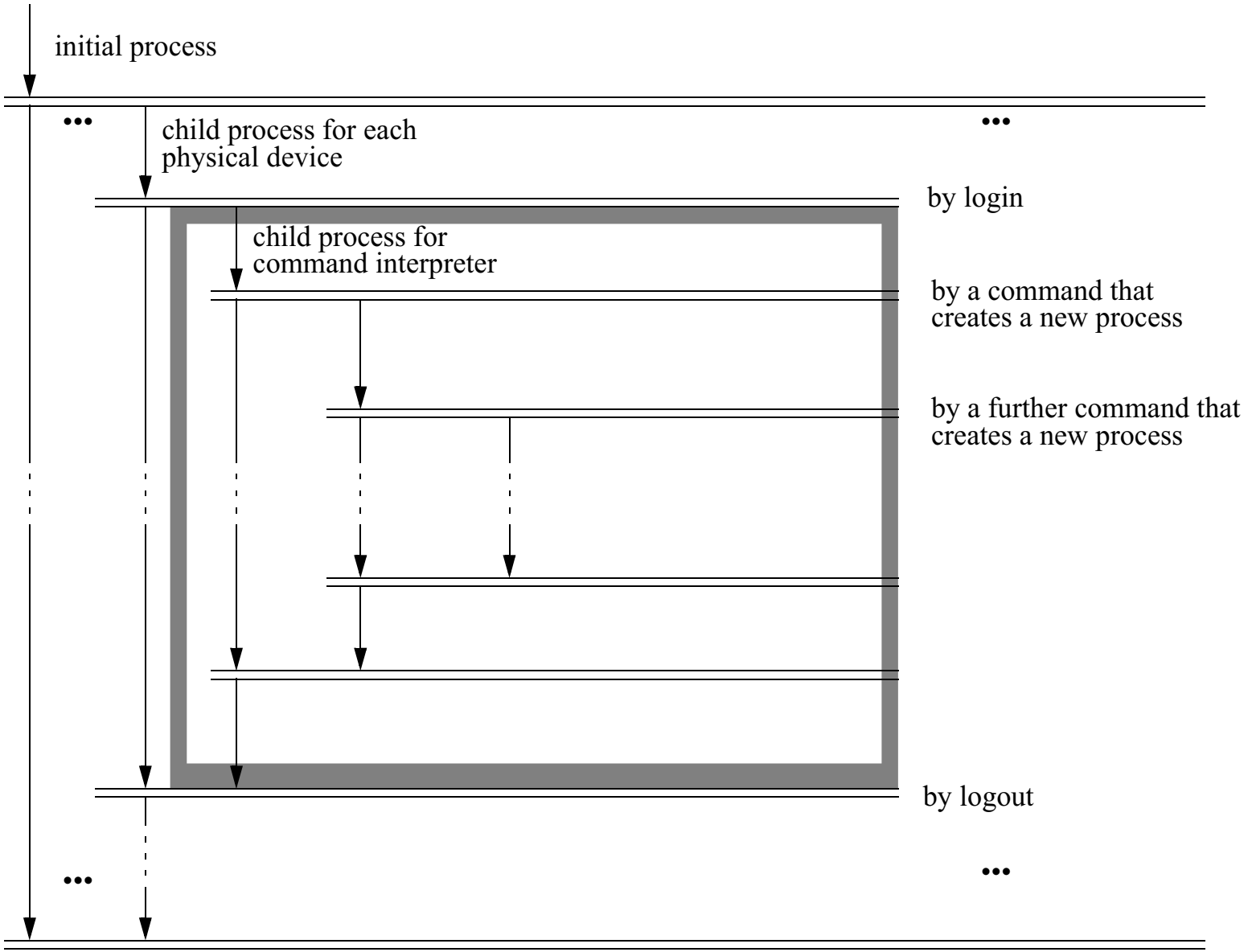
Processes as active subjects

- *execute* (the program contained in) a file, and in doing so
- *read* or *write* in (usually other) files
- *create* new files and *remove* existing ones
- *generate* new (child) processes
- have a *lifespan,*
  starting with the generation by a father process and
  ending with a synchronization with the pertinent father process
- constitute a *process tree*:
  - when the UNIX system is started, an initial process *init* is generated
  - an already running (*father*) *process* can generate new (*child*) *processes*

Lifespan of a process



Growing and shrinking of a process tree



Files as passive objects

- files are uniformly managed by the system using a file tree
- a file is identified by its *path name* within the file tree
- a file that constitutes a branching node in the file tree
  is a *directory* listing other files
- a file that constitutes a leaf in the file tree
  is a *plain file* containing data,
  which might be considered as an executable program

Conceptual design of the security concepts

- a participant acts as the *owner* of the files created by him

- the system administrator assigns participants as *members* of a *group*:
  - a group comprises those participants that are entitled to share files
  - an owner can make a file *available* for a group to *share* it

- for each file, the owner implicitly specifies three *disjoint* participant classes:
  - himself as *owner*
  - the members of the pertinent *group*, except the owner if applicable
  - all *other* participants

- the owner of a file *discretionarily* declares *access privileges*
  for each of these classes – for the processes mastered –
  by permitting or prohibiting the operations
  belonging to an *operational mode*:
  - **r***ead*
  - **w***rite*
  - *e***x***ecute*

Some operations with commands and their operational mode

Operation with command on plain file	**Operation with command on directory**	**Operational mode**

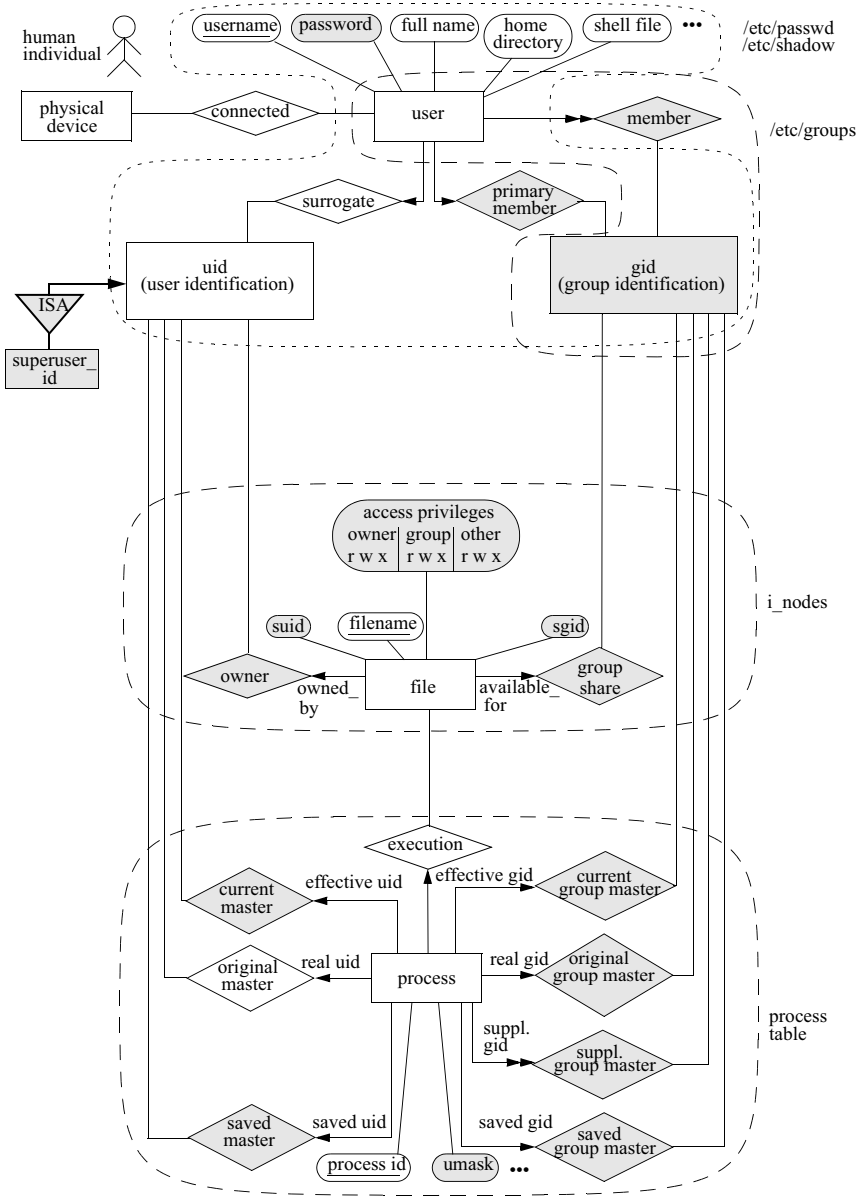
open file for reading:
`open( ,o_rdonly)`
 	
read content:
`read`
	open directory for scanning:
	`opendir`
	
	read next entry:
	`readdir`
		***r**ead*

open file for writing: `open( ,o_wronly)` modify content: `write` delete content: `truncate`	 insert entry: `add` delete entry: `remove` rename entry: `rename`	***w**rite*
execute content as program: `execute`	select as current directory: `cd`	***e**x**e**cute*

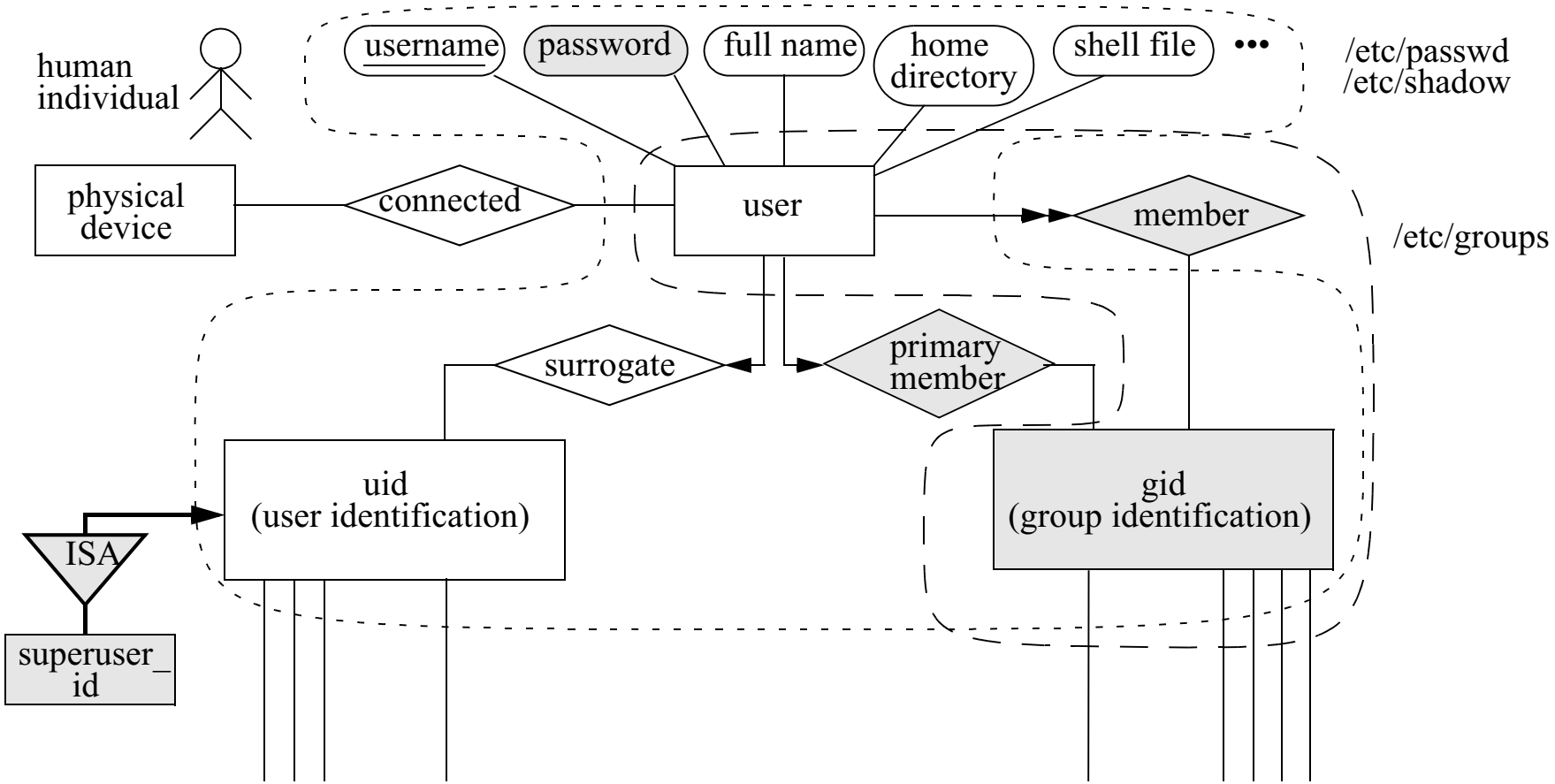
Mastership and group mastership

- normally,
  a user is the *master* of the command interpreter process that he has started,
  and of all its descendants
- additionally, the (primary) group of that user is said to be the
  group master of all those processes
- if a process requests an operation `op` on a file `file`,
  then the access privileges `file.access_privileges`
  are inspected according to the masterships of the process
  in order to take an access decision
- for each file, the owner can additionally set two *execution flags*,
  `suid` and `sgid`,
  that direct its usage as a program, or as a directory, respectively:
  - for a plain file containing an executable program,
    the flag impacts on the *mastership* of an executing process
  - for a directory,
    the flag impacts on the *ownership* of inserted files

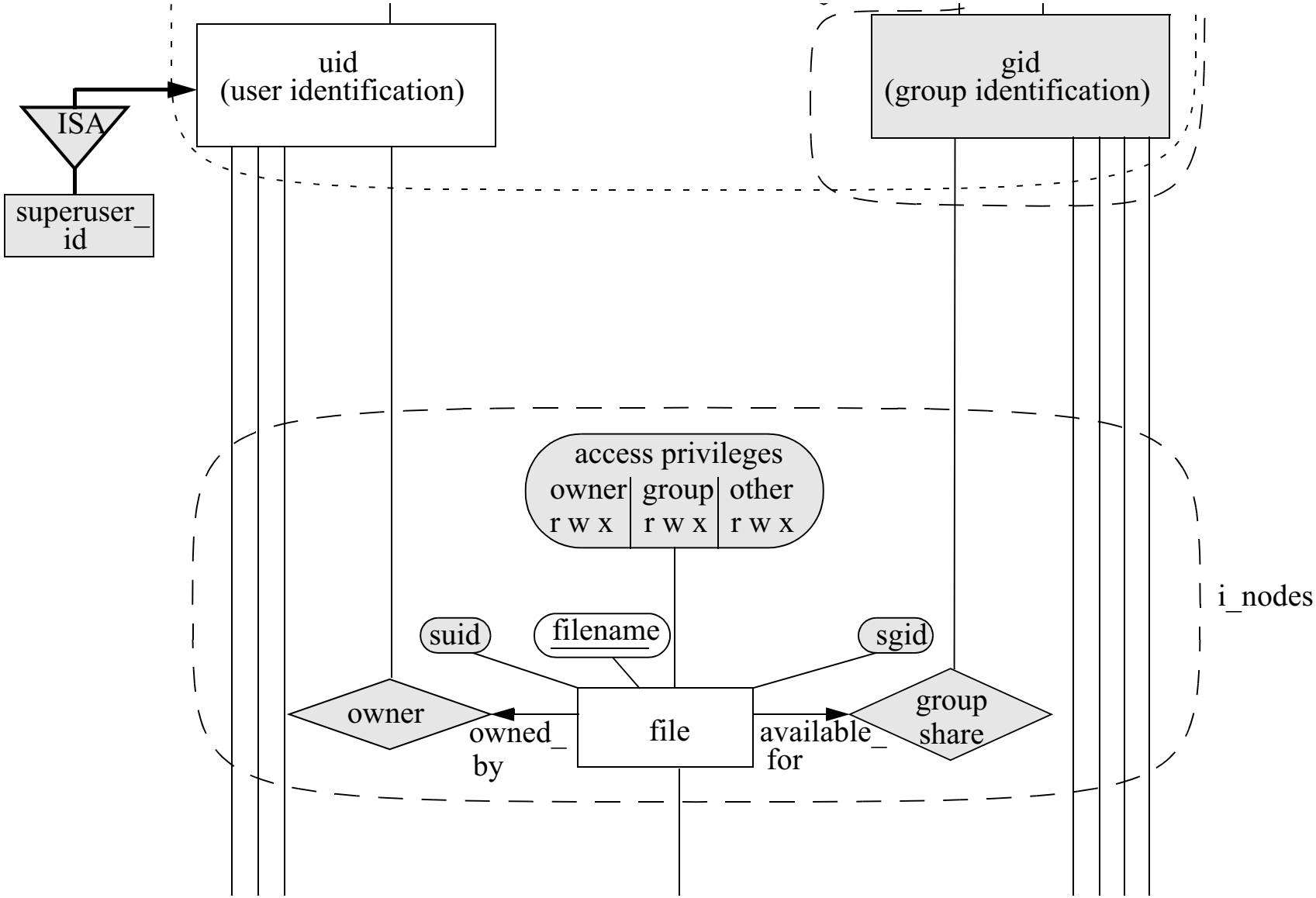
Refined ER model of the functional features and security concepts



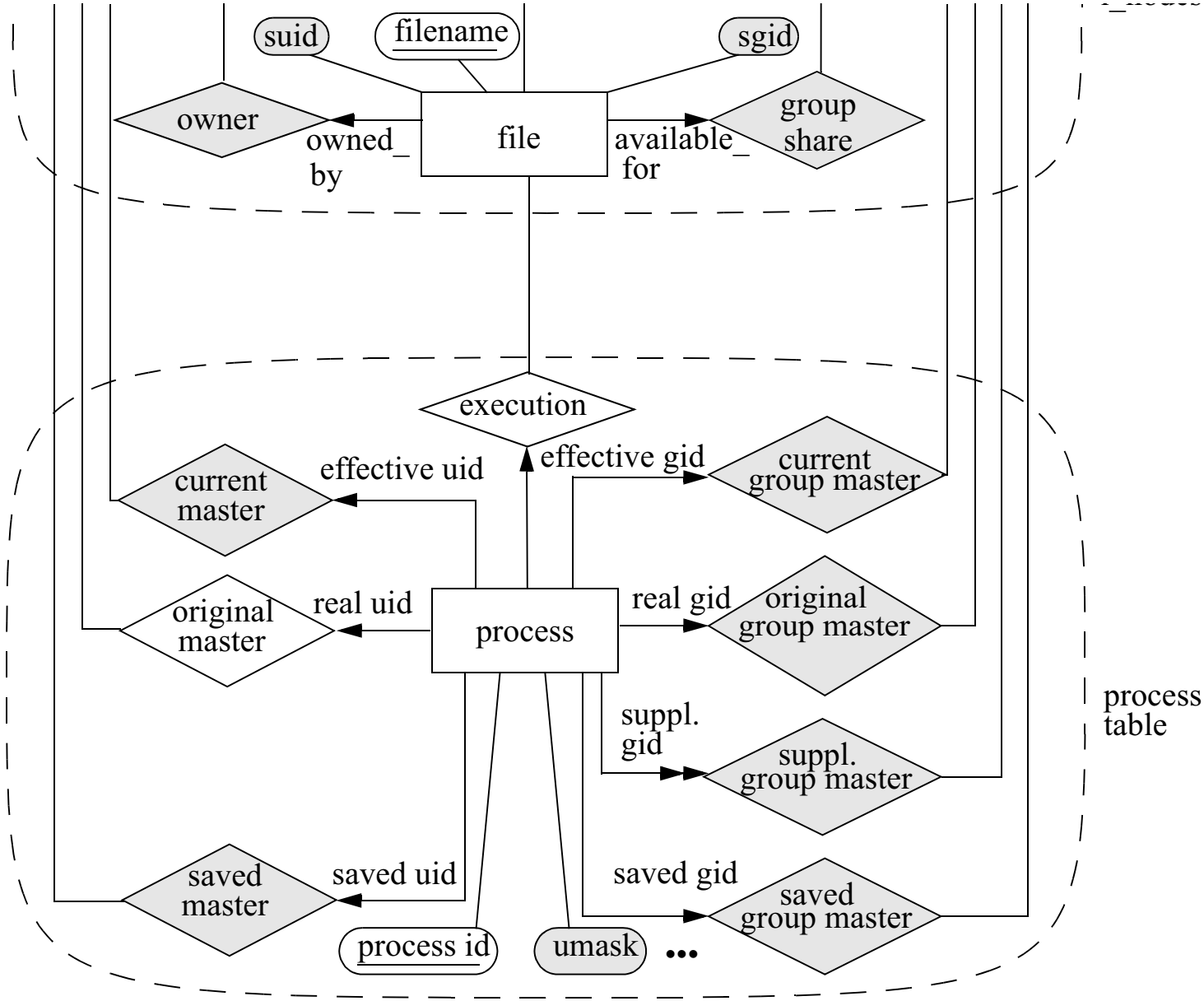
Refined ER model: users



Refined ER model: files



Refined ER model: processes



Different notions of a participant

- a *human individual*
- the *physical device*
  from which the individual issued his last `login` command
- an abstract *user*:
  - representing the previously registered human individual within the system:
    as a result of a successful `login` command,
    the abstract user is *connected* to the
    physical device from which the command was received
  - uniquely identified by a *username*
  - associated with further administrative data, e.g.:
    - *password* data
    - *full name*,
    - (the path name of) *home directory* in the overall file tree
    - (the path name of the file containing) *command interpreter* (*shell file*)
- a *user identification*, i.e., a cardinal number *uid*,
  which serves as a (not necessarily unique) *surrogate* for an abstract user

System administrator

- is a *human individual*,
  typically registered as a distinguished *abstract user*
  whose username is *root* and
  whose surrogate is `superuser_id`
  (in general, represented by `0`)

- enjoys nearly unrestricted operational options

  (consequently, so does any human individual
  who succeeds in being related to *root*)

Groups

- a group is represented by a *group identification*, *gid*
- each abstract user is a *primary member* of one group, and can be a *member* of any further groups

Mastership and group mastership refined

- all relationships of files/processes with participants/groups are interpreted as relationships with *user identification*/*group identifications*
- the *master* and the *group master* relationships are further differentiated in order to enable dynamic modifications
- a user identification *uid*
  (the surrogate of a user connected to a physical device
  from which a human individual has issued a `login` command)
  is seen as the *original master* of the *command interpreter process*
  generated during the login procedure
  and of all its descendants
- these processes are also said to have this *uid* as their *real uid*
- correspondingly,
  a group identification *gid*
  is seen as the *original group master* of these processes,
  which are also said to have this *gid* as their *real gid*

Current masterships

- normally, the *original* masterships are intended to determine the access decision when a process requests an operation on a file
- to distinguish between normal and *exceptional* cases,
  - an additional *current mastership* (an *effective uid*) and
  - an additional *current group mastership* (an *effective gid*)

  are maintained and actually employed for access decisions
- the current mastership and the current group mastership of a process are automatically manipulated according to the execution flags `suid` and `sgid` of the executed file:
  - normally, if the respective flag is *not* set,
    then the *current mastership* is assigned the *original mastership*, and
    the *current group mastership* is assigned the *original group mastership*
  - exceptionally, if the respective flag is set,
    then the *current mastership* is assigned
    the *user identification of the owner of the file to be executed*, and
    the *current group mastership* is assigned
    the *group identification for which that file has been made available*

Right amplification

- the exceptional case is used for *right amplification*,
  to dynamically increase the operational options of a process
  while it is executing a file with a flag set

- the owner of that file allows all "participants"
  that are permitted to execute the file at all
  to act thereby as if they were the owner himself

- if the owner is more powerful than such a participant
  (e.g., if the owner is the nearly omnipotent abstract user *root*),
  then the operational options of the participant are temporarily increased

- the current masterships and current group masterships
  can also be manipulated by special, suitably protected commands

- for this option, the additional
  saved mastership and *saved group mastership*
  are used to restore the original situation

Identification and authentication

- a human individual can act as a participant of a UNIX installation only if the system administrator has *registered* him in advance as *user*, thereby assigning a *username* to him
- this assignment and further user-related data are stored in the files `/etc/passwd` and `/etc/shadow`
- the usernames serve for *identification* and for *accountability* of all actions
- whenever an individual submits a `login` command, the system
  - checks whether the username is *known* from a registration by inspecting the file `/etc/passwd`:

    if the username is found, it is considered as known, otherwise as unknown
  - evaluates whether the actual command is *authentic*, relying on:
    - appropriate registrations
    - the integrity of the underlying files

Proof of authenticity by a password procedure

- if the individual can input the agreed password,
  then the command is seen as authentic
- the system relies on
  - appropriate password agreements
  - the individual's care in keeping his password secret
  - the integrity and confidentiality of the file `/etc/shadow`
- the confidentiality of this file is supported by several mechanisms:
  - passwords are not stored directly,
    but only their images under a *one-way hash function*
  - on any input of the password,
    the system immediately computes its *hash value* and
    compares that hash value with the stored value
- the hash values are stored in a specially protected file `/etc/shadow`:
  - a *write access* to an entry (password modification) is allowed only if
    the request stems from *root* or from the pertinent user
  - a *read access* to an entry is allowed only for authenticity evaluations

Access decisions

- the kernel has to take *access decisions* concerning
  - a *process* as an active subject
  - a *file* as a controlled passive object
  - a requested *operation*

- given a triple `(process, file, operation)`,
  the kernel has to decide whether
  - the process identified by `process` is allowed
  - to actually execute the operation denoted by `operation`
  - on the file named `file`

- two cases according to the *effective user identificatio*n of the process, `process.current_master`:
  - if `process.current_master = superuser_uid`,
    then nearly everything is considered to be allowed
  - otherwise, a decision procedure is called

Access decisions regarding normal users

```
function decide(process, file, operation): Boolean;

if    process.current_master = file.owner
then  return file.access_privileges.owner.mode(operation)


else
  if   process.current_groupmaster = file.group
       OR
       EXISTS process.supplementary_groupmaster:
       process.supplementary_groupmaster = file.group
  then return file.access_privileges.group.mode(operation)


  else return file.access_privileges.other.mode(operation)
```

Knowledge base on permitted operational options

- implemented by means of the fundamental functional features of UNIX
- data about *users* and *groups* is stored in the special files
  - `/etc/passwd`
  - `/etc/shadow`
  - `/etc/group`
- these files are owned by the system administrator (under `superuser_id`)
- the access privileges for these files are given by
  - `r--|r--|r--`
  - `rw-|---|---`
  - `r--|r--|r--`
- additionally, modifications of the files `/etc/passwd` and `/etc/group` are specially restricted to processes with the effective uid `superuser_id`
- security-relevant data about *files* is managed in *i-nodes*
- security-relevant data about *processes* is maintained in the *process table*

Main entries of the administration files for users and groups

`/etc/passwd`	`/etc/shadow`	`/etc/group`
username	username	groupname
reference to `/etc/shadow`	hash value of password	group password
user identification (uid)	date of last modification	group identification (gid)
gid of primary group	maximum lifetime	usernames of members
full name, comment	date of expiration	
path name of home directory		
path name of shell file		

Modifications of the knowledge base: user and group administration

- the commands `useradd`, `usermod` and `userdel`
  manipulate the entries for *users*
  in the files `/etc/passwd`, `/etc/shadow` and `/etc/group`:

  only executed for a process
  whose effective user identification is `superuser_uid`

- the commands `groupadd`, `groupmod` and `groupdel`
  manipulate the entries for *groups*
  in the file `/etc/group`:

  only executed for a process
  whose effective user identification is `superuser_uid`

Modifications of the knowledge base: password management

- the command `passwd`
  modifies an entry of a user in the file `/etc/shadow`:

  only executed for a process
  whose effective user identification is

  – `superuser_uid`

  or

  – equal to the user identification of the user
    whose password is requested to be changed

Modifications of the knowledge base: login procedure

- the command `login` tries to identify and authenticate the issuer
- on success, the issuer is recognized as a known registered `user`
- by a system call `fork`, a new process is generated for that user
- that process, by use of a system call `exec`,
  starts executing the `shell file` of the user as a command interpreter
- the masterships and group masterships are determined as follows:
  - the `real uid`, `effective uid` and `saved uid` are all assigned
    the user identification of the user, i.e., `user.surrogate`
  - the `real gid`, `effective gid` and `saved gid` are all assigned
    the primary group of the user, i.e., `user.primary_member`
  - the `supplementary gid` is assigned
    the set of elements of `user.member`
- subsequently, this process is treated as the original ancestor of all processes that are generated during the session started by the `login` command

Modifications of the knowledge base: mastership assignments

- normally,
  a process inherits its masterships and group masterships
  from its immediate ancestor

- exceptionally,
  masterships and group masterships are determined differently, namely if
  - the file executed has an execution flag `suid` or `sgid` set,

  or
  - some explicit command modifies the implicit assignment

Modifications of the knowledge base: file management

- the system call
  `create(filename, access_privileges, suid, sgid)`
  creates a new file

- the `owner` and the `group share` of the file are assigned
  the `effective uid` and the `effective gid`, respectively,
  of the creating process

- the `access privileges` and
  the execution flags `suid` and `sgid` are assigned
  according to the respective parameters of the call,
  possibly modified according to the mask `umask`

Modifications of the knowledge base: masking access privileges

- the mask `umask` specifies nine truth values,
  one for each value contained in the parameter for the access privileges:
  - each mask value is complemented
  - the conjunction with the corresponding parameter value is taken

- a mask value `true` (or `1`) is complemented into `false` (or `0`) and thus
  always results in the corresponding access privilege being set to `false` (or `0`),
  thereby expressing a *prohibition*

- in general, individuals are strongly recommended
  to prohibit write access to files with an execution flag `suid` or `sgid` set:
  avoids unintended/malicious modification of the program contained,
  resulting in unwanted effects of right amplification

- the system call `umask(new_umask)`
  modifies the current nine truth values of the mask `umask` into
  the values specified by the parameter `new_umask`

Modifications of the knowledge base: process management

- the system call `fork` generates a new process
- a subsequent system call `exec(command_file)`
  exchanges the content of its address space, thereby loading the program that is contained in the file specified as the parameter `command_file`, whose instructions are then executed
- masterships, group masterships and the mask `umask` of that process:
  - if the flags `suid` and `sgid` of the file `command_file` are *not* set, then the new process inherits all masterships and group masterships from its father process
  - if the flag `suid` is set,
    then the `effective uid` and the `saved uid` are assigned to `command_file.owner`
  - if the flag `sgid` is set,
    then the `effective gid` and the `saved gid` are assigned to `command_file.group share`
  - the mask `umask` is inherited from the father process

Modifications of the knowledge base: execution flags

- the system call `setuid(uid)` assigns
  the masterships `real uid`, `effective uid` and `saved uid`
  the parameter value `uid`:
  only executed for a process that satisfies the following precondition:
  the `effective uid` equals `superuser_uid`,
  or the `real uid` equals the parameter value `uid`
  (i.e., in the latter case, the original situation is restored)

- the system call `seteuid(euid)` assigns
  the current mastership `effective uid`
  the parameter value `euid`,
  which might be the `real uid` or the `saved uid`

- thereby, while executing a file with the execution flag `suid` set,
  a process can repeatedly change its `effective uid`:
  the process can select
  the `uid` of that `user` who has generated the original ancestor, or
  the `uid` of the `owner` of the file executed

Modifications of the knowledge base: some further manipulations

- the system calls `setgid(gid)` and `setegid(egid)`
  manipulate the group masterships
- the command `/bin/su -`
  changes the `effective uid` of the currently executed process
  into `superuser_uid`
  (thus the system administrator can acquire the mastership of any process):
  only executed if the issuer is successfully authenticated
  with the agreed password for the system administrator with username *root*
- the command `chown` changes the `owner` of a file:
  only executed for a process that satisfies the following precondition:
  the `effective uid` equals `superuser_uid` or
  equals the current `owner` of the file
- the command `chmod` changes the `access privileges` of a file:
  only executed for a process that satisfies the following precondition:
  the `effective uid` equals `superuser_uid` or
  equals the current `owner` of the file

Knowledge base on usage history

- basically, UNIX does not maintain an
  explicit *knowledge base* on the *usage history* for taking *access decisions*,
  except for keeping track of process generations

- most UNIX versions offer log services for *monitoring* that
  - produce *log data* about issued commands and executed system calls
  - store that data in special *log files*

Examples of UNIX log files

- the file `lastlog` contains the
  date of the last issuing of a `login` command for each of the registered users,
  whether successful or failed

- the file `loginlog` contains
  entries about all failed issuings of a `login` command,
  comprising the username employed, the physical device used and the date

- the file `pacct` contains entries about all issued commands,
  including their date

Examples of UNIX log files, continued

- the file `sulog` contains
  entries about all successful or failed attempts to issue the critical `su` command;
  for each attempt, the following is recorded:
  - success or failure
  - the username employed
  - the physical device used
  - the date

- the files `utmp` or `wtmp` contain
  entries about the currently active participants;
  in particular, the following is recorded:
  - the username employed
  - the physical device used
  - the process identification of the original ancestor process
    that was started by the `login` command
    to execute the user's command interpreter

Audit services

- log services send their log data as *audit messages* to an audit service that unifies and prepares that data for persistent storage or further monitoring

- the audit service `syslog` works on audit messages that are sent
  - by the kernel, exploiting `/dev/klog`
  - by user processes, exploiting `/dev/log`
  - by network services, exploiting the UDP port 514

- the audit messages consist of four entries:
  - the name of the *program* whose execution generated the message
  - a *classification* of the executing process into one of a restricted number of event sources, called *facilities*, which are known as *kern*, *user*, *mail*, *lpr*, *auth*, *daemon*, *news*, *uucp*, *local0*, …, *local7*, *mark*
  - a *priority level*, which is one of *emerg*(ency), *alert*, *crit*(ical), *err*(or), *warning*, *notice*, *info*(rmational), (from) *debug*(ging), *none*
  - the actual notification of the *action* that has occurred

Configuration of an audit service: example

- the system administrator can configure the audit service `syslog` using the file `/etc/syslog.conf`, which contains expressions of the form `facility.priority destination`

- such an expression determines how an audit message
  - that stems from an event source classified as `facility` and
  - has the level `priority` should be treated, i.e.,
  - to which `destination` it has to be forwarded

- `destination` might denote
  - the path name of a file
  - a username,
  - a remote address,
  - a pipe
  - the wildcard * (standing for all possible receivers)

Overall architecture

- control and monitoring are part of the operating system kernel
- the *kernel* realizes the system calls offered by UNIX
- a *system call* is treated roughly as follows:
  - the kernel checks the operator and the parameters of the call and then deposits these items in dedicated registers or storage cells
  - a software interrupt or trap dispenses the calling process
  - the program determined by the specified operator is executed with the specified parameters
  - if applicable, return values for the calling process are deposited
  - subsequently, the calling process can be resumed
- this procedure needs special hardware support for security: *storage protection*, *processor states*, *privileged instructions*, *process space separation*, ...
- most UNIX installations are part of a *network*, and thus employ various features for *securing the connections* to remote participants and the interactions with them

5 Discretionary Access Control and Privileges

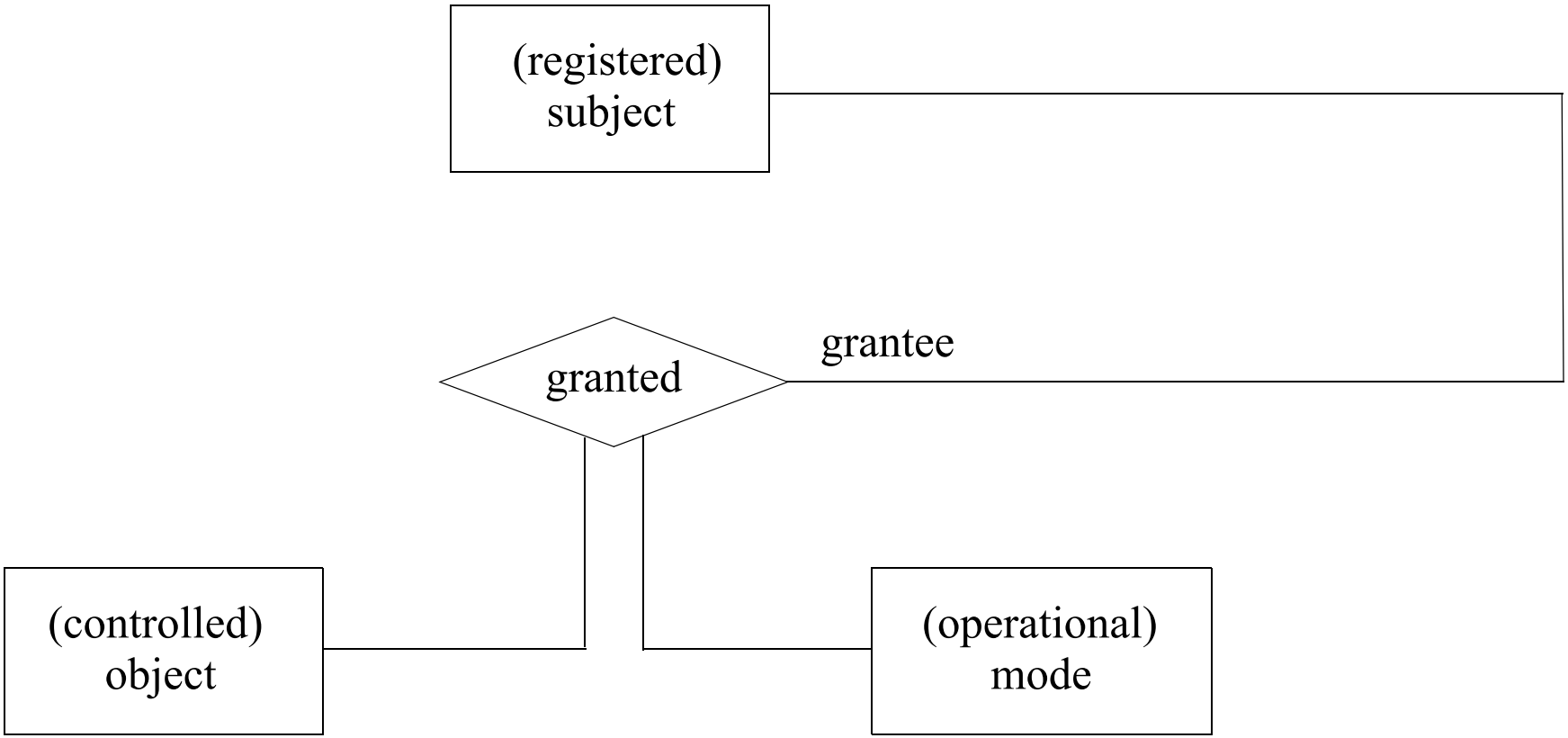
Permissions and prohibitions as discretionary access rights

- *access rights*:
  at least conceptually, maintained by an appropriate knowledge base

- *static aspects* of the knowledge base:
  structures for representing access rights

- *dynamic aspects* of knowledge base:
  operations on access rights:
  - *taking an access decision* (including *solving conflicts*)
  - *updating*
  - *analyzing* (determining the possible future instances under updates)

ER model of lookup representation



- an identifiable and registered *subject* that is a participant seen as a grantee
- a controlled *object* that is a possible operand of an access request
- an (operational) *mode* that signifies a set of operations on the object
- a relationship *granted* that a *subject* is permitted to perform any operation of a specified *mode* on an *object*

A relational implementation

- an *instance*:

Granted	Subject	Object	(Operational) Mode
	user	application	execute
	user	data_file	read
	user	recovery_file	write
	application	data_ file	read
	application	data_ file	write
	application	recovery_file	read

- *access decisions* by means of a simple lookup:

```
function decide(subject, object, operation): Boolean;
return (subject, object, mode(operation)) ∈ Granted.
```

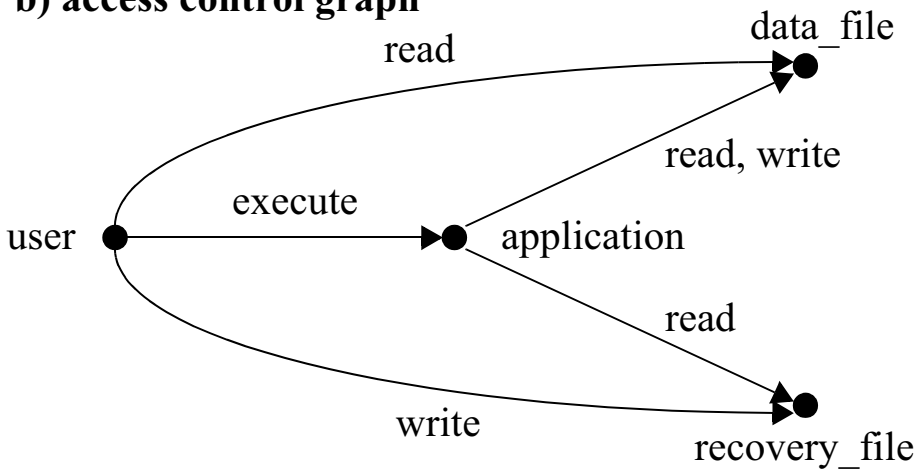
- *updates*:
  explicitly inserting, modifying or deleting tuples

Access control matrix/graph and privilege/access control lists

a) access control matrix

	application	data_ file	recovery_ file
user	execute	read	write
application		read, write	read

b) access control graph



c) privilege lists

Cl(user) = { [application,execute], [data_file,read], [recovery_file,write] }

Cl(application) = { [data_file,read], [data_file,write], [recovery_file,read] }

d) access control lists

Acl(application) = { [user,execute] }

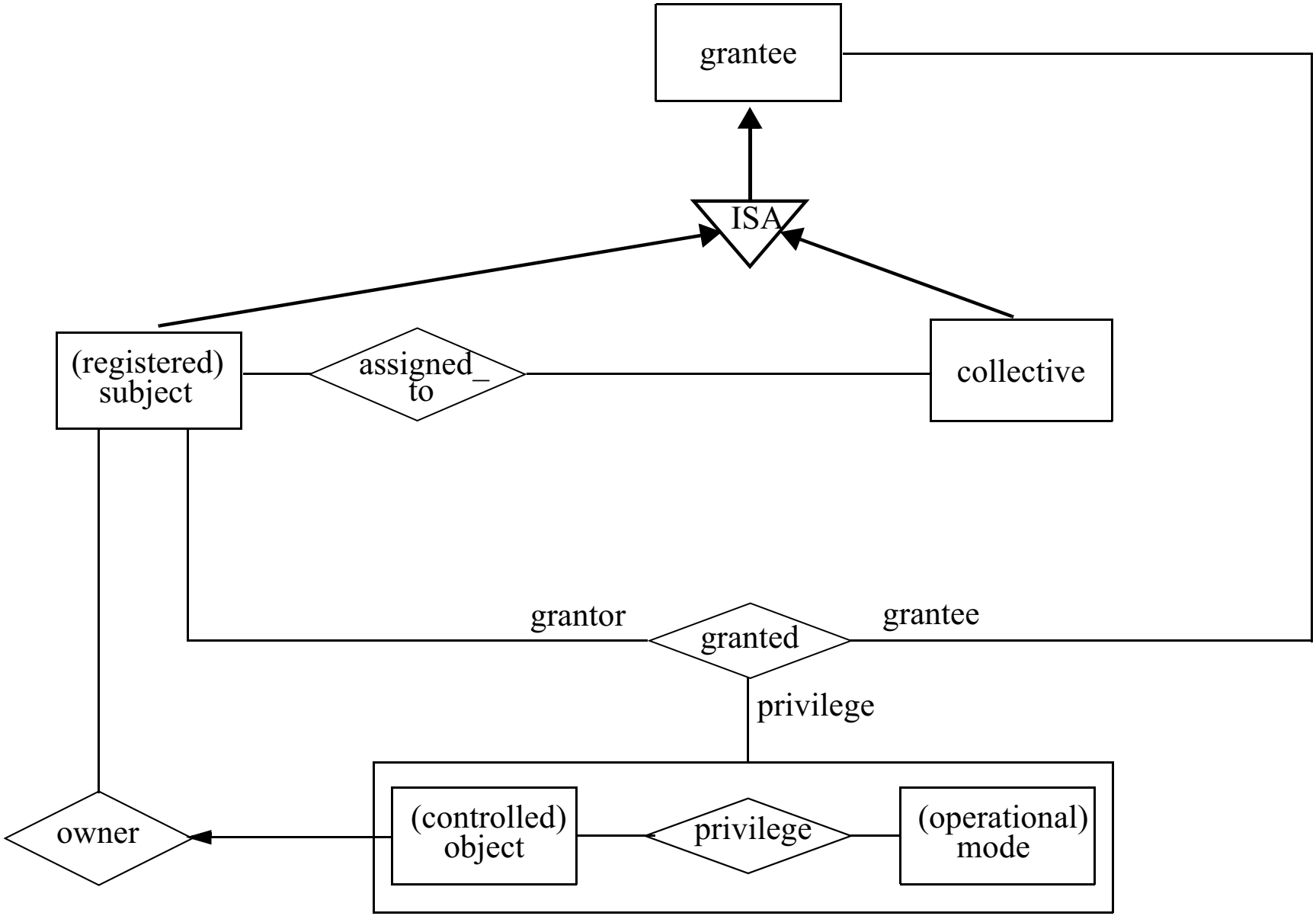
Acl(data_file) = { [user,read], [application,read], [application,write] }

Acl(recovery_file) = { [user,write], [application,read] }

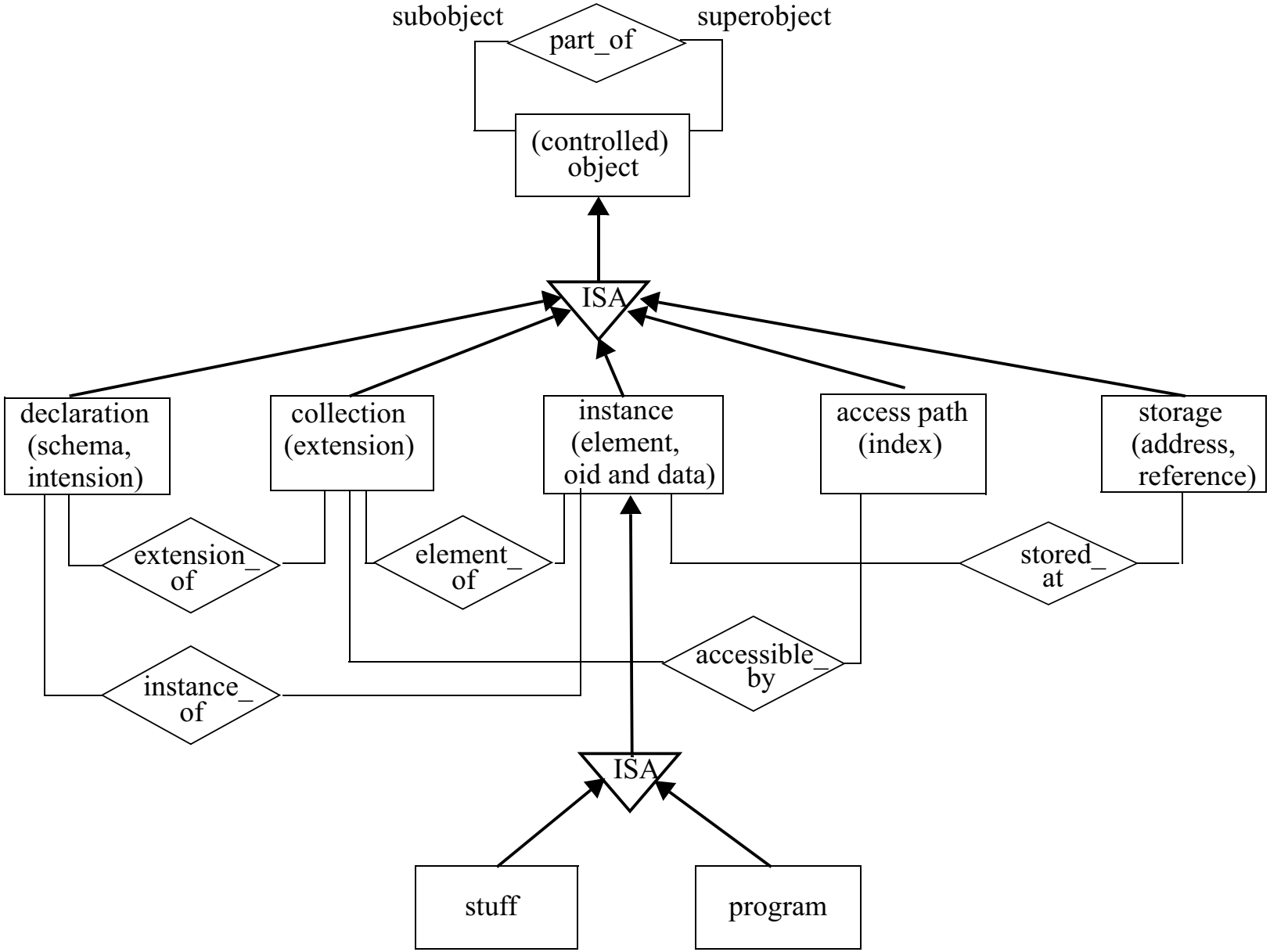
Some features of more sophisticated knowledge base structures

- *privilege*: aggregate of a controlled object and an operational mode
- *collectives*: grantee might be a
  - *group* (understood as set of equally treated participants)
  - *role* (seen as collection of privileges)
- *grantor*: might have an impact on access decisions or updates
- *owner*: assigned to a controlled object
- *relationships* on controlled objects (e.g., the *part_of relationship*) and *specializations* of the object class (in particular: executable *programs*)
- *structural relationships* and *specializations* for grantees and grantors
- *masterships*: a program is executed by a dynamically generated *process* that in turn is *mastered* by an individual participant
- *inclusion relationships* for the class of operational modes; specialization of modes into *functional* and *administrative* ones
- *usage constraints*: *temporal conditions*, conditions on *computing history*, ...
- *revocation semantics*: might have cascading effects, by using the *issue time*
- *negative privileges* (*access rights*): explicit prohibitions

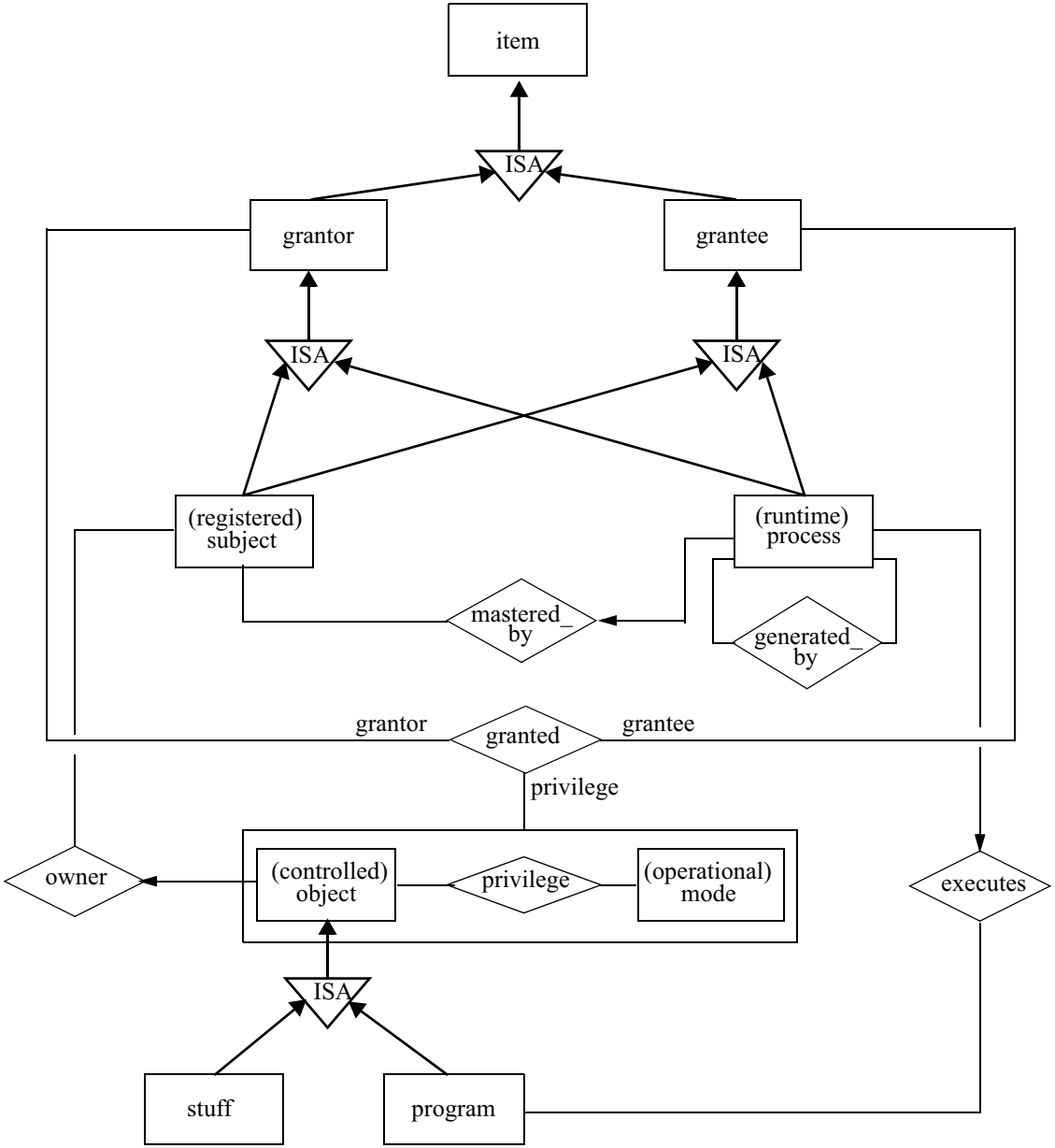
Refined ER model for permissions



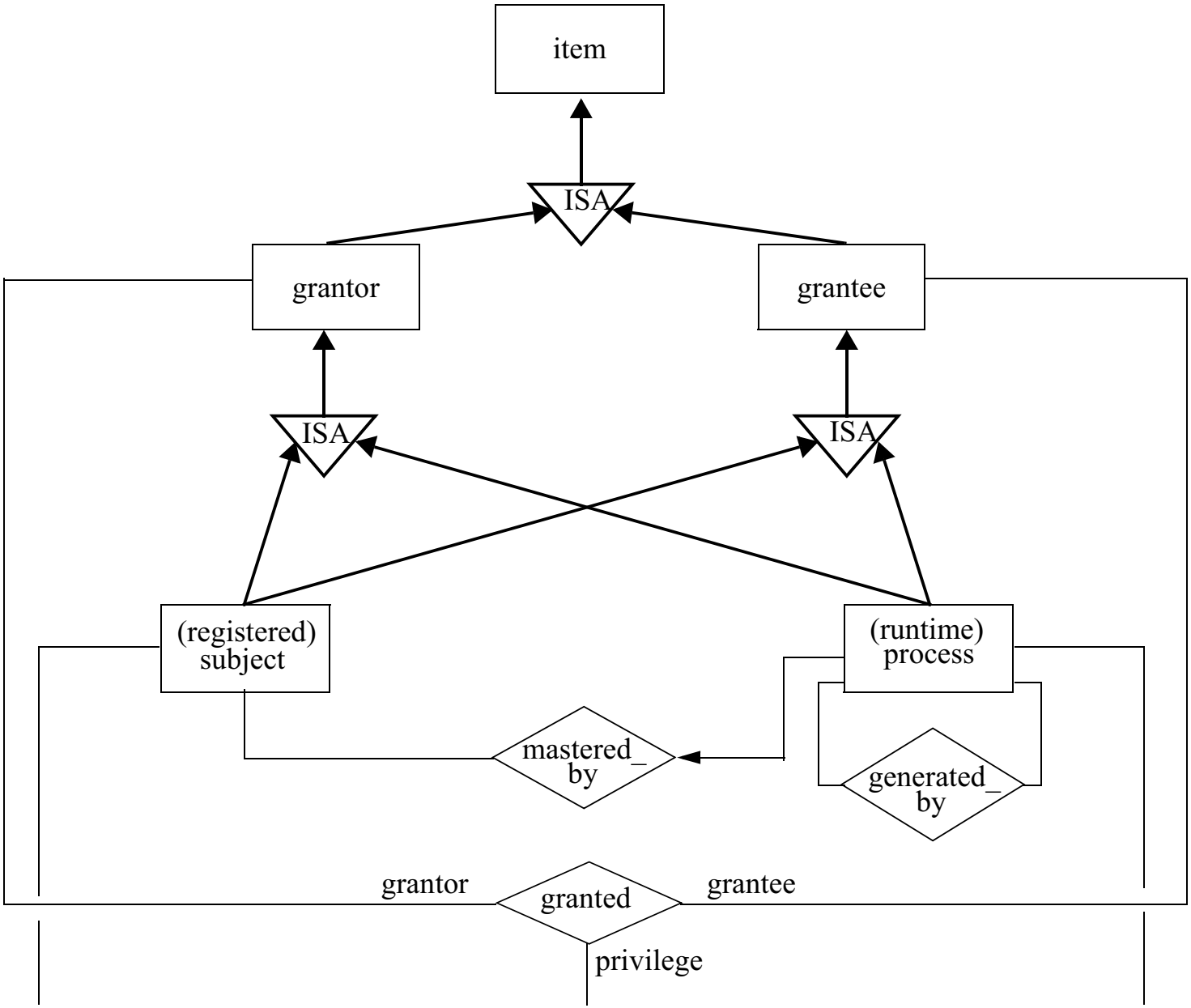
ER model of structural relationships and specializations of objects



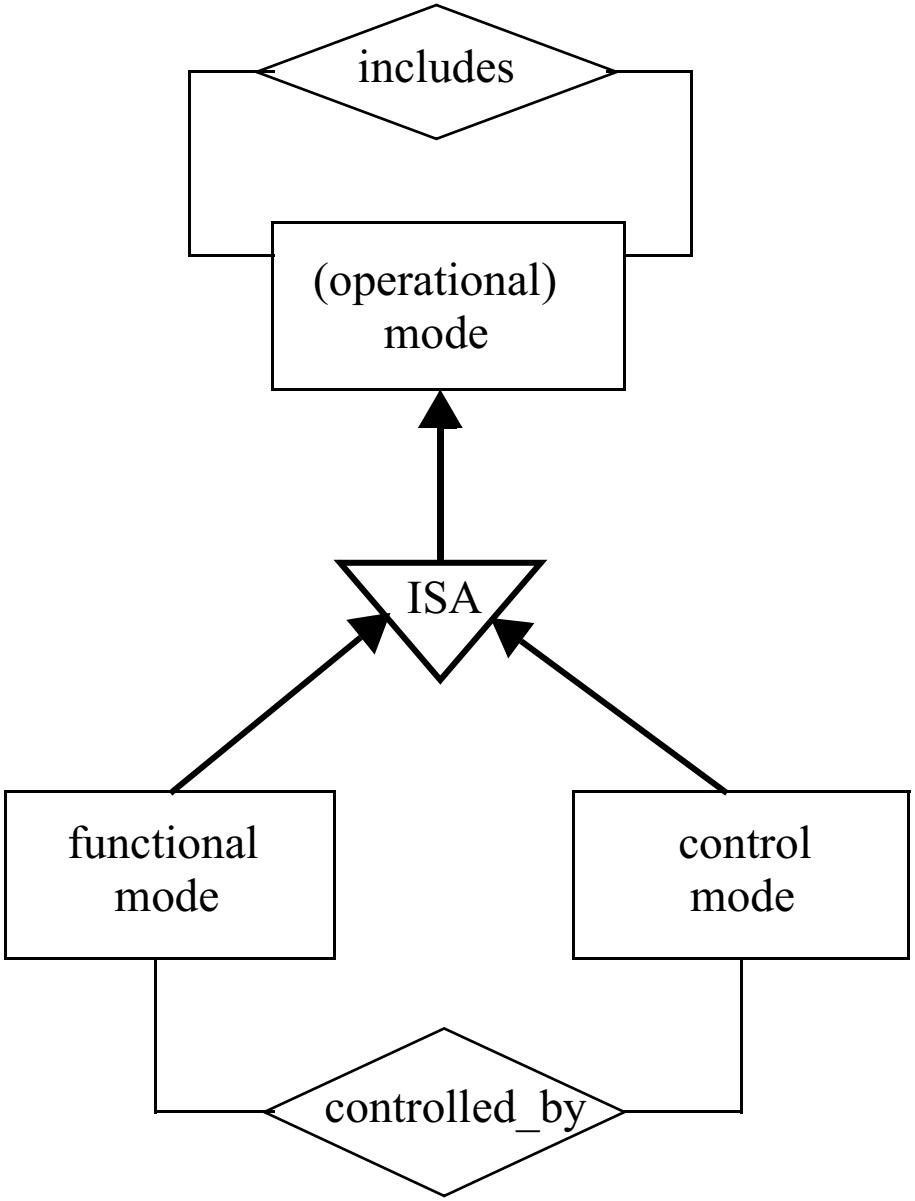
ER model of programs, processes and masterships



ER model of programs, processes and masterships (subpart)



ER model of operational modes



Functional modes in a pure object-oriented environment

- operations are called by sending, receiving and interpreting *messages*
- an object o_{act}, acting as a subject,
  is granted a permission to *invoke* an operation *op* on an object o_{exec}, i.e.,
  o_{act} is permitted to *send* a message to o_{exec},
  where the body of the message contains an identifier for the operation *op*
  (subject o_{act} sees the *message* as "controlled object" under operation *send*):
  o_{act} is the *activator* of an operation to be performed by o_{exec}
- the object o_{exec}, acting as a subject,
  is granted a permission to *interpret* a message *received* from the object o_{act}
  such that the operation *op* denoted in the body of the message
  is actually executed
  (subject o_{exec} sees the *message* as "controlled object" under
  operation *receive and interpret*):
  o_{exec} is the *executor* of an operation invoked by the object o_{act}

- *two* permissions are *independently* granted:
  - a *send* permission to the activator and
  - a *receive and interpret* permission to the executor

- appropriate in *distributed systems* with autonomous components acting as activators and executors:
  control and monitoring of *send* and *receive and interpret*
  can be implanted into the *channel* between the activators and the executors
  (like by *firewalls*)

- on the activator side, *outgoing* messages are controlled;
  on the executor side, the *incoming* messages are inspected

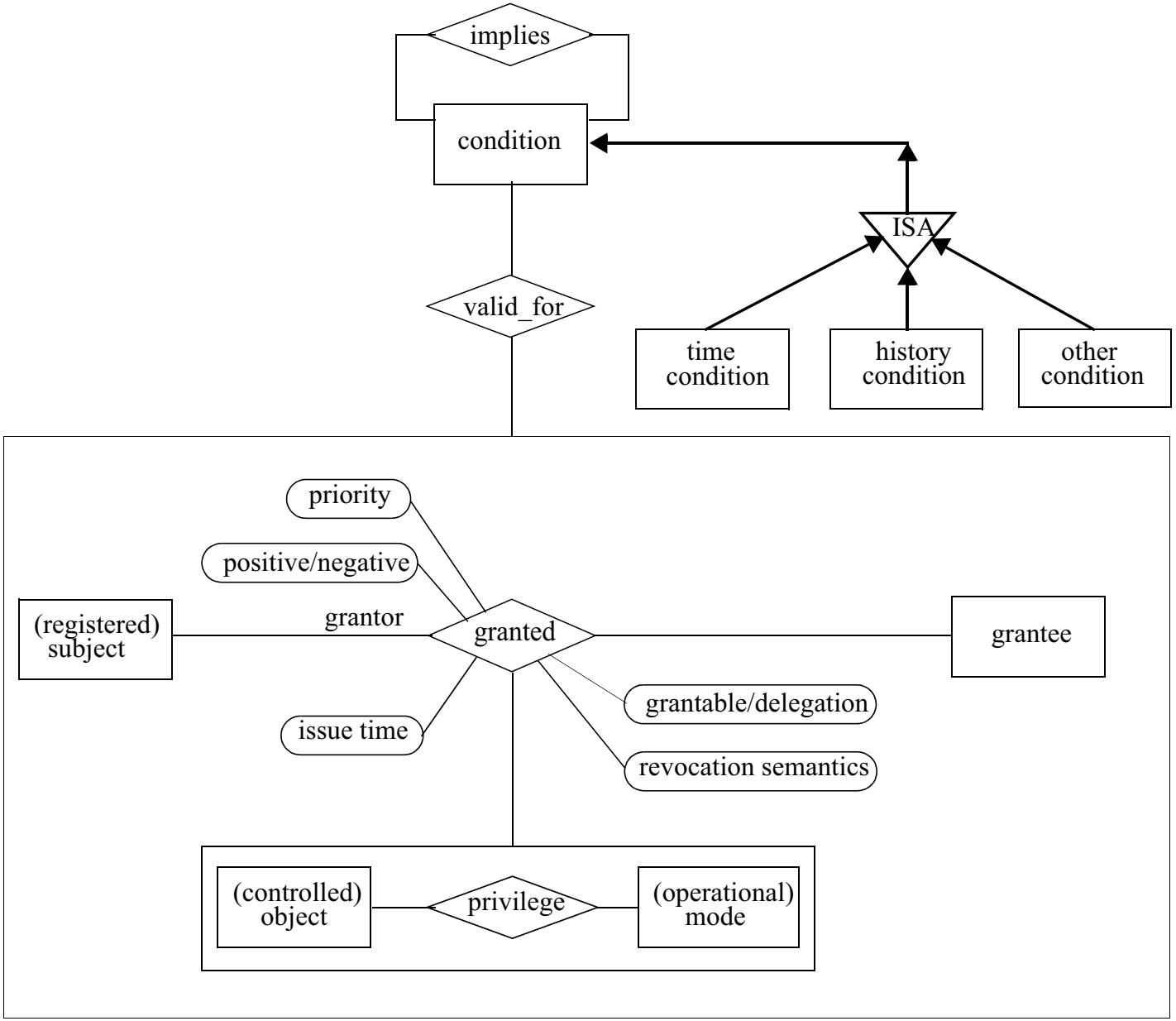
Control modes: examples

- *granting* a privilege to a subject as a grantee
- *transferring* a privilege to another subject
- *taking* a privilege from another subject
- *delegating* the usage of a privilege to another subject
- *revoking* a privilege from a subject

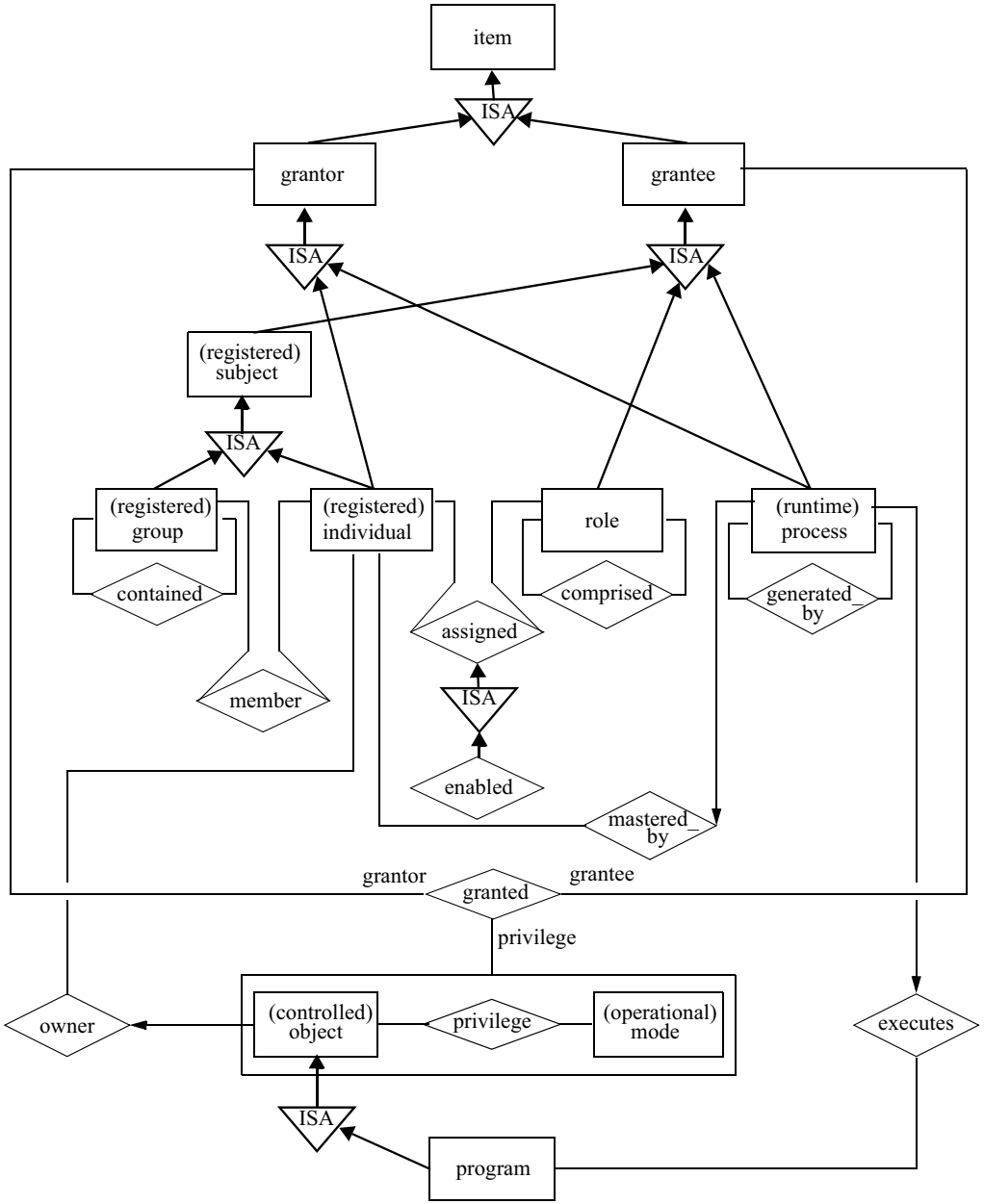
for controlling privileges, the following operation is also important:

- *generating* a new item:
  - classified as potentially acting as a subject, a controlled object or both
  - supplied with some initial privileges
  - accessible by some privileges given to the creator

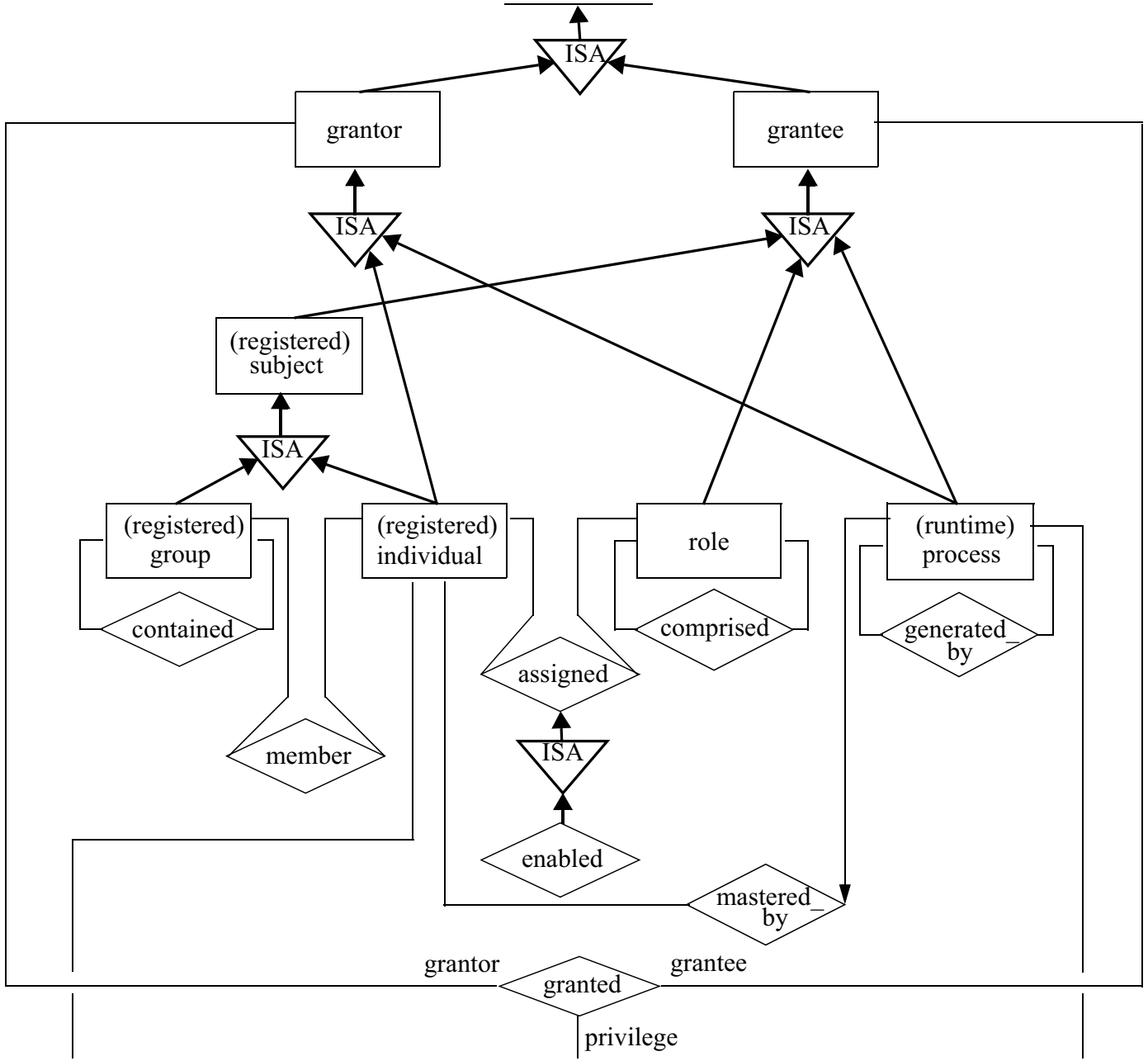
ER model of qualifications and conditions



ER Model of privileges with collectives



ER Model of privileges with collectives (subpart)



Role-based access control (RBAC)

- above: roles seen as an optional feature of discretionary access rights

- alternatively: *role-based access control*, *RBAC*, as specific approach:
  - rich body of insight and tools
  - widely used in practice
  - comprehensive treatment of implementation and application aspects

- role-based access control can be simulated
  by privileges directly granted to individuals,
  essentially by expanding all implicit inferences due to roles

- some features for privileges can also be employed for roles, e.g.:
  - *functional* and *control roles*, each having their own hierarchy
  - various *conditions*

Some specific pitfalls of RBAC

role concepts identified in the application environment
are not properly translated into computing concepts, e.g.:

- an individual is charged with many obligations and tasks,
  which can be partly overlapping and partly quite separate;
  simply defining one very powerful role for such an individual
  could violate *need-to-know*/*act* and *separation of roles*

- an organizational hierarchy is interpreted by *operational power*/*authority*:
  an individual acting in a higher organizational or social role is permitted
  to act like any individual in a lower organizational or social role
  (a *senior* might take the right to perform all actions
  that his *subordinates* are permitted to perform);
  naively translating this idea into roles
  can turn out to be extremely dangerous
  (an omnipotent user who is not well trained to operate the system)

Semantics for access decisions

- **conceptually:**
  decisions are taken by calling a function
  `decide(subject, object, operation): Boolean`

- **simplest case:**
  implemented by a lookup of a tuple
  `(subject, object, mode(operation))`
  in a table

- **more sophisticated cases:**
  complex inferences are necessary,
  based on various features managed by the knowledge base,
  foundation on precise semantics, in particular:
  - how to deal with *hierarchical relationships* between entities?
  - how to resolve *conflicts* between permissions and prohibitions?
  - how to always ensure a *defined decision*?

Inheritance rules for hierarchical relationships: examples

Hierarchical relationship	Permission	Prohibition
subrole $\leq_R$ superrole	upwards	downwards
subobject $\leq_O$ superobject	downwards	downwards
more special mode $\leq_M$ more general mode	downwards	upwards

Conflict resolution by priority rules: examples

- *prohibition prevails over permission*

- *specialization prevails over generalization*
  - considers a more special case as some kind of an *exception* to a larger case
  - if only permissions (*positive access rights*) are explicitly declared,
    then in accordance with the *default rule*:
    every request is prohibited
    unless it is explicitly proven to be permitted

- *higher-ranked grantor prevails over lower-ranked grantor*
  - sees grantors been ranked in a *command hierarchy*:
    orders of higher-ranked individuals invalidate
    conflicting orders of lower-ranked individuals

A metarule for priority rules

- in general, priority rules cannot uniquely resolve all conflicts:
  several rules might be equally applicable but deliver different results

- a simple *metarule* for a collection of priority rules:
  - consider the rules of the collection in a fixed *predetermined sequence*;
  - the result of the first applicable priority rule
    is taken as the final access decision

- this metarule can turn out to be rather dangerous:
  in general, the impact of sequencing rules
  is difficult to understand and to manage

Completion rules: examples

- *closed completion*:
  an undefined situation results in a final *prohibition*:

  a request is finally permitted
  only if
  a permission can be derived from the information in the knowledge base

- *open completion*:
  an undefined situation results in a final *permission*:

  a request is finally permitted
  not only if a permission can be derived
  but also if no prohibition can be derived

  in other terms:

  if a permission can be derived
  or
  if no prohibition can be derived

Requirements for formal specification language/formal semantics

- *expressiveness*: a rich variety of conceptual features is covered
- *manageability*: administrators can easily declare their wishes
- *completeness*: for any request, an access decision can be inferred
- *soundness*: for any request, the access decision is unique
- *computational efficiency*:
  access decisions and control operations
  can be implemented such that the
  storage overheads and runtimes
  are acceptable in practical applications

Flexible Authorization Framework: basic concepts

- *Inst*: set of *instance objects*
  Coll: set of *collections* or similar concepts
  $\leq_{IC}$: (for simplicity) common *hierarchy* (instance objects are minimal), denoting `element_of` relationships or `part_of` relationships
- *Ind*: set of *individual users*
  Gr: set of *groups*
  $\leq_{UG}$: (for simplicity) common *hierarchy* (users are minimal), denoting *group memberships* or *group containments*
- *Ro*: set of *roles*
  $\leq_{R}$: *hierarchy*, denoting *role comprising*
- *Mode*: set of operational *modes*
- $Rel_1, \ldots, Rel_n$: some relations $Rel_1, \ldots, Rel_n$ of appropriate arities, including the binary relation *Owner*

Flexible Authorization Framework: basic concepts (continued)

- $Grantee = Ind \cup Gr \cup Ro$: set of (possible) *grantees*
- $Object = Inst \cup Coll \cup Ro$: set of (possible) controlled *objects*
- $Qual = \{\texttt{pos},\texttt{neg}\}$: set of *qualifications*

- $QGranted \subseteq Grantee \times Object \times Mode \times Qual$:
  relation for *explicitly* declared `granted` relationships,
  qualified as *positive* (for permissions) or *negative* (for prohibitions)

  a role r can occur in a tuple of $QGranted$ in two different positions:

  – in (r, o, m, q),
    role r is grantee holding the privilege $[o, m]$ with qualification q

  – in (u, r, m, q) with $m \in \{\texttt{assign}, \texttt{enable}\}$,
    individual user u has role r assigned/enabled, qualified by q

Flexible Authorization Framework: basic concepts (continued)

- $Done \subseteq Ind \times Ro \times Object \times Mode \times Time$:
  relation for recording selected aspects (u, r, o, m, t) of the *usage history*:
  - an *individual user u* (assumed to have at most one role enabled)
  - acting in a *role r*
  - has operated on an *object o*
  - in some *mode m*
  - at a specific *time t*

Flexible Authorization Framework: concepts derived by rules

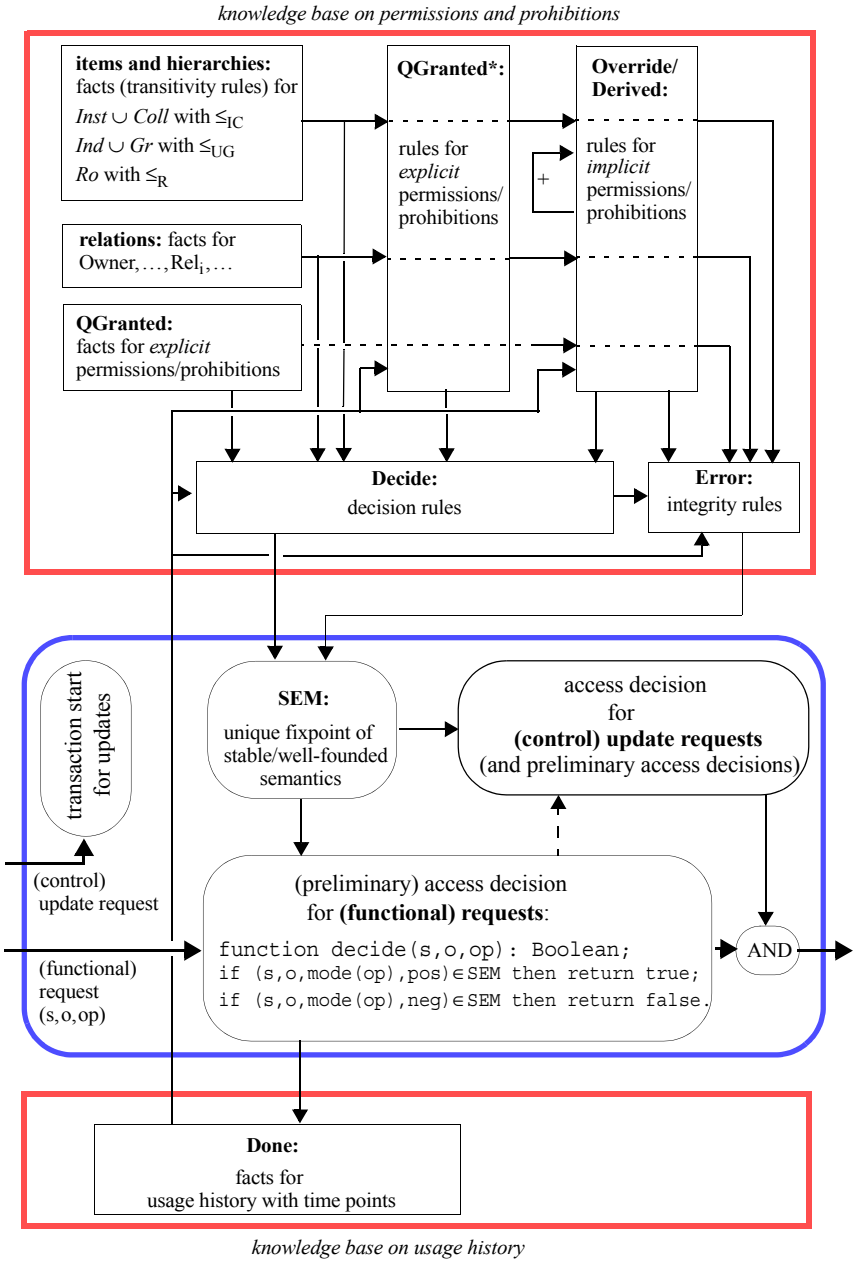
- $QGranted^* \subseteq Grantee \times Object \times Mode \times Qual$:
  relation for extending the relation *QGranted*
  by further *explicit qualified* `granted` relationships,
  which might be conditional in terms of basic items and the usage history

- $Derived \subseteq Grantee \times Object \times Mode \times Qual$:
  relation for representing *implicit* qualified permissions and prohibitions,
  where an auxiliary relation *Override* together with appropriate rules is used
  to prepare for *resolving conflicts*

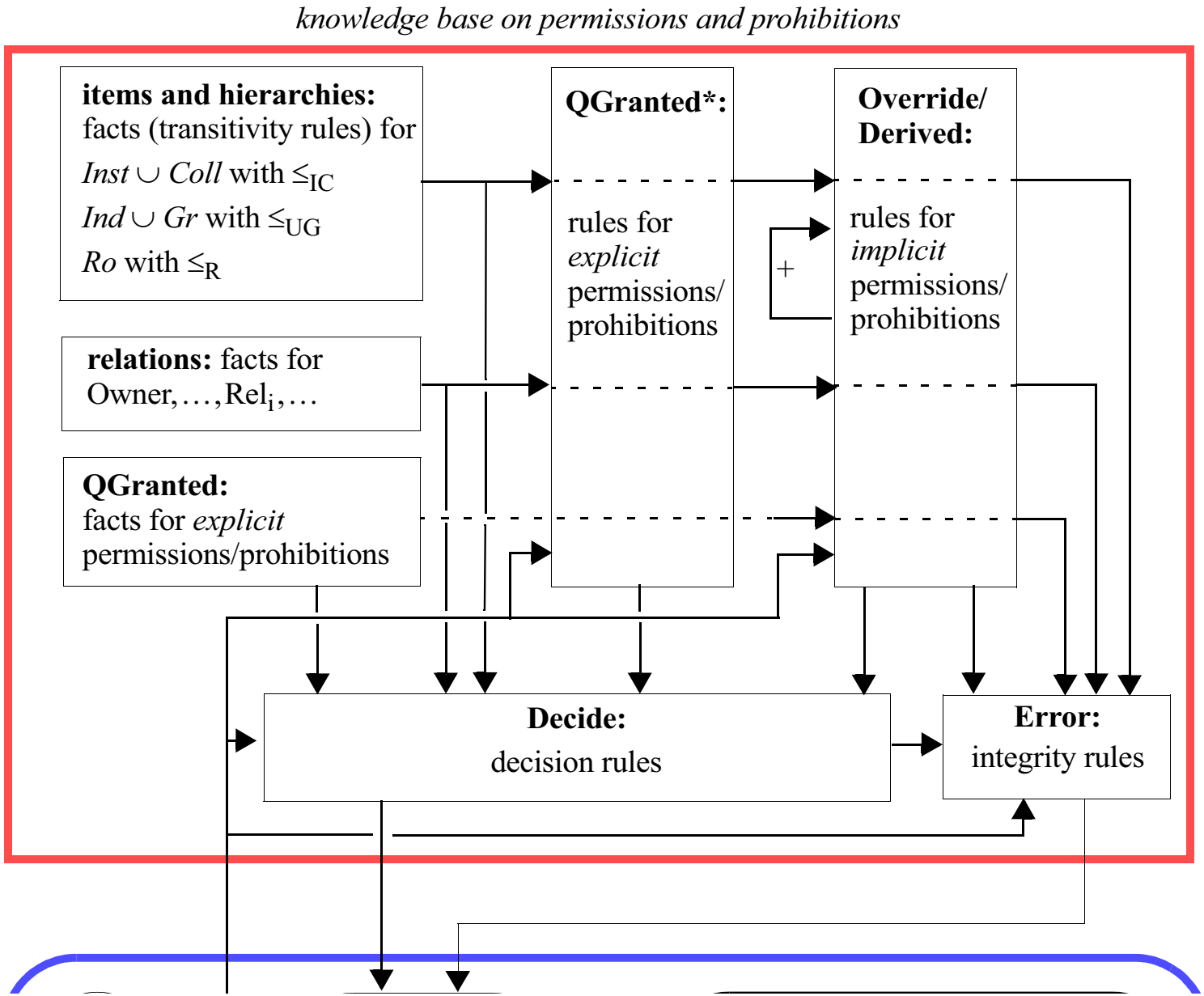
- $Decide \subseteq Grantee \times Object \times Mode \times Qual$:
  relation for representing the overall *security policy*,
  including final conflict resolution and enforcing completeness

- $Error \subseteq \{\varnothing\}$:
  relation (Boolean predicate) to detect erroneous specifications

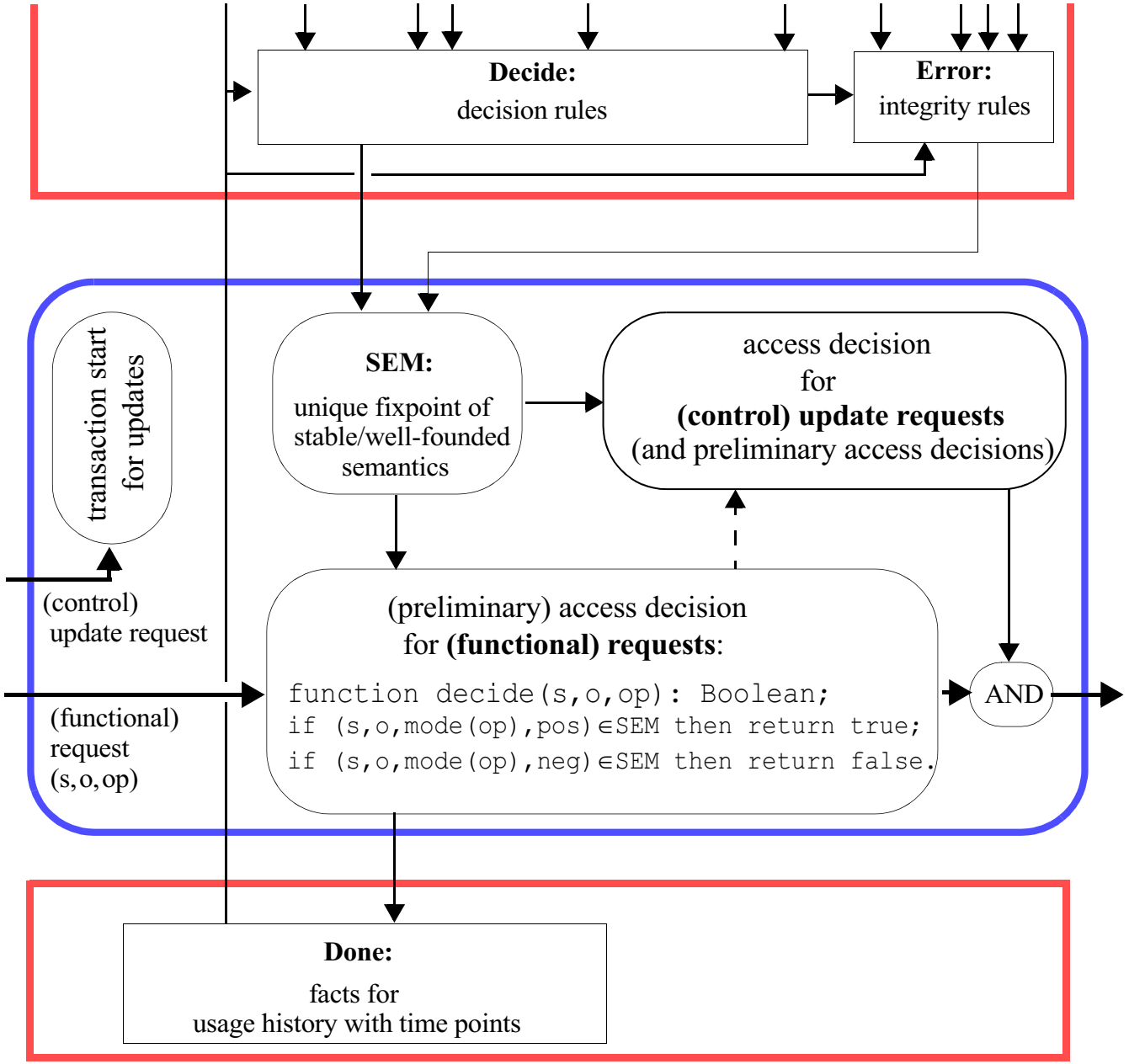
Architecture of FAF: overview



Architecture: knowledge base on permissions and prohibitions



Architecture: access decisions



Syntax of Flexible Authorization Specification Language: outline

- vocabulary:
  - sorted *constant symbols* for any item occurring in the computing system
  - sorted *variables* for such items
  - sorted *predicate symbols* for the components
- *terms*: either constants (no further function symbols) or variables
- *atoms*: formed by a predicate symbol followed by a list of terms;
  literal: either an atom or a negated atom (written as $\neg atom$)
- *rules*: implicational formulas of the form
  $atom \leftarrow literal_1 \wedge \ldots \wedge literal_n.$

  conclusion (head): single atom;
  premise (body): conjunction of atoms and,
  under some essential restrictions, of negated atoms
- *facts*: rules of the form $atom \leftarrow$.
- *strata*: 6-level dependency structure of rules,
  as roughly indicated in the architecture
- *program*: finite set of rules

Strata of logical program in FASL

Stratum	Head	Body	Goal
1	Inst(t), Coll(t), Ind(t), Gr(t), Ro(t)	empty	facts for basic items
	$\leq_{IC}(t_1,t_2)$, $\leq_{UG}(t_1,t_2)$, $\leq_{R}(t_1,t_2)$	empty, or the respective relation symbols	facts and recursive closure rules for hierarchies
	Owner(t_1,t_2), …	empty	facts for relations
	QGranted(t_1,t_2,t_3,t_4)	empty	facts for explicit `granted` relationships
	Done(t_1,t_2,t_3,t_4,t_5)	empty	facts for usage history
2	QGranted*(t_1,t_2,t_3,t_4)	literals for basic items, hierarchies, usage history	rules for explicit, `granted` relationships with conditions

Stratum	Head	Body	Goal
3	Override(t_1,t_2,t_3,t_4)	(not treated in this text)	rules for preparing conflict resolution
4	Derived(t_1,t_2,t_3,t_4)	literals for basic items, hierarchies, usage history, explicit `granted` relationships, conflict resolution; *atoms* for implicit `granted` relationships	rules for implicit `granted` relationships recursive rules for implicit `granted` relationships

Stratum	Head	Body	Goal
5	Decide(t_1,t_2,t_3,`pos`)	literals for basic items, hierarchies, usage history, explicit and implicit `granted` relationships;	decision rules for ***final permissions***
	Decide(x,y,z,`neg`) as head of a *single* rule with *variables* x,y,z	the *single* literal ¬Decide(x,y,z,`pos`)	one default decision rule for ***final prohibitions***
6	Error()	literals for basic items, hierarchies, usage history, explicit and implicit `granted` relationships, decisions	integrity rules

Semantics of a logical program in FASL

- semantics is determined as the unique minimal *fixpoint* of the program, with respect to *stable/well-founded semantics* for *locally stratified* programs
- a rule (under a suitable substitution of variables by constant symbols) generates a new head fact from previously available body facts
- the rules of each stratum are exhaustively treated before proceeding to the next stratum
- negative atoms from preceding strata are always treated according to *negation as failure*
- in stratum 5, the negative atom ¬`Decide(x,y,z,pos)` is first determined by negation as failure, and then the single rule for final prohibitions is used
- concerning negation, there is a difference between
  - "negation of a permission" (a negated atom with qualification `pos`)
  - "prohibition" (an atom with qualification `neg`)
- stratum 5 finally resolves conflicts that may have potentially occurred in preceding strata

A simple fragment of a security policy in FASL: scenario

- reading and writing a file `pub_f` of low sensitivity
- an individual user *admin* acting as administrator
- arbitrary *requestors* denoted by the variable *x*
- arbitrary operational modes denoted by the variable *m*

A policy: explicit permissions/prohibitions in strata 1 and 2

administrator: granted a positive read privilege
but a negative write privilege

owner of the file: acquires positives read and write privileges:

```
QGranted(admin,pub_f,read,pos) ← .
QGranted(admin,pub_f,write,neg) ← .

QGranted*(x,pub_f,read,pos) ← Owner(x,pub_f).
QGranted*(x,pub_f,write,pos) ← Owner(x,pub_f).
```

A policy: implicit permissions/prohibitions in stratum 4

operational modes:

- `read` is considered to be included in `write`
- corresponding inheritance rules are instantiated
- explicit statements are converted in implicit ones:

```
Derived(x,pub_f,read,pos) ← Derived(x,pub_f,write,pos).
Derived(x,pub_f,write,neg) ← Derived(x,pub_f,read,neg).

Derived(x,pub_f,m,pos) ← QGranted(x,pub_f,m,pos).
Derived(x,pub_f,m,pos) ← QGranted*(x,pub_f,m,pos).
Derived(x,pub_f,m,neg) ← QGranted(x,pub_f,m,neg).
Derived(x,pub_f,m,neg) ← QGranted*(x,pub_f,m,neg).
```

A policy: decisions and conflict resolution in stratum 5

- read accesses are finally permitted
  if some implicit permission can be derived or
  if an implicit prohibition cannot be derived
- write accesses are finally permitted only in the former case
- thus, for both modes, a permission prevails over a prohibition
- while an *open* policy is stated for reading
  (finally permitted if no prohibition can be derived), and
  a *closed* policy is preferred for writing
  (finally permitted only if a permission can be derived):

```
Decide(x,pub_f,read,pos) ← Derived(x,pub_f,read,pos).
Decide(x,pub_f,read,pos) ← ¬Derived(x,pub_f,read,neg).
Decide(x,pub_f,write,pos) ← Derived(x,pub_f,write,pos).
```

- *prohibitions*, as generally required for stratum 5 of any logical program in FASL, one generic default decision rule is specified:

```
Decide(x,y,z,neg) ← ¬Decide(x,y,z,pos).
```

A policy: integrity enforcement in stratum 6

an implicit permission of a read or write access to the file `pub_f`
together with the respective implicit prohibition is treated as an error, i.e.,
any update request resulting in such a situation should be rejected):

$$\texttt{Error()} \leftarrow \texttt{Derived(x,pub\_f,read,pos)} \wedge \texttt{Derived(x,pub\_f,read,neg).}$$
$$\texttt{Error()} \leftarrow \texttt{Derived(x,pub\_f,write,pos)} \wedge \texttt{Derived(x,pub\_f,write,neg).}$$

Access decision on a functional request

- (*functional*) *request* (s,o,op) with
  - s is an individual requestor u or, if applicable, his enabled role r
  - op is a wanted operation on a controlled object o such that $m = mode(op)$
- the unique minimal *fixpoint SEM* of the logical program is computed
- a preliminary *access decision* is taken:

```
function decide(s,o,op): Boolean;
if (s,o,mode(op),pos)∈SEM then return true  fi;    / permitted
if (s,o,mode(op),neg)∈SEM then return false fi.    / prohibited
```

- if preliminary access decision returns `false`:
  request is immediately rejected;
- otherwise:
  - an appropriate tuple (u,r,o,m,t) is tentatively inserted into *Done*,
  - fixpoint is recomputed and checked for integrity
- if the integrity is preserved:
  preliminary permission is confirmed/tentative insertion is committed;
  otherwise: request is rejected/tentative insertion is aborted

Access decision on an update request (control operation)

- an access decision is taken,
  similarly to what is done for a functional request
- a transaction is started
- the requested modifications are tentatively executed,
  allowing various *revoking* strategies to be implemented
- the *Error* predicate for checking *integrity* is evaluated
  using the fixpoint *SEM*
- depending on the result of the integrity check,
  the transaction either commits or aborts

Strata, goals and responsible agents

1	facts for basic items, hierarchies, relations facts for explicit `granted` relationships facts for usage history	automatic extraction from declarations and runtime data system administrator and respective owners monitoring component
2	rules for explicit `granted` relationships	respective owners and application administrator
3	rules for preparing conflict resolution	application administrator and security officer
4	(recursive) rules for implicit `granted` relationships	application administrator and security officer
5	decision rules for permissions and prohibitions	security officer
6	integrity rules	application administrator and security officer

Basic properties of FAF

- *expressiveness*
  - by design, many features of access control can be formally treated
  - determined by the power of the chosen fragment of logic programming

- *manageability*
  - layered approach supports reliable administration of access rights, even if the administration is not centralized but partially distributed
  - for example, responsibilities can be reasonably assigned to
    - system components
    - several individual owners
    - a system administrator
    - an application administrator
    - a security officer

- *completeness* and *soundness*
  - ensured by stratification and the restrictions imposed

FASL programs are complete and sound: theorem

Let *AS* be a logical program
according to the syntax of the Flexible Authorization Specification Language.

The following properties then hold:

- *AS* has a unique minimal fixpoint *SEM* as a stable/well-founded model.
- For each (functional) request (s, o, op),
  exactly one of the literals

  $(s, o, mode(op), \texttt{pos})$ and $(s, o, mode(op), \texttt{neg})$

  is an element of *SEM*.

Proof idea

- existence of a unique minimal fixpoint is ensured by local stratification, i.e., the restrictions concerning negation
- completeness is enforced by the default decision rule for prohibitions together with negation as failure
- soundness is a consequence of having just one default decision rule for prohibitions

Properties of FAF: efficiency

- general design allows
  tractable (polynomial-time computable) access decisions

- if advanced techniques of logic programming are employed,
  including *materialization* of the fixpoint *SEM*,
  then decisions with an acceptable delay appear to be achievable

6 Granting and Revoking, and Analysis

Granting

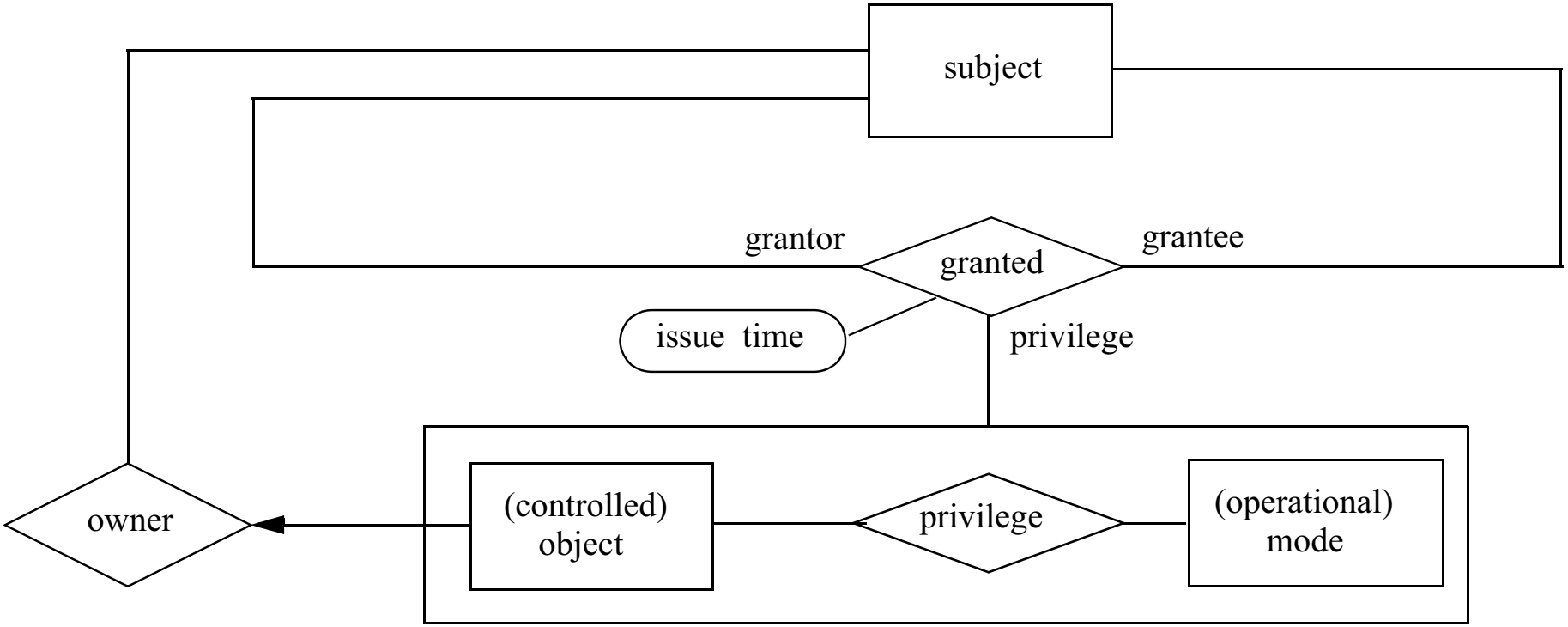
- a current holder of a privilege, as a grantor,
  assigns this privilege to a subject as a further grantee
- in doing so, the grantor can declare the privilege to be grantable again
- the following options for a *grantable* attribute can be meaningful:
  no: receiver must not grant the received privilege further
  limited: receiver may grant the received privilege further,
  under the provision that the *grantable* attribute is then set to *no*
  unlimited: receiver may grant the received privilege further,
  without any restrictions
- a privilege can ultimately be held by many grantees
- a single grantee might have repeatedly received a privilege in several ways,
  from different *grantors* and
  at different *issue times*

A model with simplifying assumptions

- originally: a privilege is held only by the *owner* of the object concerned
- later on: all grantings are recorded with the *issue time* and permit further *unlimited* grantings
- this model can be implemented as a database relation *KB* with five attributes:
  - (*Issue*) *Time*
  - *Grantor*
  - *Grantee*/*Subject*
  - (*Controlled*) *Object*
  - (*Operational*) *Mode*
- a tuple $(t, g, s, o, m) \in KB$ means:
  at the issue time t,
  a grantor g has assigned
  to the grantee/subject s a privilege with respect to
  the controlled object o
  for the operational mode m

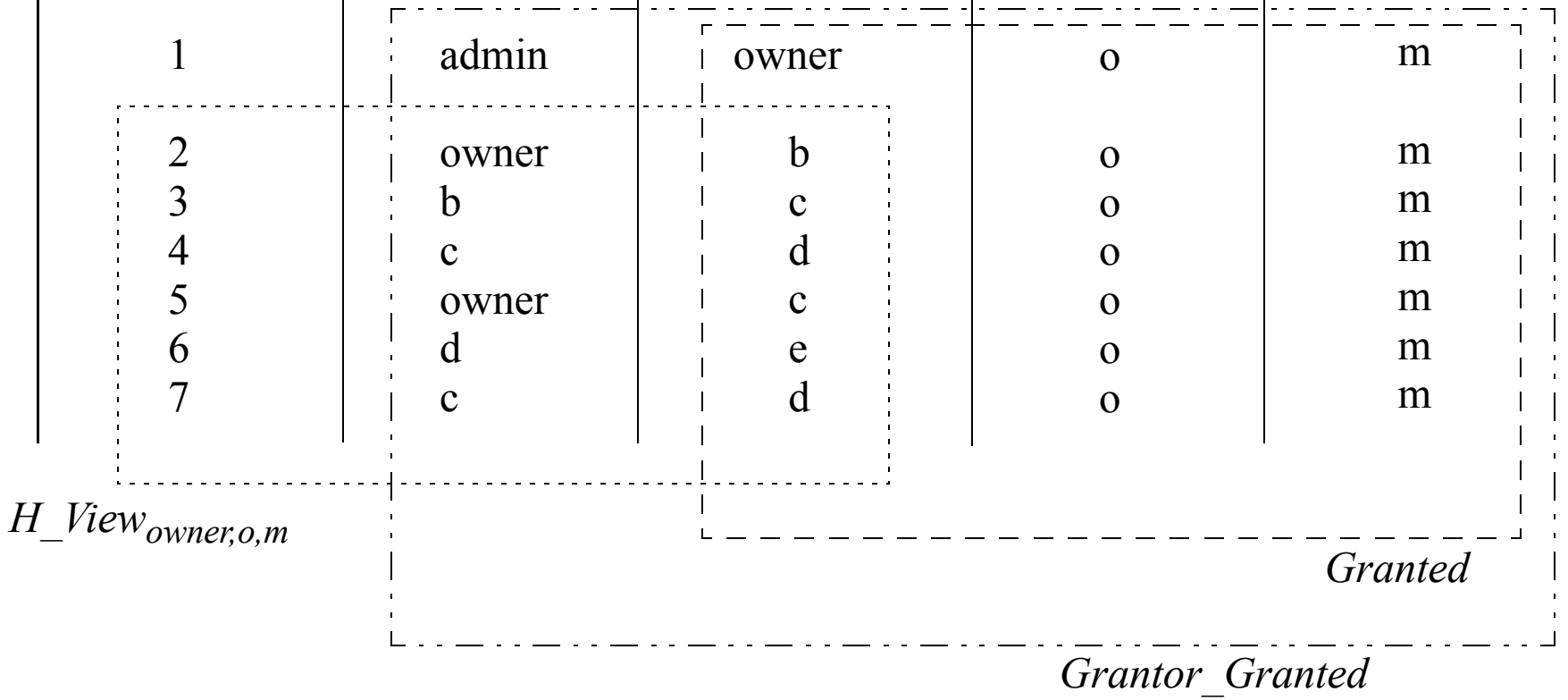
  (where the special mode `own` indicates ownership)

An ER model for grantings



An instance of a relational implementation

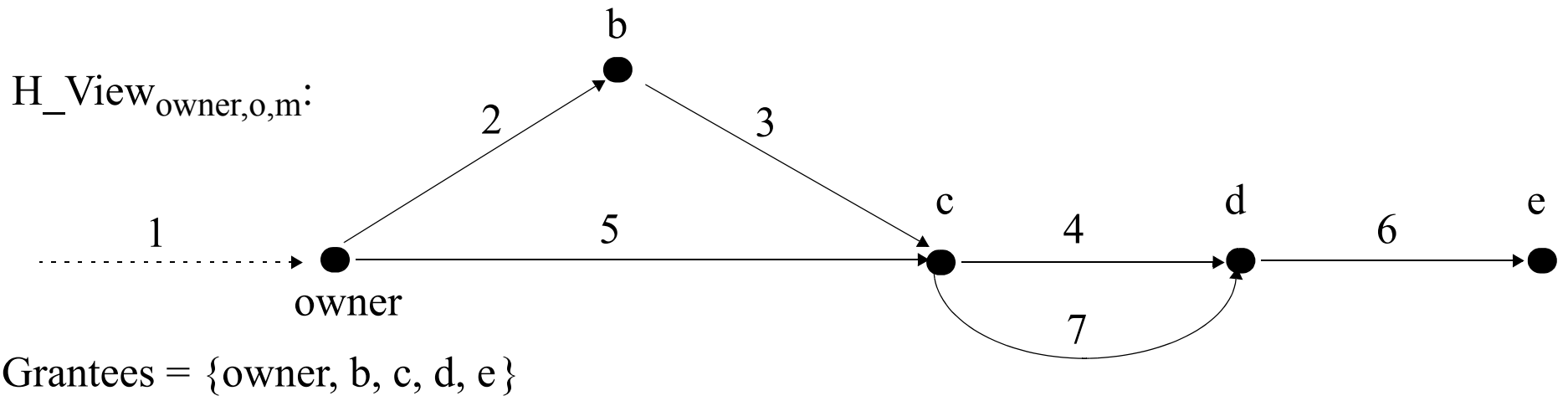
KB	Time	Grantor	Grantee/ Subject	Object	Mode
	0	admin	owner	o	own
	1	admin	owner	o	m
	2	owner	b	o	m
	3	b	c	o	m
	4	c	d	o	m
	5	owner	c	o	m
	6	d	e	o	m
	7	c	d	o	m



A grant graph corresponding to a history subrelation

- a subrelation $H\_View_{owner,o,m}$ exhibits the full *history* of grantings
  for a single privilege $[o,m]$
  that originate directly or indirectly from the subject *owner*

- a corresponding *grant graph*
  represents each triple $(time, grantor, grantee) \in H\_View_{owner,o,m}$
  by a labeled, directed edge with
  - origin *grantor*
  - target *grantee*
  - label *time*

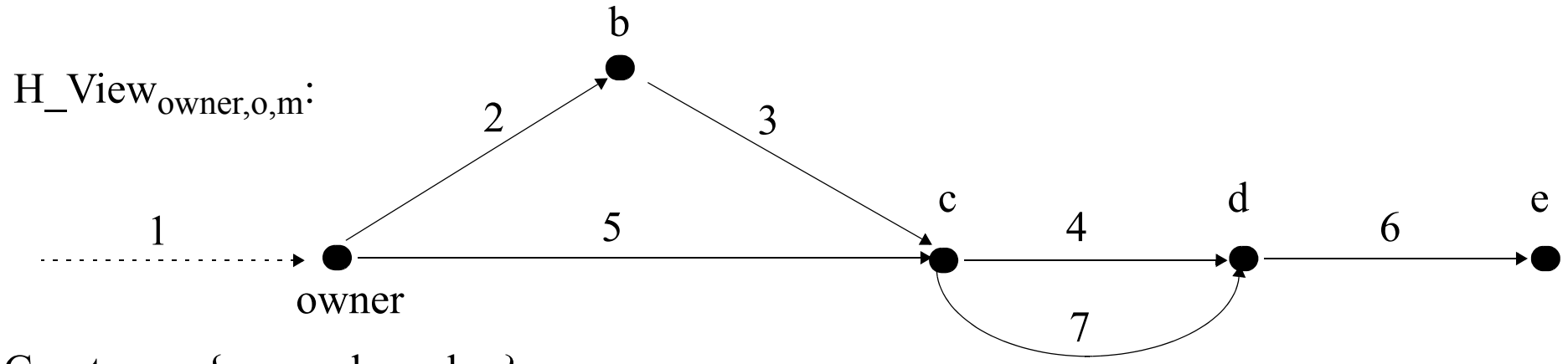


Grantees = {owner, b, c, d, e}

A formalization of granting

```
procedure grant_owner,o,m(time,grantor,grantee);
 { precondition:  owner ∈ Grantees;
   import:        Grantees, H_View, t_max
 }
if                                          / access decision:
 [grantor = owner                           / owner is always permitted
  OR
  EXISTS t, EXISTS x ∈ Grantees: (t,x,grantor) ∈ H_View
                                            / a current holder is permitted
 ]
 AND t_max < time                           / issue times are monotone
 AND grantor ≠ grantee                      / no self-granting
 AND grantee ≠ owner                        / no grantings for owner
then                                        / updating of grant graph:
 Grantees := Grantees ∪ {grantee};  / insert grantee
 H_View := H_View ∪ {(time,grantor,grantee)}
                                            / insert privilege with issue time
fi.
```

Producing a grant graph: example



Grantees = {owner, b, c, d, e}

is produced by the following calls,
where all requested updates have been permitted:

```
grant owner,o,m( 2 , owner, b )
grant owner,o,m( 3 , b    , c )
grant owner,o,m( 4 , c    , d )
grant owner,o,m( 5 , owner, c )
grant owner,o,m( 6 , d    , e )
grant owner,o,m( 7 , c    , d )
```

Options for revocation semantics: examples

Option	Knowledge base	Precondition for permission/invariant	Postcondition for knowledge base
simple deletion	*Granted*	revoker is administrator or owner	granting is completely deleted
grantor-specific deletion	*Grantor_ Granted*	revoker has been grantor	granting of revoker is deleted
deletion with renewed further grantings	*Grantor_ Granted*	revoker has been grantor invariant: unique grantor, and existence of unique granting chain from owner	granting is deleted, and further grantings are renewed
deletion with deleted further grantings	*Grantor_ Granted*	revoker has been grantor invariant: existence of granting chains from owner	granting is deleted, and invariant is satisfied
time-specific deletion with recursive revocation of further grantings	*KB* (all *H_View* instances)	revoker has been grantor invariant: existence of time-increasing granting chains from owner	*KB´* is the instance that would be produced if revoker had never granted the privilege to grantee

Simple deletion

- request:
  revoker r wants to revoke privilege $[o,m]$ from grantee s at time t

- precondition for a permission:

  $r = admin$

  or

  $(r,o,own) \in Granted$

- postcondition for knowledge base:

  $(s,o,m) \notin Granted'$

- implementation:

  `Granted := Granted \ {(s,o,m)}`

Grantor-specific deletion

- request:
  revoker r wants to revoke privilege $[o,m]$ from grantee s at time t

- precondition for a permission:
  $(r,s,o,m) \in Grantor\_Granted$

- postcondition for knowledge base:
  $(r,s,o,m) \notin Grantor\_Granted'$

- implementation:
  `Grantor_Granted := Grantor_Granted \ {(r,s,o,m)}`

Deletion with renewed further grantings

- precondition for a permission to revoker r:
  $(r,s,o,m) \in Grantor\_Granted$
- invariant for knowledge base:
  existence of unique granting chains from owner to grantees
- postcondition for knowledge base:
  $(r,s,o,m) \notin Grantor\_Granted'$
  and
  for all $y \neq r$ with $(s,y,o,m) \in Grantor\_Granted$:
  $(s,y,o,m) \notin Grantor\_Granted'$ and $(r,y,o,m) \in Grantor\_Granted'$
- implementation:

```
Grantor_Granted := Grantor_Granted \ {(r,s,o,m)};
forall y do
if  (s,y,o,m) ∈ Grantor_Granted  AND  y ≠ r
then
     Grantor_Granted := Grantor_Granted \ {(s,y,o,m)};
     Grantor_Granted := Grantor_Granted ∪ {(r,y,o,m)}
fi.
```

Deletion with deleted further grantings

- request:
  revoker r wants to revoke privilege $[o,m]$ from grantee s at time t

- precondition for a permission to revoker r:
  $(r,s,o,m) \in Grantor\_Granted$

- invariant for knowledge base:
  existence of granting chains from owner to grantees

- postcondition for knowledge base:
  $(r,s,o,m) \notin Grantor\_Granted'$
  and the invariant

- (sketch of) implementation:
  - the entry (r,s,o,m) is deleted in $Grantor\_Granted$
  - apply a graph search algorithm
    to enforce the invariant by minimal further deletions

Time-specific deletion with recursive revocation of further grantings

- precondition for a permission to revoker r:
  $(r,s,o,m) \in Grantor\_Granted$

- invariant for knowledge base:
  existence of issue time respecting granting chains from owner to grantees

- (informal) postcondition for knowledge base:
  KB´ is the instance
  that would have been produced
  if the revoker r had never granted the privilege $[o,m]$ to the grantee s

- discussion of implementation:
  - needs enough information to allow one to construct
    fictitious instances of the knowledge base
    that could have been produced *in the past*
  - the information represented in *KB* suffices
  - this information is also necessary

Recursive revocation

```
procedure revoke_owner,o,m(time,revoker,grantee);
{ precondition: owner ∈ Grantees;
  import:       Grantees, H_View, t_max}
                                / at time the revoker invalidates his grantings
                                / of privilege [o,m] concerning object o of owner
                                / to grantee
if                                  / access decision:
  t_max < time                      / issue times are monotone
then                                / updating of grant graph:
  revoke*(time,revoker,grantee);    / first call of recursive auxiliary procedure
  delete isolated elements from Grantees except owner
fi.

procedure revoke*(t,x,y);       / recursive auxiliary procedure for revoke_owner,o,m
{ precondition: owner ∈ Grantees;
  import:       Grantees, H_View, t_max}
if                                  / access decision:
  EXISTS t_early: t_early < t  AND  (t_early,x,y) ∈ H_View
                                / x has granted privilege to y before time t
then                                / updating of grant graph:
  H_View := H_View \ {(t_early,x,y) | t_early < t };
                                / delete grantings from x to y before time t

  VALID := { t_other | EXISTS x_other: (t_other,x_other,y) ∈ H_View };

  if VALID ≠ ∅ then t := minimum(VALID) else t := ∞ fi;
                        / compute earliest different granting time t for y;
                        / if there is none, define this time as greater than all "real times"

  forall w ∈ Grantees do revoke*(t,y,w)
                        / y recursively revokes all invalidated grantings, namely
                        / those before the earliest different granting time t for y
fi.
```

Recursive revocation: main procedure

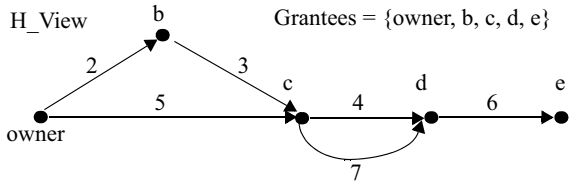
```
procedure revoke_owner,o,m(time,revoker,grantee);
{  precondition: owner ∈ Grantees;
   import:       Grantees, H_View, t_max}
                                  / at time the revoker invalidates his grantings
                                  / of privilege [o,m] concerning object o of owner
                                  / to grantee
if                                      / access decision:
   t_max < time                         / issue times are monotone
then                                    / updating of grant graph:
   revoke*(time,revoker,grantee);       / first call of recursive auxiliary procedure
   delete isolated elements from Grantees except owner
fi.

procedure revoke*(t,x,y);          / recursive auxiliary procedure for revoke_owner,o,m
{  precondition: owner ∈ Grantees;
   import:       Grantees, H_View, t_max}
```

Recursive revocation: recursive auxiliary procedure

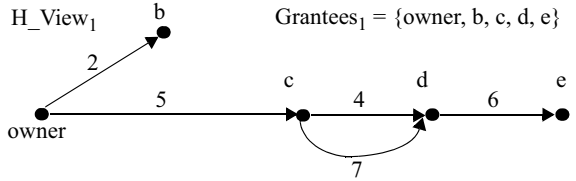
```
procedure revoke*(t,x,y);        / recursive auxiliary procedure for revoke_{owner,o,m}
{ precondition: owner ∈ Grantees;
  import:       Grantees, H_View, t_max}
if                                / access decision:
  EXISTS t_early: t_early < t  AND  (t_early,x,y) ∈ H_View
                                  / x has granted privilege to y before time t
then                              / updating of grant graph:
  H_View := H_View \ {(t_early,x,y) | t_early < t };
                                  / delete grantings from x to y before time t

  VALID := { t_other | EXISTS x_other: (t_other,x_other,y) ∈ H_View };

  if VALID ≠ ∅ then t := minimum(VALID) else t := ∞ fi;
                        / compute earliest different granting time t for y;
                        / if there is none, define this time as greater than all "real times"

  forall w ∈ Grantees do revoke*(t,y,w)
                        / y recursively revokes all invalidated grantings, namely
                        / those before the earliest different granting time t for y
fi.
```

Procedure call `revoke(8,b,c)`: the run

Initial instance of grant graph *H_View* when the procedure $\texttt{revoke}_{\texttt{owner,o,m}}\texttt{(8,b,c)}$ is called.

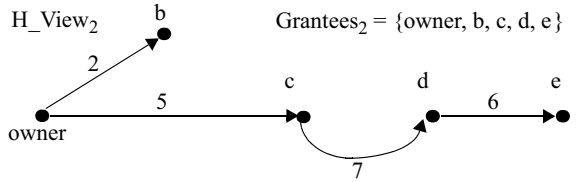


First call of auxiliary procedure, `revoke*(8,b,c)`, delivers H_View_1 with $VALID_1 = \{5\}$ and $t_1 = 5$.



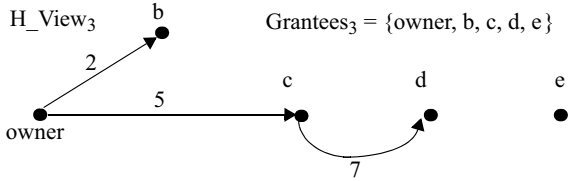
Recursive calls for $w \neq d$ do not change the grant graph.

Recursive call for $w = d$, `revoke*(5,c,d)`, delivers H_View_2 with $VALID_2 = \{7\}$ and $t_2 = 7$.



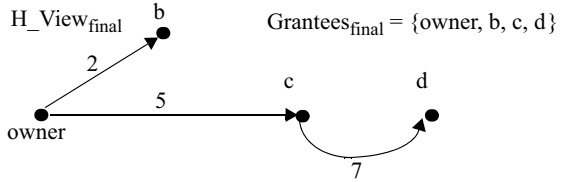
Recursive calls for $w \neq e$ do not change the grant graph.

Recursive call for $w = e$, `revoke*(7,d,e)`, delivers H_View_3 with $VALID_3 = \emptyset$ and $t_3 = \infty$.



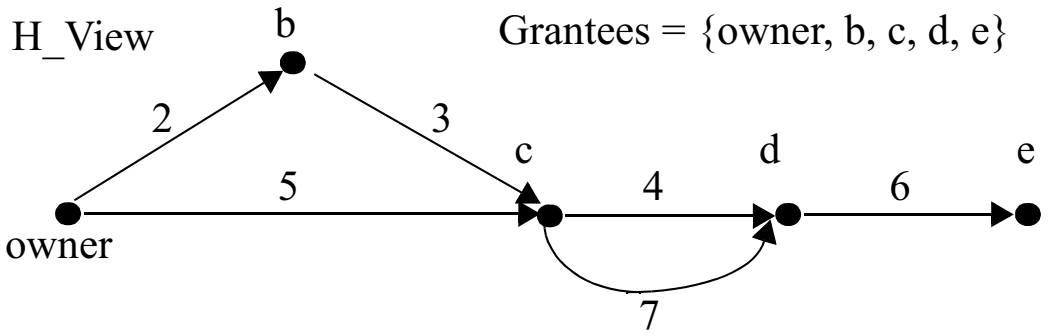
All further recursive calls do not change the grant graph.

Finally, all isolated nodes are removed.

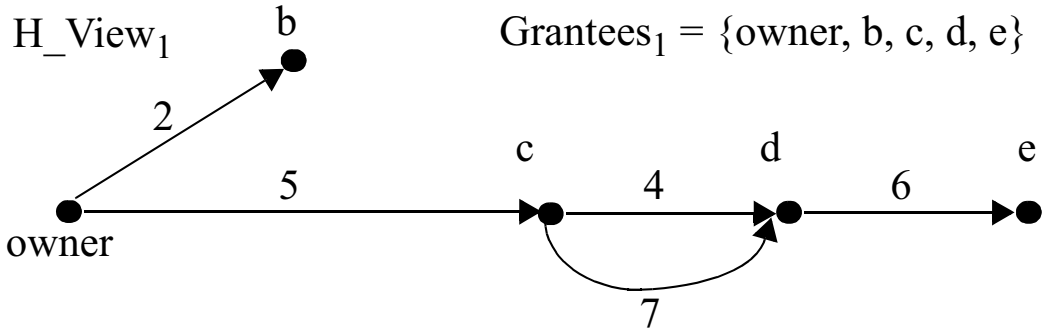


Procedure call `revoke(8,b,c)`: call of auxiliary procedure

Initial instance of grant graph *H_View* when the procedure
$\texttt{revoke}_{\texttt{owner,o,m}}\texttt{(8,b,c)}$
is called.



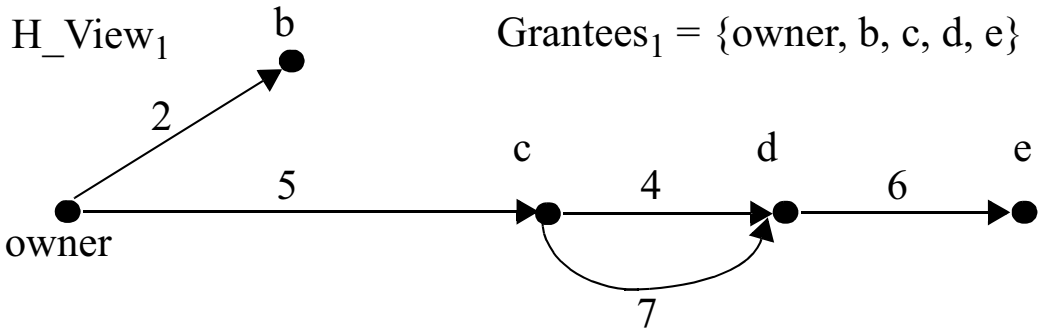
First call of auxiliary procedure,
`revoke*(8,b,c)`,
delivers H_View_1 with
$VALID_1 = \{5\}$ and $t_1 = 5$.



Recursive calls for $w \neq d$
do not change the grant graph.

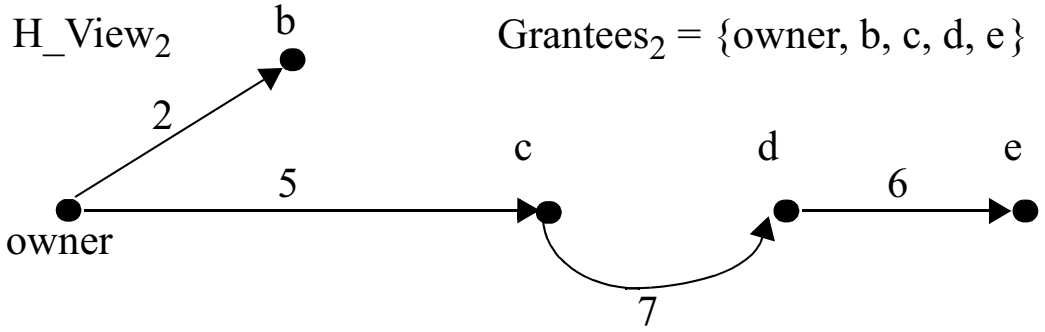
Procedure call `revoke(8,b,c)`: a recursive call

First call of auxiliary procedure,
`revoke*(8,b,c)`,
delivers H_View_1 with
$VALID_1 = \{5\}$ and $t_1 = 5$.



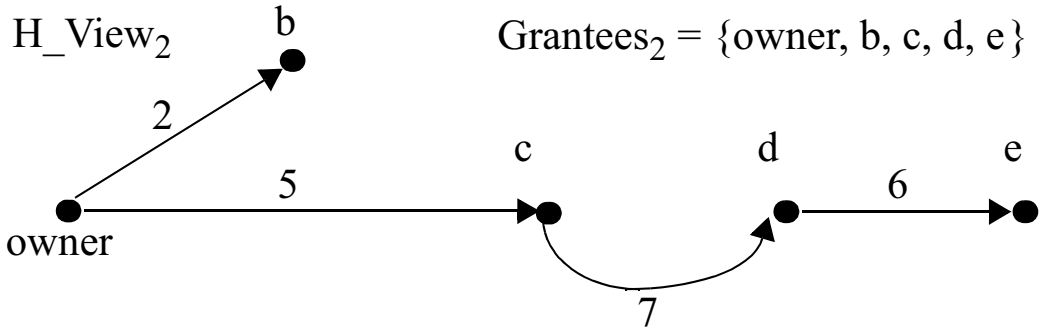
Recursive calls for $w \neq d$
do not change the grant graph.

Recursive call for $w = d$,
`revoke*(5,c,d)`,
delivers H_View_2 with
$VALID_2 = \{7\}$ and $t_2 = 7$.



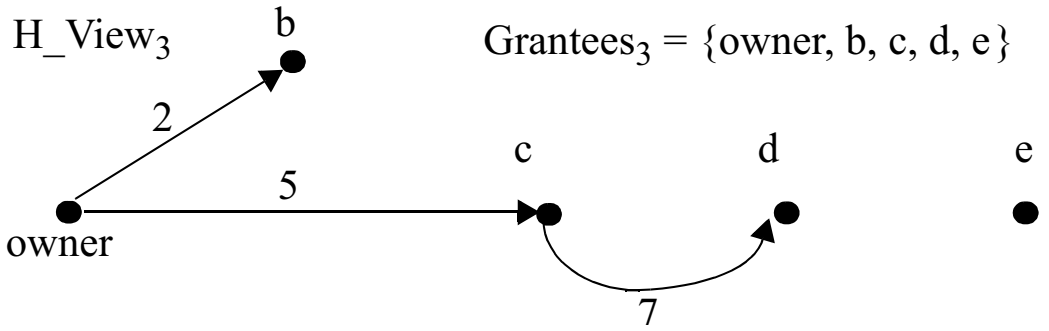
Procedure call `revoke(8,b,c)`: a further recursive call

Recursive call for $w = d$,
`revoke*(5,c,d)`,
delivers H_View_2 with
$VALID_2 = \{7\}$ and $t_2 = 7$.



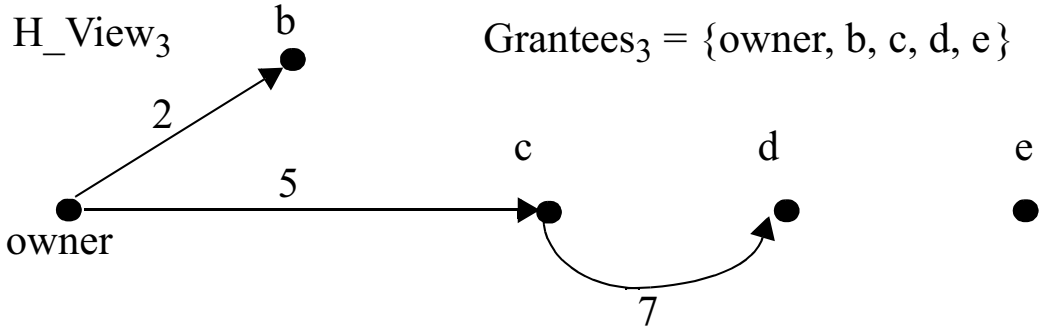
Recursive calls for $w \neq e$
do not change the grant graph.

Recursive call for $w = e$,
`revoke*(7,d,e)`,
delivers H_View_3 with
$VALID_3 = \varnothing$ and $t_3 = \infty$.



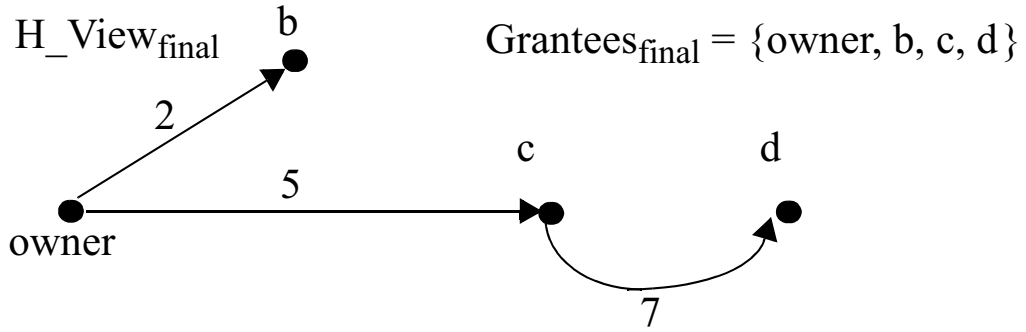
Procedure call `revoke(8,b,c)`: removing isolated nodes

Recursive call for $w = e$, `revoke*(7,d,e)`, delivers H_View_3 with $VALID_3 = \emptyset$ and $t_3 = \infty$.



All further recursive calls do not change the grant graph.

Finally, all isolated nodes are removed.



Dynamic and state-dependent permissions

- basic concepts ensure *availability* in two steps:
  - some administrator grants the needed permissions, *permanently* represented in a *knowledge base* on *permissions and prohibitions*
  - a grantee can *repeatedly* employ his permissions whenever he *himself* wants to do so

- the availability of a resource can be explicitly terminated by *revoking* the pertinent privileges

- one can *further restrict* the availability of a resource by employing the *knowledge base* on the *usage history*

- an administrator can express a policy that
  - "statically" permits a requestor to access a resource
  - but additionally requires the validity of some "dynamic conditions" for any actual request

Control automatons

- *static* layer:
  the “principally permitted” options for using a computing system
  are declared in some suitable way

- *dynamic* layer:
  control automatons specify *security contexts* by their states;

  a security context represents the collection of
  those permissions (and prohibitions)
  that are actually exploitable by an individual or a community
  at a specific point in time

Some purposes of a security context

- *selecting* a narrow subset of the "principally permitted" options or *selecting* one alternative out of several mutually exclusive possibilities
- monotonically *decreasing* the usability of "principally permitted" options; *resetting* previously decreased usability
- temporarily *amplifying* the "principally permitted" options for special tasks
- partially *implementing* "principally permitted" options by means of runtime concepts of operating systems and programming languages
- *enhancing* the runtime complexity of access decisions with respect to the "principally permitted" options, owing to appropriate precomputations
- *sequencing* the actual employment of "principally permitted" options

State transitions of control automatons/switches of security contexts

- explicitly,
  owing to a *control operation*

- implicitly (as a side effect),
  owing to a *functional operation*

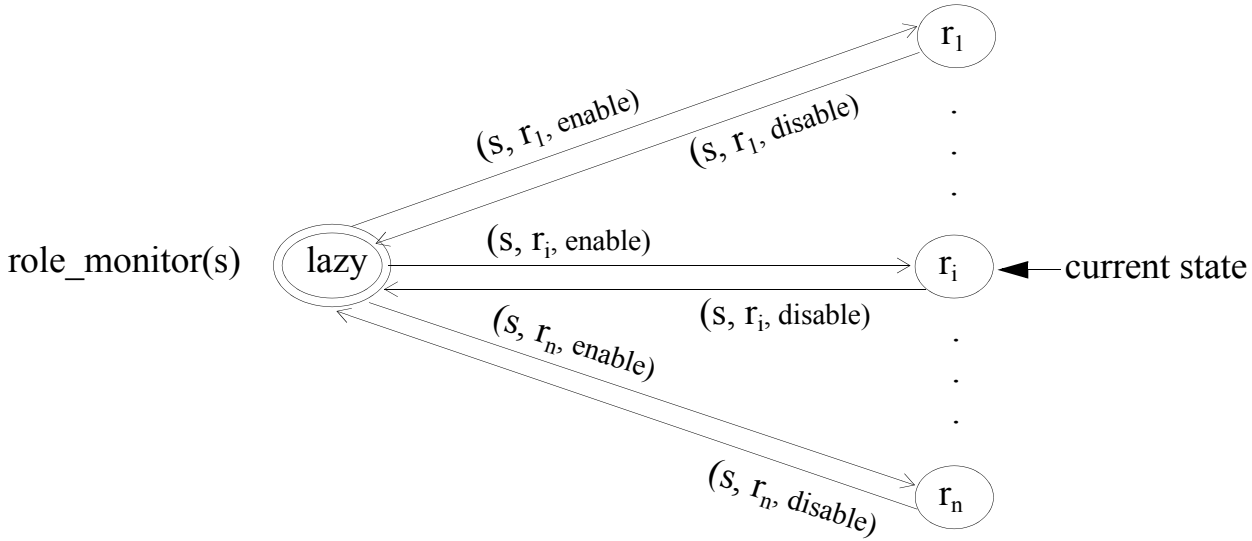
- “spontaneously”,
  owing to an *error condition* or a detected *security violation*

Role enabling and disabling: an example

Granted	Role	Object	Mode
	.	.	.
	.	.	.
	r_i	$o_{i,1}$	$m_{i,1}$
	…	…	…
	r_i	$o_{i,k}$	$m_{i,k}$
	.	.	.
	.	.	.

Role_Assignment	Subject	Role
	.	.
	.	.
	s	r_1
	…	…
	s	r_i
	…	…
	s	r_n
	.	.

Role_Enabling	Subject	Role
	.	.
	.	.
	s	r_i
	.	.



Information flow monitoring

- initially,
  an individual subject is *statically* granted a permission
  to access some information sources “in principle”

- while the individual is enjoying his privileges,
  a monitoring automaton aims at *dynamically* preventing the individual
  from gathering “more information than intended”

Chinese Walls

- in principle:
  a participant can advise several companies, and thus
  is permitted to access objects owned by different companies

- however:
  - if two companies are *competing*, then the consultant should not simultaneously obtain *information* from both companies
  - any *information flow* from one company via a consultant to the other competing company must be strictly prevented

  thus dynamically:
  once the consultant has read an object owned by one company:
  - he is dynamically prohibited to *access* any object of a competing company
  - to avoid transitive information flows,
    writing is restricted to objects of just that company

Experimental operating system HYDRA

- system maintains a *runtime stack* for procedure calls
- a procedure call triggers the dynamic creation of a *local name space* object
- this object contains or references all runtime data needed,
  including the *privileges* needed to access other objects
- usually, these privileges are dynamically *granted* in two ways:
  - the (dynamic) *local name space* object of the *calling procedure*
    can copy any selection of its own privileges and
    pass the copies as actual parameters
  - the (static) *program* object of the *called procedure*
    transmits its own privileges
- *in general*, the permissions of a specific execution of a procedure
  are strictly bounded by the permissions held by the two source objects
- *right amplification* might supply a *local name space* object
  with a privilege that is neither held by the calling dynamic object
  nor possessed by the called static object

Java protection framework: local and remote code

- the framework includes rules for *deciding on access requests* issued by the execution of either local or remote (program) code
- *local* code might be assumed to be "trustworthy" and thus qualify to discretionarily receive privileges to access local resources
- *remote* code is seen to be potentially "suspicious" and thus treated with special care;
  as an extreme option, executed in a *sandbox*

Java protection framework: enabling flexible cooperation

- *subjects* are formed by a set of Java classes characterized by:
  - origin
  - acceptance of digital signatures
  - certificates
  - ...

- such a subject is assigned a *protection domain*,
  which is granted concrete *privileges*

- later on,
  all runtime instance objects of the pertinent classes
  inherit these privileges from the protection domain

- however,
  the usage of privileges is further dynamically *restricted* by stack inspection

Java stack inspection

- a *runtime stack* for each thread:
  to keep track of a chain of pending *method invocations*
- fundamental policy:
  a nested execution of a method
  may not be more powerful that any of its predecessors in the chain
- mechanism of *stack inspection*:
  if a method execution *requests* to access a resource,
  then the mechanism inspects both
  - the privileges of the current protection domain
    assigned to the *relevant subject*
  - the privileges of *all* protection domains
    assigned to the *predecessors*
- the request is *permitted* only if
  all items in the chain possess appropriate privileges
- there are further refinements, optimizations and even exceptions

Workflow control

- an administrator statically declares a *workflow schema*
- suitable participants dynamically execute one or more *workflow instances*
- while an instance is progressing, at any point in time:
  - any participant scheduled to perform the next step should effectively receive the pertinent privileges
  - all other participants deemed to be waiting for a call should be temporarily prevented from acting
  - after the completion of a step, one or more succeeding steps must be enabled
- the workflow schema and the current state of an instance jointly specify a *security context*
- such a security context can be described by a *control automaton*, suitably formed as a *finite automaton* or a *Petri net*

Analysis of control states: basic problem

- may a subject *s ever* acquire a privilege [*o*,*m*]?
- can a subject *s never* acquire a privilege [*o*,*m*]?
- given a current *control state* (permissions/prohibitions)
  does there exist a *sequence* of permitted *control operations*
  such that, *afterwards*, a request
  - from subject *s*
  - to perform an operation of mode *m*
  - on object *o*

  is *permitted*?
- in the positive case,
  - which participants,
  - using which control operations,

  can achieve such acquisition?

Undecidability of the analysis problem of control states/operations

- elaborate a formal model for control state
- elaborate a formal model of control operation
- show that the halting problem is reducible to the analysis problem

A model of control states

- **time-independent declarations:**
  - I infinite set of potential system *items*; may act both as *subject* and as *object*
  - $Actor \subseteq I$ infinite set of *actors*; may request control operations
  - $Mode$ finite set of *modes*
    $FM \subseteq Mode$ set of *functional modes*
    $KM \subseteq Mode$ set of *control* and *relational modes*
  - $Priv := \{ [x,m] \mid x \in I \text{ and } m \in Mode \}$ set of *privileges*

- **time-dependent *control state* (L_t,$Granted_t$,$Cond_t$):**
  - t actual abstract *time*
  - $L_t \subseteq I$ finite set of system items that are *alive* at time t
  - $Granted_t \subseteq L_t \times L_t \times Mode$ representing the actual *permissions* or other *relationships* at time t
  - $Cond_t$ possibly parameterized *further condition*

A model of control operations

- a *control operation* is a call of a parameterized *control procedure*:

  `procedure` `control_schema_ident`
  (*formal_mode_list*; *formal_item_list*);

  { **`import:`** `Granted, L, Cond` }

  `if` subrelation *Required* is contained in relation `Granted`
  `AND`
  condition `Cond` is satisfied

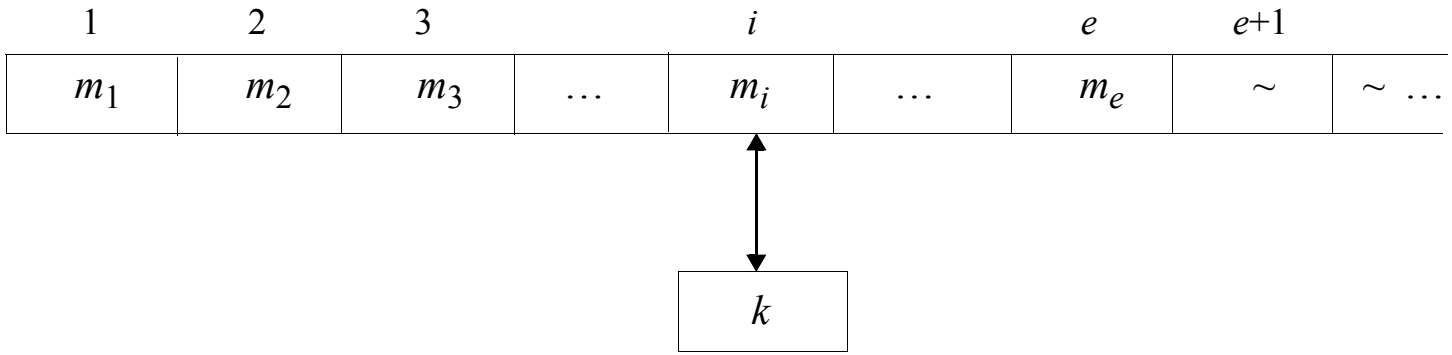
  `then` modify `Granted` and, if required, also `L;`
  if required, adapt `Cond`

  `fi`.

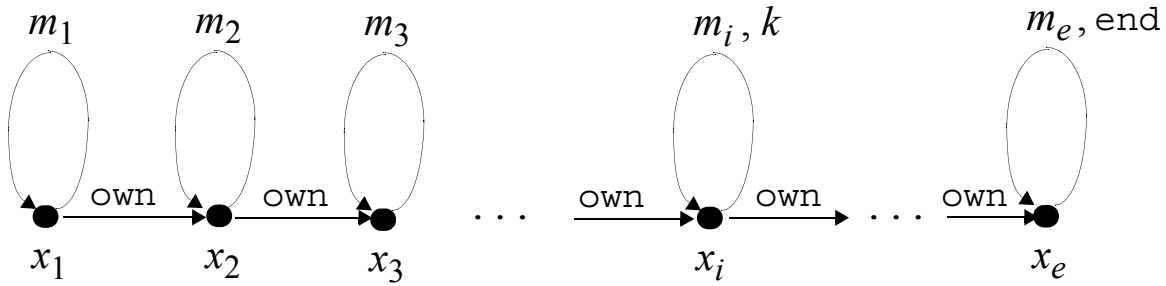
- a modification of *Granted* or *L* consists of a sequence of *elementary actions*:
  - *insert*(s,o,m): $Granted := Granted \cup \{ (s,o,m) \}$
  - *delete*(s,o,m): $Granted := Granted \setminus \{ (s,o,m) \}$
  - *create*(y): $L := L \cup \{ y \}$ with $y \notin L$
  - *destroy*(y): $L := L \setminus \{ y \}$

Reduction: simulation of TM configurations by of control states

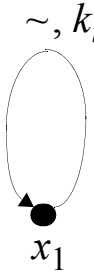
a configuration of *TM*:



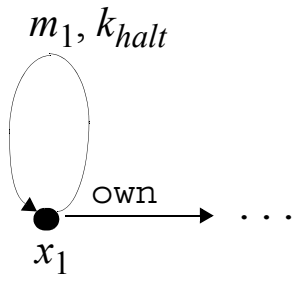
a simulating control state:



the initial control state S_1:



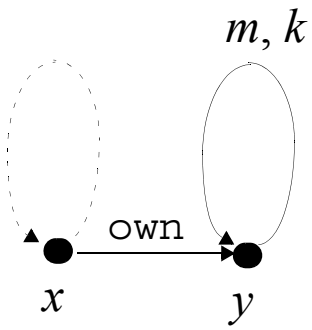
a halting control state:



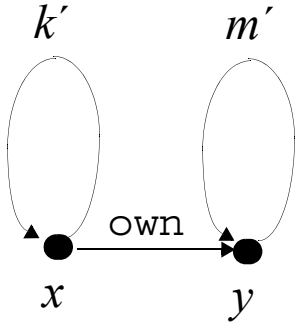
Reduction: simulation of TM moves by of control operations

Case $\delta(k,m) = (k',m',left)$:

$\texttt{sim}_{k,m}(x,y)$: **if**

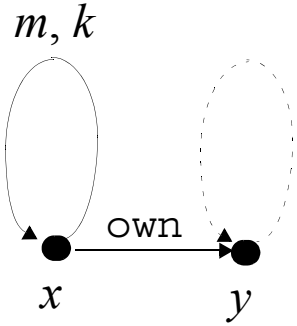


then

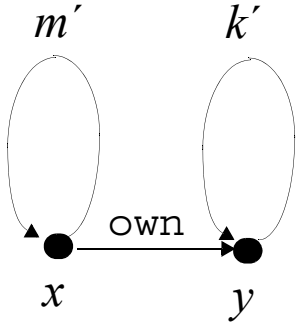


Case $\delta(k,m) = (k',m',right)$:

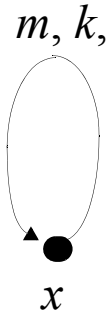
$\texttt{sim}_{k,m}(x,y)$: **if**



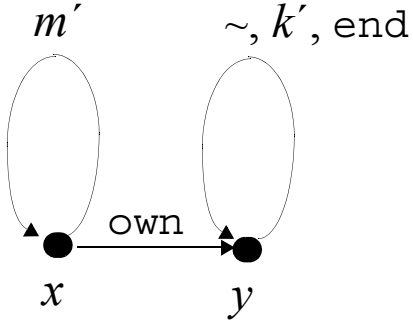
then



$\texttt{sim\_tape\_ext}_{k,m}(x,y)$: **if**



then



Some crucial insight

- though control operations appear to be very simple,
  they can be expressive enough
  to simulate the *local behavior* of a Turing machine

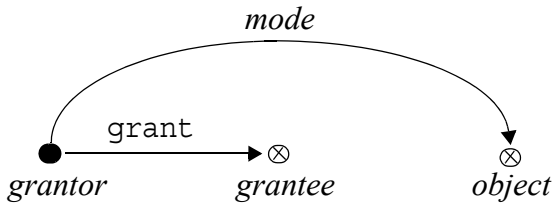
- together with the option to *create* new items,
  the computational power of *universal* programming languages is reached

- we can try to achieve decidability,
  by suitably restricting the expressiveness, for example:
  - just prevent the TM simulation
  - avoid the interaction of granting and taking together with creating
  - employ suitable typing

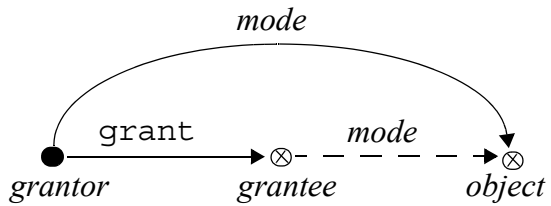
Generic take–grant operations and create

`grant(mode;grantor,grantee,object):`

if

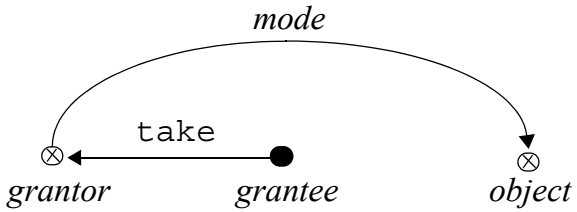


then

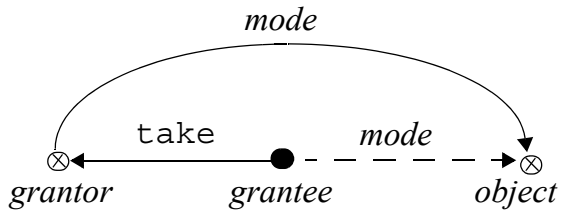


`take(mode;grantor,grantee,object):`

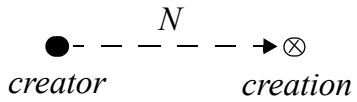
if



then



`create`$_N$`(creator,creation):`



Analysis problem for generic take, grant and create: theorem

consider *Control*: generic take, grant and create as *control operations*
S_1: a current *control state* (permissions/prohibitions)
s: a *subject*
o: an *object*
m: a *mode*

claim:
there exists a *sequence* of permitted *control operations* such that a request from subject s to perform an operation of mode m on object o is *permitted*
iff
there exists an item $p \in L_1$ such that

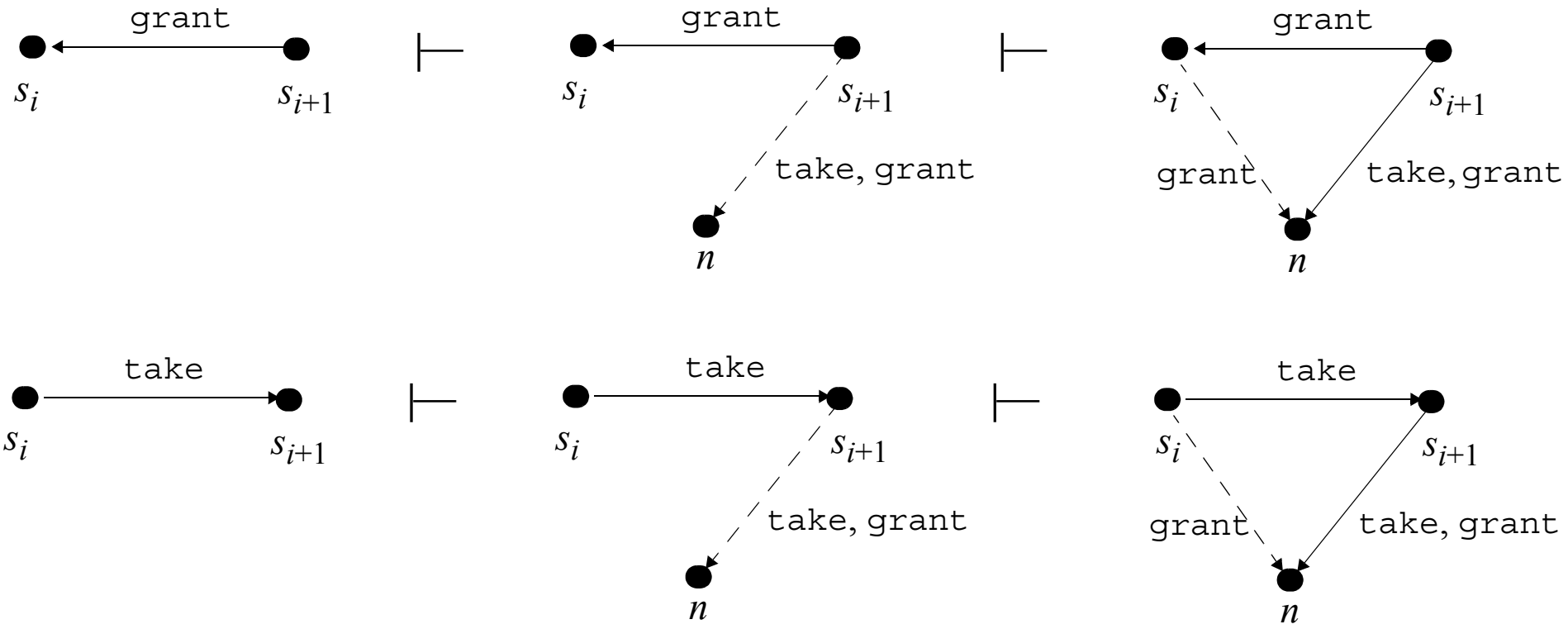
(i) $(p,o,m) \in Granted_1$

p already holds the examined privilege $[o,m]$ in the original control state S_1

(ii) p and s are *take/grant-connected* in the original control state S_1

in the corresponding access control graph there is a path from p to s,
such that, *ignoring directions*,
each edge on the path is labeled with `take` or `grant`

Reversing directions of grant and take privileges



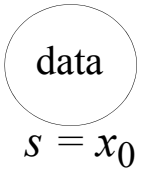
Privileges and information flow: extended analysis problem

- may a subject *s* **ever** be enabled
  to *learn the information* contained in some object *o*?

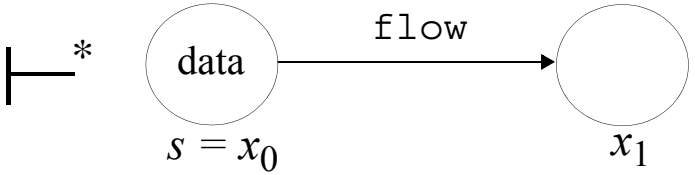
- will a subject *s* **never** be enabled
  to *learn the information* contained in some object *o*?

Interactions of control operations and functional operations: example

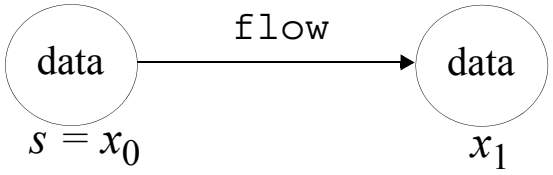
initial state with source s containing data:



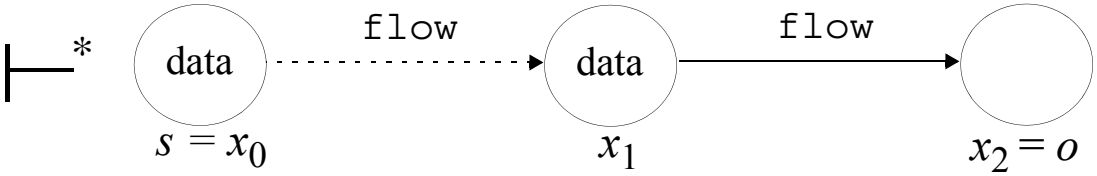
control operations establish transmission edge to x_1 for data:



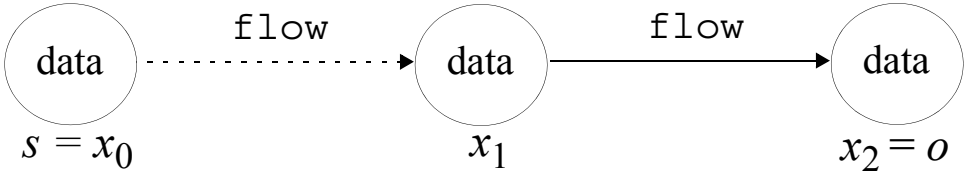
functional operation transmits data to x_1:



control operations establish transmission edge to x_2 for data:

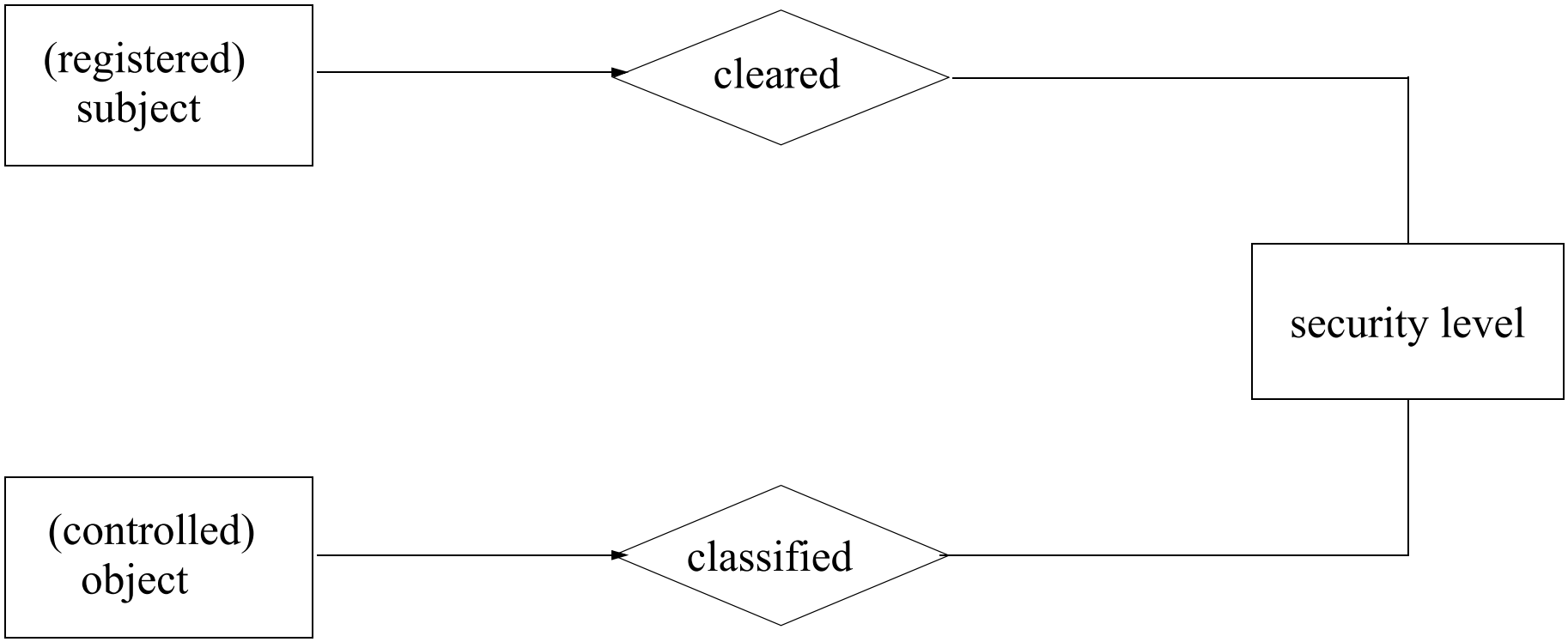


functional operation transmits data to x_2:



7 Mandatory Access Control and Security Levels

ER model



- a security level assigned to a *subject* as a *clearance*
  roughly expresses a degree of its *trustworthiness* (concerning confidentiality)
- A security level assigned to an *object* as a *classification*
  roughly expresses a degree of its *sensitivity* (concerning confidentiality)

Partial orders for relative trustworthiness and relative sensitivity

- use a partial order $\leq_{SL}$ on the set of security levels SL

- an ordering $l_1 \leq_{SL} l_2$ expresses:
  - *relative trustworthiness* ("less trustworthy than") for subjects
  - *relative sensitivity* ("less sensitive than") for objects

Mandatory security policy

- *information flows* must respect the *orderings between security levels*:
  - a request is *permitted*
    only if
    the trustworthiness of the *requestor* (postulated properties of subject)
    suffices for (are expected to cover)
    the sensitivity of the *target* (protection requirements for object)
  - information *may flow* from an item with level l_1 to an item with level l_2
    only if
    $l_1 \leq_{SL} l_2$

- we have to know exactly the direction of the potential information flows:
  - *reading*: from the accessed object to the requesting subject
  - *writing*: from the requesting subject to the accessed object
  - *any*: function `mode` has to correctly assign mode `read` or `write`

Access decisions to enforcing confidentiality

```
function decide(subject, object, operation): Boolean;
if    mode(operation) = read
then  return  classification(object)  ≤_SL  clearance(subject)
fi;

if    mode(operation) = write
then  return  clearance(subject)  ≤_SL  classification(object)
fi.
```

- the function `decide` is supported by a conceptual *knowledge base* (on permissions and prohibitions) expressing:
  - `cleared` relationships of the form (*subject*, *clearance*)
  - `classified` relationships of the form (*object*, *classification*)

- the achievements are often briefly referred to as the *read-down/write-up rule* for *upwards information flow*

Mandatory control of information flow: debates

- achievements rely on strong suppositions concerning:
  - a common understanding of security levels by all administrators
  - correct assignments of operational modes to all operations

- achievements might be too restrictive:
  - allow only *unidirectional* information flows
  - thus prevent full back and forth communications

- achievements might nevertheless be too weak:
  - potential *inferences* about the results of permitted operations are not captured in general

Dynamic mandatory access control

- so far:
  an object is *statically* assigned a fixed security level as its *sensitivity*

- now refined:
  - treat an object as a *container*
  - capture the *dynamic* evolution of the sensitivity of the *content* during a sequence of operations
  - increase the classification for the object like a *high-water mark*, according to the most sensitive information that has ever flowed in

- convenient postulate:
  the partial order $\leq_{SL}$ on the domain SL of security levels
  forms a finite *lattice*

Security levels as a finite lattice: underlying assumptions

- domain *SL* of security levels:
  to represent some *aspects* regarding information

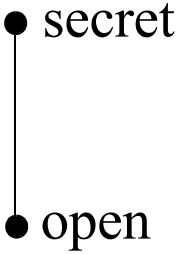
- $\leq_{SL}$ is a *partial order* on the domain *SL*:
  to treat *transitive* information flows

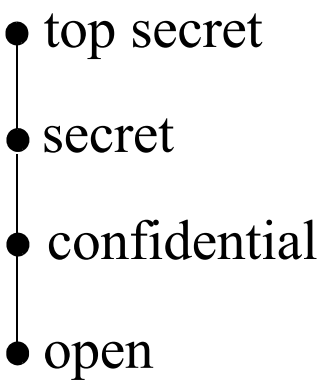
- $\leq_{SL}$ allows *infimums* (*greatest lower bounds*) $inf_{SL}(l_1, l_2)$:
  to capture the *common part* of the aspects represented by l_1 and l_2

- $\leq_{SL}$ allows *supremums* (*least upper bounds*) $sup_{SL}(l_1, l_2)$:
  to capture the *accumulation* of the aspects represented by l_1 and l_2

Example of security levels: linear orders

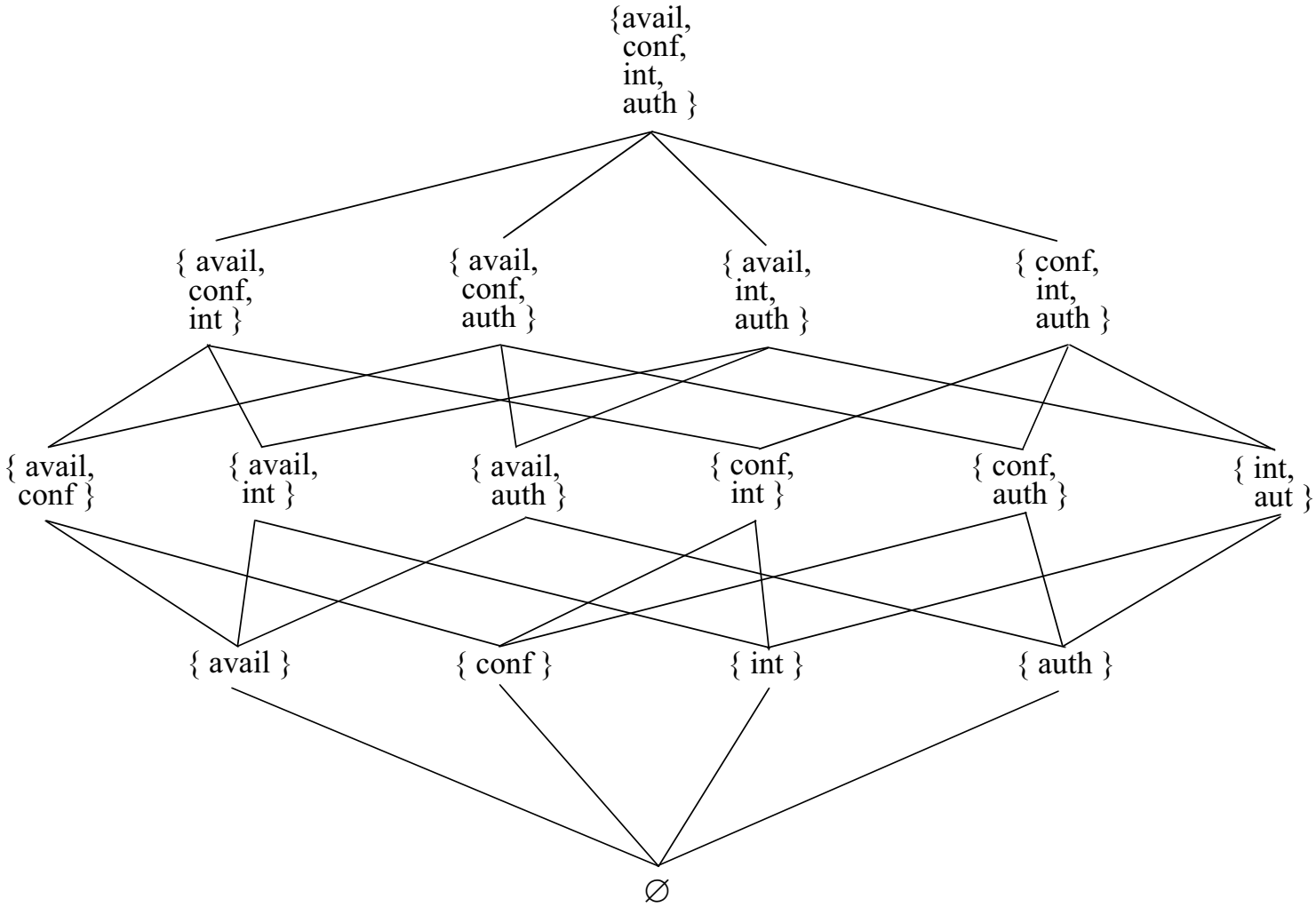
to *characterize* information under the interest in *confidentiality*:





Example of security levels: power set lattices

to describe information by *subject matter*,
using a set of keywords *KW*, e.g.,
$KW = \{\text{avail(ability)}, \text{conf(identiality)}, \text{int(egrity)}, \text{auth(enticity)}\}$



Approximation of dependance by levels: container objects

- *in principle*, keep track of
  which items the current content of the container object *co* actually depends on
- as an efficient *approximation*,
  maintain a *dynamic* (high-water mark) security level $sl_{co}(t)$,
  where t denotes the time parameter:
  - if some data d is *read* from the container at some point in time t,
    then d inherits the container's current security level $sl_{co}(t)$
    as its *dynamic classification*
  - if some data d, supposed to carry some security level sl_d,
    is *written* to the container *co* at some point in time t,
    then the container's security level is updated to the
    least upper bound of the container's previous level and the data's level:
    $$sl_{co}(t) := sup_{SL}\{ sl_{co}(t-1), sl_d \}$$
- crucial issue: specify convincingly
  how the data d to be written obtains its dynamic classification sl_d

Dynamic classification of data: expressions

- partial answer by the rule for reading:
  if data *d* is *just* read and then written *without* any further interactions,
  then take the *inherited* dynamic classification

- extended answer:
  if data *d* results from performing some operation *op* on arguments *a* and *b*,
  each of which has *only* been read *just before* the operation,
  then take the *supremum* of the *inherited* dynamic classifications

- generalized for arbitrary expressions:
  take the *supremum* of all the *inherited* dynamic classifications involved

Dynamic classification of data: active subjects

for a subject *su* that actively participates in forming the data to be written:

- if the subject *only* persistently stores previously read data over time,
  then treat the subject like a container,
  in particular,
  su obtains a dynamic security level $sl_{su}(t)$

- if a subject dynamically "generates" new data,
  then let the data inherit
  either the subject's static clearance
  or some lower label

Combining static and dynamic features: outline of a formal model

- each subject *su* obtains a *static* security level $clearance(su)$
  expressing its *trustworthiness*
- each container object *co* obtains a *static* security level $classification(co)$
  expressing its *initial sensitivity*
- all operations are monitored by a *control automaton*
  whose internal *states* are composed of
  the *dynamic* security levels $sl_{co}(t)$ and $sl_{su}(t)$
  for each container object *co* and each subject *su*, respectively
- for $t = 0$, the dynamic security levels are *initialized* by:
  - $sl_{co}(0) := classification(co)$, for each container object *co*
  - $sl_{su}(0) \leq_{SL} clearance(su)$, for each subject *su*
- for $t > 0$, an *access request* is decided
  according to the current state resulting from time $t-1$,
  and a *state transition* reflecting the decision is performed

Static and dynamic features: access decisions and state transitions

- subject *su* requests to *read* from a container object *co*:
  - access permitted iff $sl_{co}(t-1) \leq_{SL} clearance(su)$
  - in the case of a permission,
    $sl_{su}(t) := sup_{SL}\{ sl_{su}(t-1), sl_{co}(t-1) \}$

- subject *su* requests to *write* to a container object *co*:
  - access (always) permitted
  - $sl_{co}(t) := sup_{SL}\{ sl_{su}(t-1), sl_{co}(t-1), l \}$, where $l \leq_{SL} clearance(su)$

- satisfied *security invariant* (*for confidentiality*):
  $sl_{su}(t) \leq_{SL} clearance(su)$, for each subject *su*, at any point in time *t*

  a subject sees only data
  which results from operating on arguments
  whose sensitivities have been classified as lower than or equal
  to the clearance of that subject

Models attributed to Bell and LaPadula

- exploring the fundamental concepts of:
  - a “secure state”: satisfying a security invariant
  - a “secure action”: preserving the invariant

- enforcing a *(*star*)-*security property*:
  reading an object and subsequently *writing* to another object
  is permitted only if
  the label of the former object is *lower than or equal*
  to the label of the latter object

- establishing a *Basic Security Theorem*:
  starting a system in a secure state and
  employing only secure actions
  guarantee that the system is “secure” (always satisfies the invariant)

Downgrading

- pure mandatory approach:

  only unidirectional information flows,
  "from low to high"

- many applications require exceptions, e.g.:
  - critical information to be kept top secret for some time might age over the years and thus becomes less critical
  - a subject acts in highly critical missions for some time but subsequently is given a less critical task

- some *downgrading* of an object/subject is due, e.g.:
  - an original classification/clearance "top secret" is substituted by "confidential"

Sanitation

- downgrading may possibly be preceded by *sanitation*, i.e.:
  - data is inspected for parts that are still critical
  - such data is then individually removed or suitably replaced by harmless variants

Trusted subjects and violation of the basic security property

- downgrading and sanitation are performed
  by special subjects that are considered as *trusted*, i.e.,
  being *exempted* from obeying the pure rules of mandatory access control

- downgrading may violate the basic security property:
  - only suitably relaxed formal security properties are still valid
  - in the extreme case,
    there are no formal guarantees of confidentiality anymore

Confidentiality and integrity

- *confidentiality* security levels are designed to preserve *confidentiality*:
  - data can be distributed to the equally or higher-labeled items
  - such data can be *written* into these items
  - all these items can be *modified*, and
    thus are subject to concerns about their *integrity*
  - in general, integrity will *not* be preserved

- close relationship between
  - "(no) information flow"
  - "(non)interference"

- these two notions are dual:
  there might be an information *flow* from some item *a* to another item *b*
  if and only if
  item *a* might *interfere* with item *b*
  (i.e., *a* might have an impact on the behavior of *b*)

A dual approach to enforcing integrity

- fully *dualize* the mandatory approaches for preserving confidentiality, including the procedure for access decision

- a *read-up/write-down rule* for *downwards interference* is employed:
  - *reading* is allowed *upwards*, i.e., from equally or higher-labeled items
  - *writing* is allowed *downwards*, i.e., to equally or lower-labeled items

Enforcing both confidentiality and integrity

- *jointly* apply
  the “read-down/write-up rule” and
  the “read-up/write-down rule”

- then only *accesses* within the set of equally labeled items are allowed, independently of whether the operational mode is `read` or `write`

- this strong restriction might impede the wanted application functionality

Additional integrity security levels

- use a separate set SL_{int} of *integrity security levels*:
  - an integrity security level assigned to a *subject* as a *clearance* expresses a degree of *trustworthiness* of the subject concerning preserving the *integrity* of items
  - an integrity security level assigned to an *object* as a *classification* expresses a degree of *sensitivity* of the object concerning the need for its *integrity* to be preserved

- trustworthiness and sensitivity concerning *confidentiality* and the corresponding concepts concerning *integrity* might differ essentially

- accordingly, both kinds of security levels should be applied in parallel, simultaneously following both permission rules:
  - *upwards information flow* regarding *confidentiality* security levels
  - *downwards interference* regarding *integrity* security levels

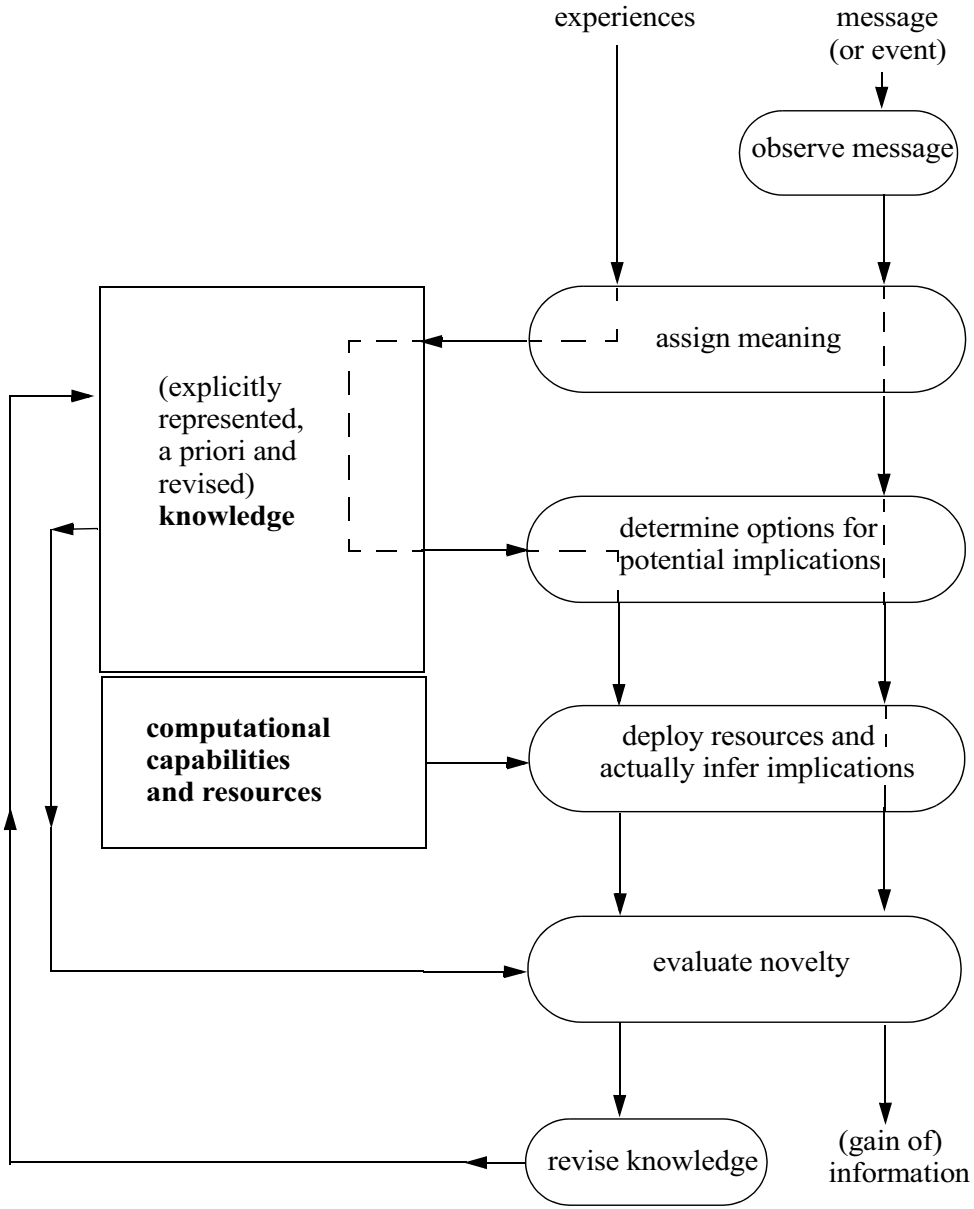
8 Inference Control

Information gain

- an *observer* of a *message* or other *event*
  achieves an *information gain*
  if he can convert his *a priori knowledge*
  into strictly increased *a posteriori knowledge*
  - when adding the *meaning* of the message or event and
  - making all possible *inferences*

- such a gain might
  remain merely *potential* or
  be *actually* realized,
  depending on
  - the fundamental computational capabilities and
  - the available computational resources of the observer

Information, knowledge, computational capabilities and resources



Information gain by an observer

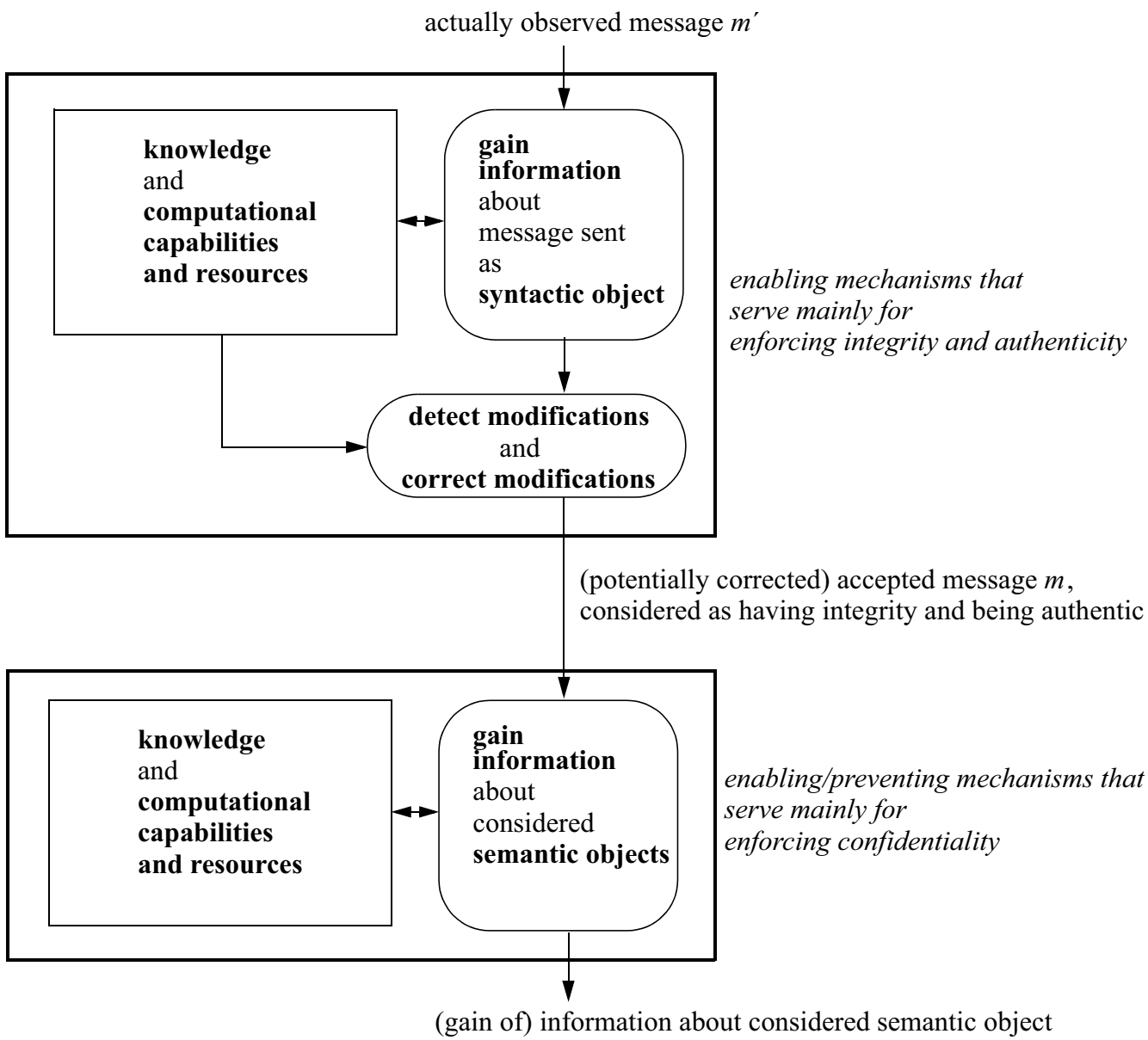
- selects a *framework for reasoning*
  as the pertinent *communicative context* or *universe of discourse*
- interprets an observation and assigns a *meaning* to the observation
- has some *a priori knowledge*
- employs a declarative notion of *implication*, using
  first-order logic, *probabilities*, *vagueness*, *uncertainty*, *preferences*, ... ,
  and thus reasons about the fictitious *implicational closure*
- *computationally infers*
  – *deploying* the *computational resources* available to him –
  selected or even all implications, and
  evaluates actual inferences concerning *novelty*
- treats the newly inferred implications as the *information gained*
- appropriately *revises* his previous *knowledge*,
  thereby getting *a posteriori knowledge*

Two extreme cases for the information gain

- the *a priori knowledge* and the *a posteriori knowledge* are *identical*:
  - the knowledge has remained invariant
  - the observer has learnt nothing novel
  - the set of possible worlds has not changed

- the *a posteriori knowledge* determines *exactly one* possible world:
  - the knowledge has become complete
  - the observer has learnt any property expressible in the selected framework

Enabling/preventing information flow concerning semantic objects



Simple mathematical model: inversion of functions/solving equations

- *framework for reasoning*:
  - function $f: D \to R$
  - domain $D = dom(f)$ containing at least two elements
  - range $R = range(f)$
  - an abstract assignment $x \to f(x)$ of function values to arguments
- *observation*:
  - message m, seen as a *syntactic object* in the form of a bit string
- *interpretation*:
  - m, seen as a *semantic object* $y \in R$, generated by the sender by applying the function f to some semantic object $x \in D$
  - m possibly contains information about some (hidden) *semantic object* $x \in D$ such that $f(x) = y$
- *gain of information*:
  - try to invert the function f for the given range value y
  - attempt to find the solutions of the equation $f(z) = y$ for the unknown variable z

A classification of functions: an (everywhere) injective function

for each $y \in R$ there exists a unique $z \in D$ such that $f(z) = y$:

- the observer can *potentially* gain *complete* information
- the *actual* gain depends on the observer's possibilities to actually compute the unique solution of the given equation

A classification of functions: a nowhere injective function

for each $y \in R$ there exist at least two different domain values $z_1 \in D$ and $z_2 \in D$ such that $f(z_1) = f(z_2) = y$:

- the observer cannot gain the sought information completely
  (he cannot *distinguish* the candidate domain values)

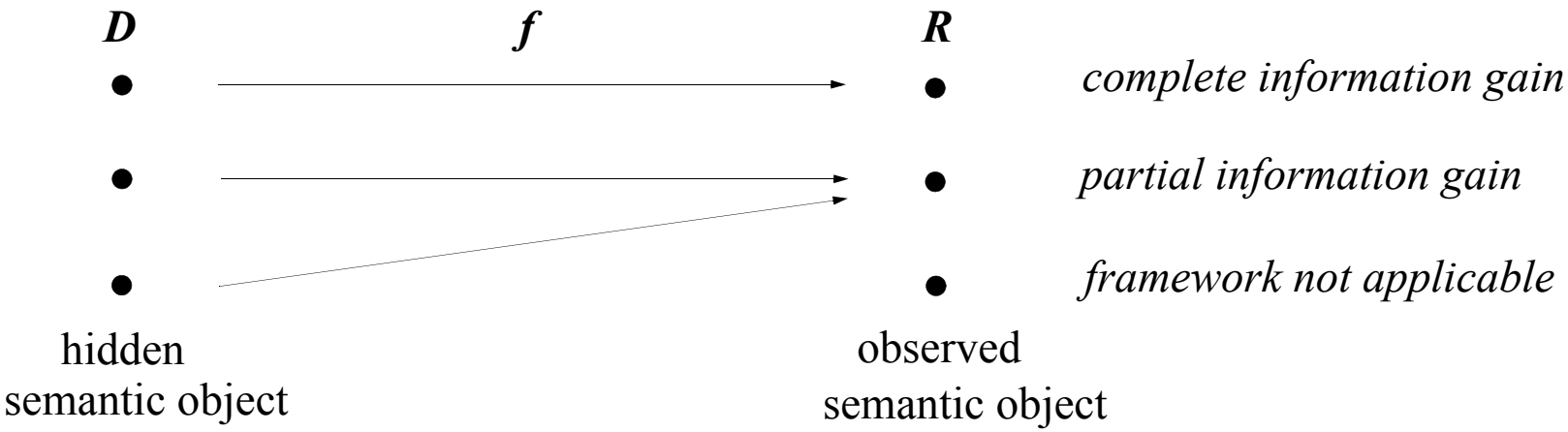
- the observer can possibly gain some *partial information*:
  - his *a posteriori* knowledge comprises
    $x \in \{z \mid f(z) = y\} \subseteq D$
  - if $\{ z \mid f(z) = y\} \neq D$,
    then the observer can *potentially* gain novel *partial* information;
    he can *exclude* the possibility that the hidden object x is
    an element of the difference set $D \setminus \{z \mid f(z) = y \}$
  - the *actual partial* gain depends
    on the observer's possibilities to actually compute the relevant items

A classification of functions: arbitrary functions

given the interpretation $y \in R$ of an observed message,
the observer determines the *pre-image* $\{ z \mid f(z) = y \}$:

- *complete* (potential) information gain:
  the pre-image contains exactly one element x, i.e.,
  card $\{ z \mid f(z) = y \} = 1$, and accordingly $\{ z \mid f(z) = y \} = \{x\}$
- *partial* (potential) information gain:
  the pre-image contains at least two (indistinguishable) elements
  but does not comprise the full domain D, i.e.,
  card $\{ z \mid f(z) = y \} > 1$ and $D \setminus \{ z \mid f(z) = y \} \neq \emptyset$
- *no* information gain:
  the pre-image is equal to the full domain D, i.e.,
  $\{ z \mid f(z) = y \} = D$
- *framework not applicable*:
  the pre-image is empty, i.e.,
  $\{ z \mid f(z) = y \} = \emptyset$

Exemplifying three cases regarding information gain



Observing the result of a group operation

- *group* $(G, \bullet, e)$
  - G — set of group elements
  - $\bullet : G \times G \to G$ — binary *group operation*
  - e — neutral element
  - $^{\text{inverse}} : G \to G$ — *inversion* with $x^{\text{inverse}} \bullet x = e$ for all $x \in G$

- group properties ensure the *solvability of equations*:
  every equation of the form $k \bullet x = y$, where two of the items are given,
  has a *unique solution* for the third item

- example: addition modulo 3

remaining argument x

parameter k	**0**	**1**	**2**
0	(0)	1	2
1	1	2	(0)
2	2	(0)	1

observation: $y = 0$
pre-image: $\{(0,0),(2,1),(1,2)\}$

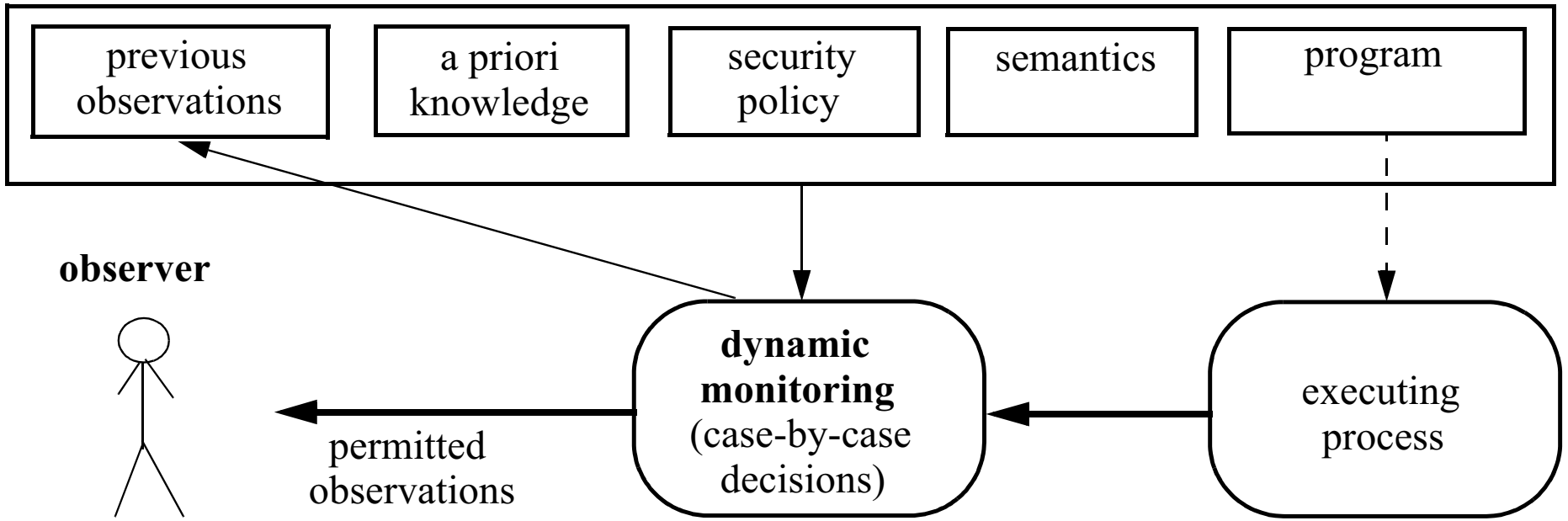
remaining argument
for known parameter $k=1$: 2

set of possible remaining arguments
for unknown parameter: $\{0,1,2\}$

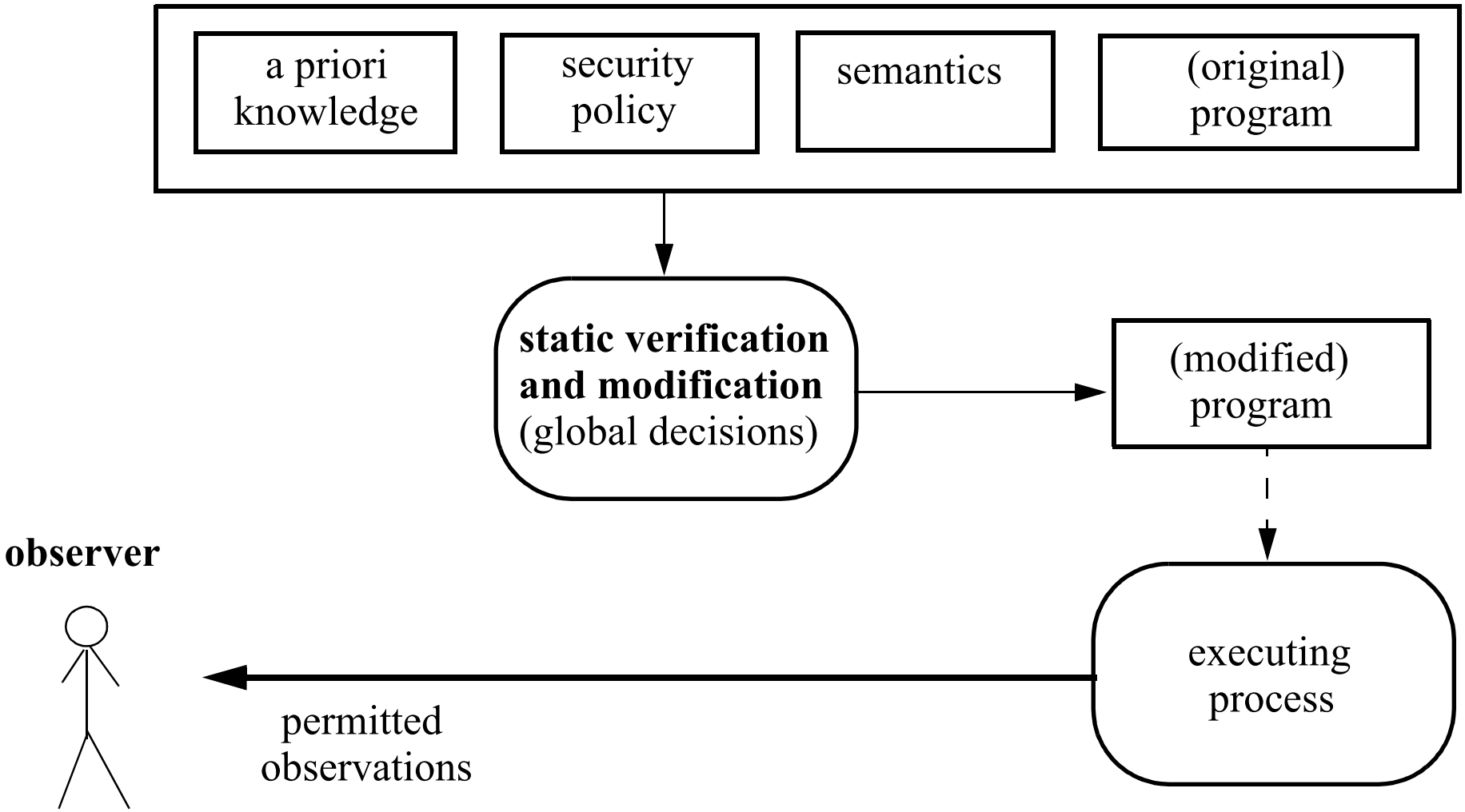
Information gain based on a parameter

- *observation*:
  $y \in G$ result of an application of the group operation
- a *partial information* gain about the arguments:
  $G \times G \setminus \{ (k,x) \mid k \bullet x = y \} \neq \emptyset$
- fix the first (or, similarly, the second) argument of the group operation to some parameter $k \in G$:
  family of functions $\bullet_k : G \rightarrow G$, where $\bullet_k(x) = k \bullet x$
- observer knows k:
  complete information about the remaining argument, since
  $\bullet_k(x) = y$ implies
  $k^{\text{inverse}} \bullet y = k^{\text{inverse}} \bullet (k \bullet x) = (k^{\text{inverse}} \bullet k) \bullet x = e \bullet x = x$
- observer does *not* know k:
  no information about the remaining argument, since
  $\{ x \mid \text{there exists } k \in G : k \bullet x = y \} = G$

Inference control by dynamic monitoring of a process



Inference control by static verification and modification of a program



Sequential programs: main constructs

- declaration of *typed identifiers* and
  generation of corresponding *program variables*,
  whose current values constitute a (*storage*) *state* of an execution
- *state transition*, caused by
  - *generating* a new program variable
  - *destroying* an existing one
  - *assigning a value* to a program variable
  - *passing an actual parameter* during a *procedure call*
- *control* of the execution sequence by
  - *sequential composition* of commands
  - *guarded commands* such as a *conditional* or a *repetition*
- evaluation of an *expression* occurring
  - in an assignment
  - as an actual parameter
  - as a guard
- computation of a *function* value needed during the evaluation of an expression,
  where the function is implicitly given by a fundamental type
  or has been explicitly declared

Sequential programs: an example

```
procedure flow(
  in    init, guard, x, y: integer;
  out   result: integer);

  local help: integer;

begin
  help := 2;
  help := help + init * init;

  if guard ≥ 0
  then help := help + x
  else help := help + y
  fi;
  result := help
end flow
```

Goals of analysis

- information gain
  about the actual parameter values of the
  input variables `init`, `guard`, `x` and `y`,

  passed *before* the execution of the body,

  when the value of the output variable `result`
  is observed *after* the execution

- information gain
  that may arise during subparts of the executions

Stepwise analysis: expressions and assignments

- **assignment `help := 2`**
  does *not* enable any information gain

- **expression `help+init*init`**
  - evaluates the subexpression `init*init` :
    observing the unidentified product value
    enables a nearly complete information gain
    about the actual parameter value of `init` passed

  - determines the final value of the full expression `help+init*init`:
    observing the sum value enables a
    complete information gain about the second argument, and, by *transitivity*,
    a nearly complete information gain about the value of `init` passed

- **assignment `help := help+init*init`**
  causes an *information flow*
  from the carrier `init`
  over the sum value
  to the carrier `help`

Stepwise analysis: positive branch of guarded command

- assignment `help := help+x` in the positive branch:
  - evaluates the expression `help+x`
  - delivers an unidentified sum value
  - assigns this sum value to the reused local variable `help`
- if the command is inspected separately,
  observing the sum value enables an information gain
  about neither the previous value of `help` nor the value of `x`
- an observer can achieve a partial information gain
  about the *pairs of these values*
- if the observer *knows* one of the argument values *a priori*,
  then the sum value uniquely determines the other argument value;
  by *transitivity*, the same reasoning applies for the value of `help`
- the complete command causes some *information flow*
  from `help` and `x` back to `help`

- the body is equivalent to the following command sequence:
  help := 2;
  help := help + init * init;
  help := help + x;
  result := help
- final step can be understood as a direct, explicit *data flow*
  from the local variable `help`
  to the output variable `result`
- by *transitivity*, the full sequence can be regarded as causing an *information flow*
  from `init` and `x`
  to `result`

Stepwise analysis: guarded command

```
if guard ≥ 0 then help := help + x else help := help + y fi
```

- the branch is selected by the actual parameter value of `guard`
- in general, observing the value of `help` after this command is executed does not enable an information gain about the guarding variable
- such a gain is possible, e.g., with the additional *a priori knowledge*:
  - the value of `help` is 2
  - the value of `x` is greater than or equal to 8
  - the value of `y` is less than 8
- the observed value of `help` is greater than or equal to 10 *iff* the value of `guard` is greater than or equal to 0
- the observed value of `help` is less than 10 *iff* the value of `guard` is less than 0

A classification of information flows

- ***direct* information flow** (direct *data flow* or *message transmission*)
  - a value (not known to be a constant)
    is explicitly transported from a variable to another one
  - assignment commands,
    passing actual parameters,
    providing arguments for the computation of a function
- ***indirect* information flow**
  - from the arguments to the value of the computation of a
    function
- ***transitive* information flow**
  - two “matching” information flows are combined
  - command sequences,
    nested expressions
- ***implicit* information flow**
  - a guarded command has an impact on the control;
    from the constituents of the guarding expression into the selected branch
  - conditionals,
    repetitions

Reachability or actual reaching of a command

- *formally declared* information flow:
  the pertinent command is part of the program (seen as a text)

- *realizable* (or *existential*) information flow:
  the pertinent command is reachable
  for at most one execution with appropriate input values

- *realized* (or *occurring*) information flow:
  the pertinent command is actually reached during an execution

Implicit flows without any direct flows: example

```
procedure implicit(
  in    x: boolean;
  out   y: boolean);

  local z: boolean;

begin
  y:= false;
  z:= false;
  if x then z := true fi;
  if z then y := true fi
end implicit
```

- implicit flow from `x` to `z` by the guarded command **`if`** `x` **`then`** `z := true`
- implicit flow from `z` to `y` by the guarded command **`if`** `z` **`then`** `y := true`
- transitive flow by sequencing the implicit flows

Implicit flows and the constantness problem: example

```
procedure difficult(
  in   x: integer;
  out  y: integer);

  function f(z: integer) : integer;
  { f computes a total function, as implemented by the body;
    f returns the output value 0 on the actual input parameter value z = 0}
  begin ... end f;

begin
  if f(x) = 0
  then y := 1
  else y := 2
  fi
end difficult
```

Undecidability of information flows

- the function constantly returns 0:
  - equivalent to the assignment command `y := 1`
  - no information flow

- there exists an actual input parameter value $z \neq 0$ such that the function returns a different value:
  - enables a partial gain of information
    about the actual parameter value x, by excluding
    either the value z
    or the specially treated value 0

- an information flow occurs
  iff
  the locally defined function is non-constant
  (in general undecidable)

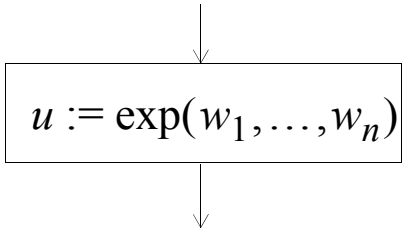
Static compiler-based verification

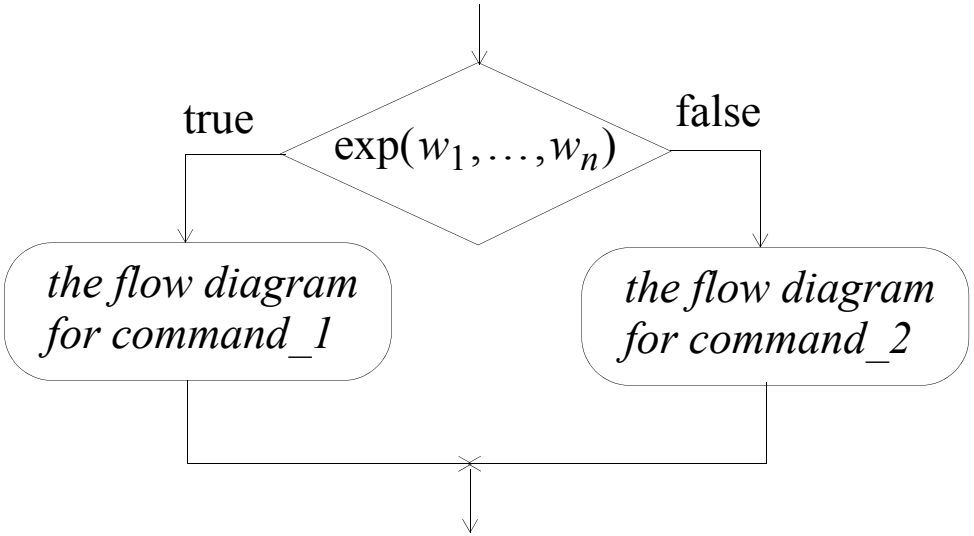
- is a *preventive* mechanism of *inference control*
- employs the structures of high-level procedural programming languages in order to deal with *implicit information flows* due to *guarded commands*
- *approximates* the information flow from the argument components of the guard into the carriers manipulated in the pertinent scope of the guard:
  - this scope is easily determined syntactically
  - leaving the scope, the impact of the guard is reset appropriately
- is *integrated* into the functional analysis of the program, on the basis of *compositional and procedural semantics*
- is supported by a pertinent *compiler*
- considers the progressively more complex *syntactical subparts* of a program as some kind of *carriers* of information

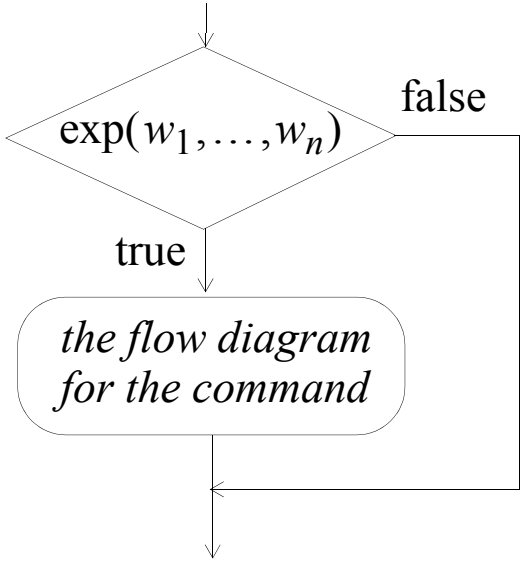
Simplified version of a procedural language

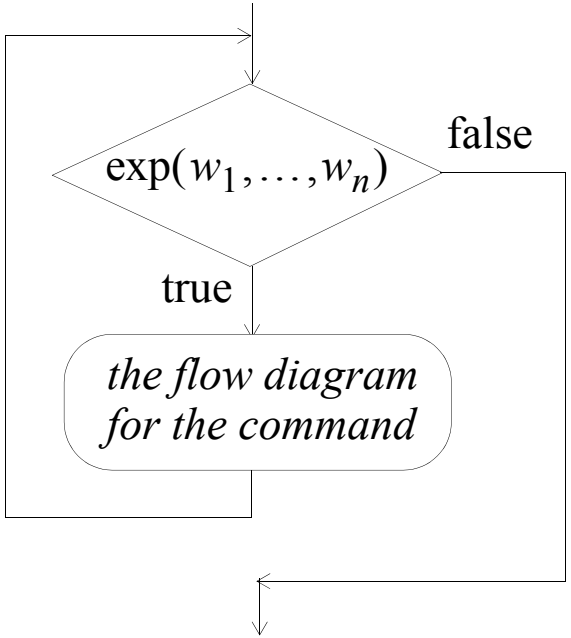
- $x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots$ typed variables (variable v extension D_v)
  $x_1, x_2, \ldots, y_1, y_2, \ldots$ declared as formal parameters
  $z_1, z_2, \ldots$ declared as local variables
- command: one of the following well-formed constructs
  - *assignment* to a variable (with an *expression* to be evaluated and assigned)
  - *sequence* of commands (bracketed by `begin`–`end`, "`;`" used as delimiter)
  - structured *conditional* (*conditional forward jump*/two-sided *alternative*)
  - *repetition* (guard in front of the body: `while` instruction)
  - *procedure call* (with appropriate *actual parameters*)
- *procedure*: identifier, *formal parameters*, *local declarations*, *body*
- *body*: *sequence* of commands without global variables
- *formal parameter*
  - either *argument parameter* (no assignments allowed)
  - or *result parameter* bzw. *argument*/*result parameter* (preceded by `var`)
- *program*: procedure, which might contain nested local procedures

Informal semantics: flow diagrams for commands

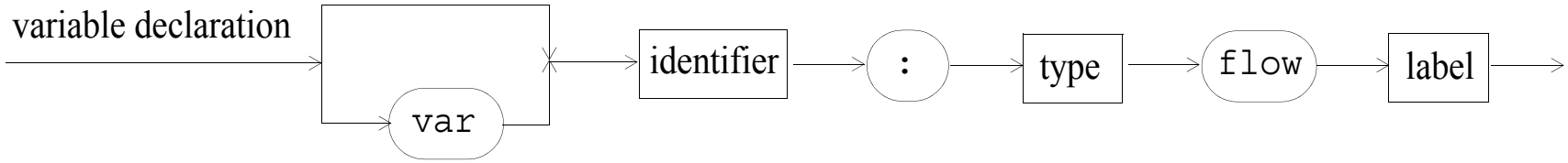








Policy specification for expressing permitted information flows



- labels are taken from the *power set lattice* $(\wp Var, \subseteq, \cap, \cup)$ with respect to the set of all variables $Var = \{x_1, \ldots, x_m, y_1, \ldots, y_n, z_1, \ldots, z_k\}$
- declaring the static label $sl(v) = V \subseteq Var$ for a variable $v \in Var$ is to *permit* only information flows into v that originate from the variables in V
- the following restrictions apply:
  - a formal *argument parameter* x_i must get $\{x_i\}$ as its static label
  - a formal *result parameter* y_j might get a static label V_j such that $y_j \notin V_j \subseteq \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$, not containing any other pure result parameter
  - a formal *argument/result parameter* y_j might get a static label V_j such that $y_j \in V_j \subseteq \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$, not containing any pure result parameter

A procedure declaration with static labels

```
proc max(
      x1 :  integer  flow {x1} ;          argument parameter
      x2 :  integer  flow {x2} ;          argument parameter
 var  y  :  integer  flow {x1, x2}        result parameter
           ) ;

begin
 if   x1 > x2   then   y := x1
                else   y := x2            returns maximum
end
```

Enforcing the intention of the static labels

- during the *syntactical analysis*:
  - for all expressions and commands: a *dynamic label* $dl(.)$ [numbered box]
  - *control invariant* concerning the variables: $dl(v) \subseteq sl(v)$
  - additional *control conditions* concerning the compositional structures

- *control conditions*:
  - expressed in terms of the dynamic labels (circle)
  - dynamically generated and verified

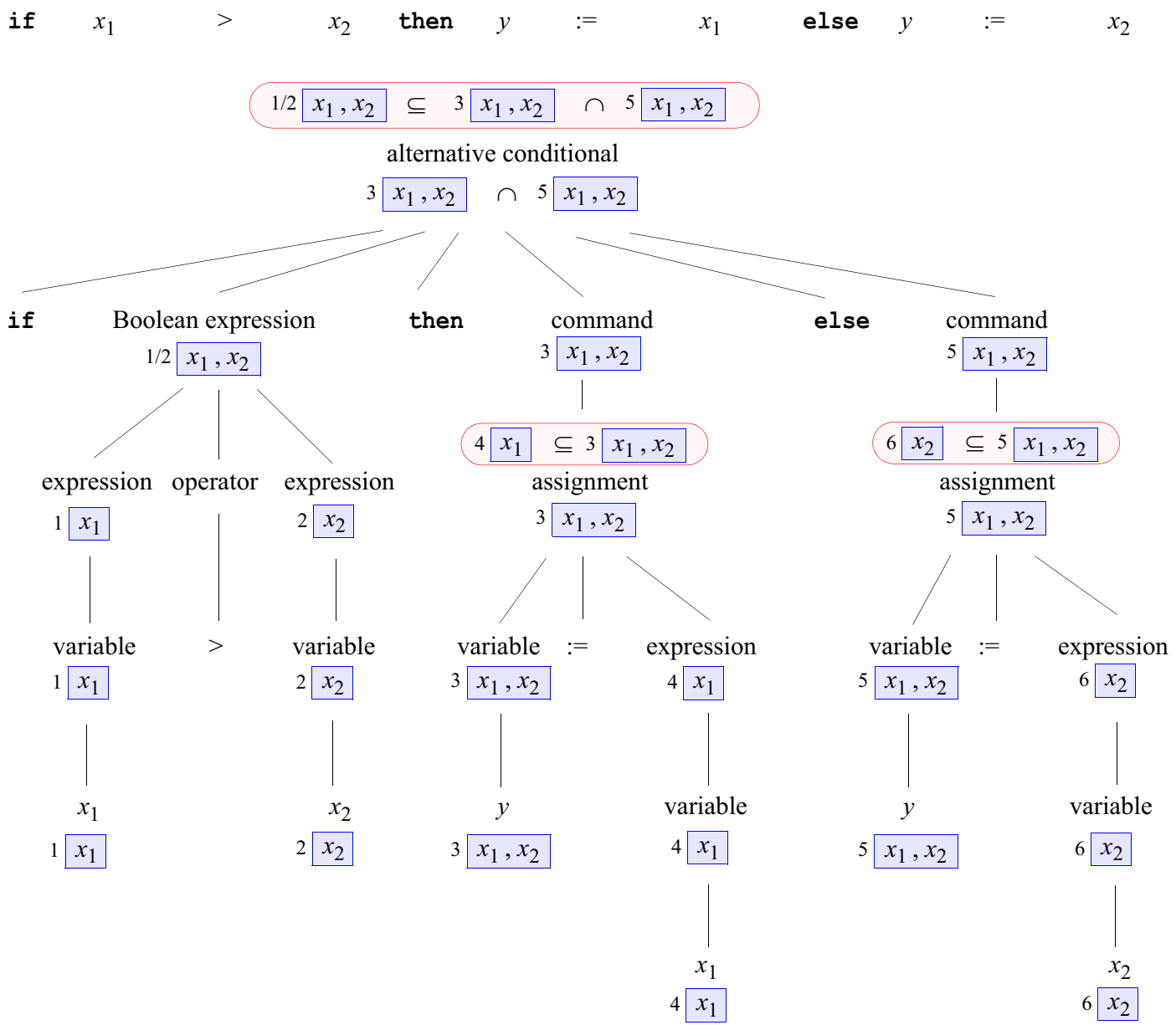
- initially (at leaves of syntax tree):
  - dynamic label of an occurrence of a *variable*:
    the respective static label (declared in the corresponding `flow` clause)
  - dynamic label of an occurrence of a *constant*:
    least label (empty set)

- afterwards (at inner nodes of syntax tree):
  - define dynamic labels stepwise in a *bottom-up* fashion
  - appropriately *propagate* the already available data up the syntax tree
  - *approximate* the relevant *information content*

Guidelines for verification rules

- **functional *expression*:**
  approximate information flow by the
  $\subseteq$-supremum of the labels of the arguments
- ***assignment*, including a *procedure call* (like a multiple assignment):**
  require following control condition:
  label of the receiving carrier dominates labels of the data to be transported
- ***guarded command*:**
  require following control condition:
  an *implicit flow* of the information
  represented by the label of the guarding expression
  is permitted for *all* assignments in the scope of the guard
- ***composed* command (*sequence* or *alternative*):**
  refer to *all* constructs,
  the assigned label is computed as the $\subseteq$-infimum of the contributing labels
- ***reaction*:**
  refuse execution after detecting a policy violation

Defining dynamic labels and generating control conditions: example



Dynamic labels and control conditions

Expression/command e / C	**Assigned dynamic label** $dl(e)$ / $dl(C)$	**Generated control condition**
constant: $e \equiv const$	$\varnothing$, i.e., least element	
variable: $e \equiv v$	$sl(v)$, i.e., static label	
functional expression: $e \equiv f(w_1,\ldots,w_n)$	$dl(w_1) \cup \ldots \cup dl(w_n)$	
assignment: $C \equiv u := e$	$dl(u)$	$dl(e) \subseteq dl(C)$
sequence: $C \equiv$ `begin` $C_1,\ldots,C_m$ `end`	$dl(C_1) \cap \ldots \cap dl(C_m)$	
conditional jump: $C \equiv$ `if` e `then` C_1	$dl(C_1)$	$dl(e) \subseteq dl(C)$
alternative: $C \equiv$ `if` e `then` C_1 `else` C_2	$dl(C_1) \cap dl(C_2)$	$dl(e) \subseteq dl(C)$
repetition: $C \equiv$ `while` e `do` C_1	$dl(C_1)$	$dl(e) \subseteq dl(C)$
procedure call: $C \equiv P(a_1,\ldots,a_m,b_1,\ldots,b_n)$	$dl(b_1) \cap \ldots \cap dl(b_n)$	for $x_i \in sl(y_j)$: $dl(a_i) \subseteq dl(b_j)$ for $y_i \in sl(y_j)$: $dl(b_i) \subseteq dl(b_j)$

Compiler-based verification: theorem

Let P be a procedure with a totally defined semantic function $|P|$.

If P satisfies all generated control conditions,
then the following property holds:

> for any execution of the calling of P,
> any *realized information flow* from a variable v to a variable w
> is *permitted* according to the declaration of the static label of w, i.e.,
> all other information gains are *blocked*.

- proof:
  by a structural induction
- perspective:
  great potential, also regarding further advanced programming constructs,
  provided that
  - all constructs are compositionally structured
  - all constructs carefully avoid unforeseen side effects
  - the approximations are acceptable (do not cause too many rejections)

Resetting and downgrading dynamic labels

- whenever a *program variable* is assigned a new value,
  the previously held value is supposed to be lost:
  the dynamic label of the variable is redefined

- as a special case,
  if the expression is just a *constant*,
  then the new label is the least element of the lattice employed:
  the dynamic label is completely reset

- whenever a *structured command* is properly left,
  control is supposed to *forget* the value of guarding expression:
  the dynamic label of the pertinent version of a *control variable*
  is reset to its value before the structured command was entered

- unfortunately, there seem to be no further generally applicable techniques for
  forgetting information

Decentralized label model: main emphasis

provide each individual *owner* of some information
with a flexible and expressive means to specify the allowed receivers,
when the execution of a program is shared

Decentralized label model: outline

- *label*: set of *policies* consisting of an *owner* and a list of *readers*:

  $\{ (owner_1: reader_{1,1}, \ldots), \ldots, (owner_k: reader_{k,1}, \ldots) \}$

- assigning a label to
  a *carrier* (input channel, internal program variable, output channel, ...) or
  some *data* (result of evaluation of an expression, ...):

  the respective information content is permitted
  to be *transferred* to a principal *prin*
  iff
  that principal is a *grantee* of *all* policies in the label, i.e.,
  iff
  $prin \in \{owner_1, reader_{1,1}, \ldots \} \cap \ldots \cap \{owner_k, reader_{k,1}, \ldots$

- while information is being processed during the execution of a program,
  the *static label* of the receiving carrier must always be
  at least as restrictive as the *dynamic label* of the transferred data:

  each grantee for the receiving carrier is a
  grantee for the transferred information as well

- while information is flowing,
  the labels assigned to a piece of information might become more restrictive:
  any deposit of a piece of information in a labeled carrier
  possibly *excludes* principals from accessing this (copy of the) information

- to maintain the needed *availability* of information,
  any of the owners can dynamically relax the exclusions by
  somehow *downgrading* (*declassifying*) their part of the label:
  by generating a copy of the information with a label
  that is less restrictive with respect to this owner's grantees

- an owner can achieve this goal only if
  - he is a member of the specific *authority set of principals*
    on behalf of which the program execution is performed, and
  - the program contains a suitable command dedicated to such a relaxation

- when the dedicated command is executed,
  a copy is generated with a new label
  where this owner's part is modified as described in the command

Programming language Jif (Java Information Flow)

- implements features of the decentralized label model
- extends (a sublanguage of) *Java*
- provides inference control by *static verification* of labeled programs
  as an extension to type checking:
  - analyzing the main constructs of Java for all kinds of information flows
  - verifying the pertinent control conditions
- demands some limited *dynamic monitoring* for downgrading
- *dynamic monitoring* also extends
  to granting authorities along chains of procedure calls,
  or dealing with additional runtime labels

Inference control for parallel programs

- there are constructs for the *parallel* execution of several threads
- threads coordinate their actions and
  synchronize at specific points of their execution and
  thus introduce new kinds of *implicit information flows*
- suppose that one thread can only proceed
  if another thread has completed some specific actions:

  then the latter thread appears like a *guard* for the former one

Inference control for parallel programs: example

```
x,y,z:  boolean;
s:      semaphore;

begin
   z:= false;

   cobegin
      thread_1: read(x);
                if x then signal(s)
   ||
      thread_2: y := false;
                wait(s);
                y := true
   coend;

   z:= y
end
```

Inference control for parallel programs: analysis of the example

- accessing only the program variable `y`,
  an observer of `thread_2` can possibly infer the value of `x` read by `thread_1`:

  since the assignment `y:=true` is guarded by the semaphore,
  which in turn is in the scope of the guard `x` in the conditional,

  observing the value `true` for `y` implies
  that `x` has been given the value `true` as well

- accessing only the program variable `z`,
  an observer can possibly infer the value of `x`:

  since `thread_2` terminates only if `thread_1` signals the semaphore
  and
  the synchronization occurs only if both threads terminate,

  observing the value `true` for `z` implies
  that `x` has been given the value `true` as well

Inferences based on covert channels

- semantics of programs is defined in terms of an abstraction
  designed to appropriately model the behavior of real computing devices

- inference control – as presented so far – refers to the pertinent abstraction

- accordingly, inference control correctly captures only those information flows
  that can be described in terms of the abstraction,
  but fails to deal with potential further flows over *covert channels*

An never-ending list of possibilities

- *timing channels* exploit observable differences in behavior in real time
- *energy consumption channels* exploit the fact that
  (hidden) different behaviors are related to
  observable differences in energy consumption
- similarly, other *physical effects*
  such as measured electromagnetic fields could be exploited
- *storage channels* exploit the status of shared storage containers
- *exception-raising channels* are based on observable parts of
  the exception handling within some protocol,
  where an exception is either triggered as
  an *observable* event within a specific context or not
- ...

Some countermeasures against detected covert channels

- "close a detected covert channel"
  by explicitly taking care that originally distinguishable events
  become *indistinguishable* for the suspected observer

- make the real execution time independent of some crucial input values,
  by performing *dummy operations* if necessary

- decouple consumers of shared resources,
  by assigning predetermined access times

- unify protocol executions,
  by eliminating case-dependent exceptions

Inference control for statistical information systems

- consider a specialized kind of *information system* and
  the dedicated usage of such systems for *statistical* purposes
- use here a simply model of an information system:
  - r an instance
  - $R(K, V)$ a relation scheme
  - K attribute, declared as a *key* and
  - V attribute, seen as some dependent *property*
    whose values are *real numbers*
- interpret each tuple (k, v) in the instance r:
  - key value k abstract identifier uniquely denoting an *individual*
  - property value v some *personal data*, to be protected (kept secret)
- regulate access according to *protection rules for personal data*, e.g.:
  seeing answers to statistical queries
  (e.g., mean or median for some sample sets of individuals)
  the *statistician* must not be able to infer
  the property of any particular individual

- resolve *conflict* of *interests*:
  - statistician: *availability* of statistically aggregated data
  - individuals: *confidentiality* of their personal data

- take care that, in general,
  there are no simple means to resolve the conflict:
  - system *refuses* to give answers to immediately harmful queries
    (e.g., queries related to samples of a size too small for hiding):

    statistician might design sufficiently long query sequences
    to set up a solvable system of equations
    whose solution reveals some particular personal data

  - system explores *lying*
    (e.g., replacing the correct values by (statistically) distorted values
    adding some random “noise”)
    such that anticipated statistical queries are not “essentially affected”:

    statistician might design calculations for “noise removal”

Summation as aggregate function: a functional model

- r (hidden) fixed instance
  N known size of r
  $\{1,\ldots,N\}$ key values occurring in r

- q a query determining a *sample* set $sample(r,q)$ of identifiers

- query language is closed under Boolean combinations, e.g.,
  $sample(r, q_1 \vee \neg q_2) = sample(r,q_1) \cup (\{1,\ldots,N\} \setminus sample(r,q_2))$

- statistical *aggregate function* is *summation*:
  on input of a query q,
  the system returns the result

$$sum(r,q) = \sum_{k \in sample(r,q)} \; \sum_{(k,v) \in r} v$$

Summation as aggregate function: a refusal approach

- $t < N/2$ some suitable threshold parameter

- the system *refuses* the answer to a query q
  iff
  card $sample(r,q) < t$ or card $sample(r,q) > N - t$

  (the cardinality of the sample set is either *too small* or *too large*)

Summation as aggregate function: a refusal situation

- supposed observation:
  the system *refuses* a query q

- additional assumptions:
  - threshold t is suitably small
  - observer has some helpful a priori knowledge
  - observer can select a query $q_{tracker}$ such that
    $2 \cdot t \leq \text{card}\, sample(r, q_{tracker}) \leq N - 2 \cdot t$

Summation as aggregate function: a circumvention procedure

- observer: submits queries $q_{tracker}$ and $\neg q_{tracker}$
- system: answers correctly

- observer: submits the queries $q_1 \equiv q \vee q_{tracker}$ and $q_2 \equiv q \vee \neg q_{tracker}$
- system: reacts
- observer:

  case 1, system correctly returns both answers [sample set too small]:
  derives the refused result $sum(r,q)$ by solving the *linear equation*
  $sum(r,q_1) + sum(r,q_2) = sum(r,q_{tracker}) + sum(r,\neg q_{tracker}) + sum(r,q)$

  case 2, system refuses the answer to q_1 (similarly for q_2):
  [sample set for q too large; thus sample set for $\neg q$ too small:
  thus case 1 holds for $\neg q$]

  applies circumvention for $\neg q$ and
  infers $sum(r,\neg q)$;

  derives the refused answer $sum(r,q)$ by solving the *linear equation*
  $sum(r,q) + sum(r,\neg q) = sum(r,q_{tracker}) + sum(r,\neg q_{tracker})$

Part III

Security Architecture

9 Layered Design Including Certificates and Credentials

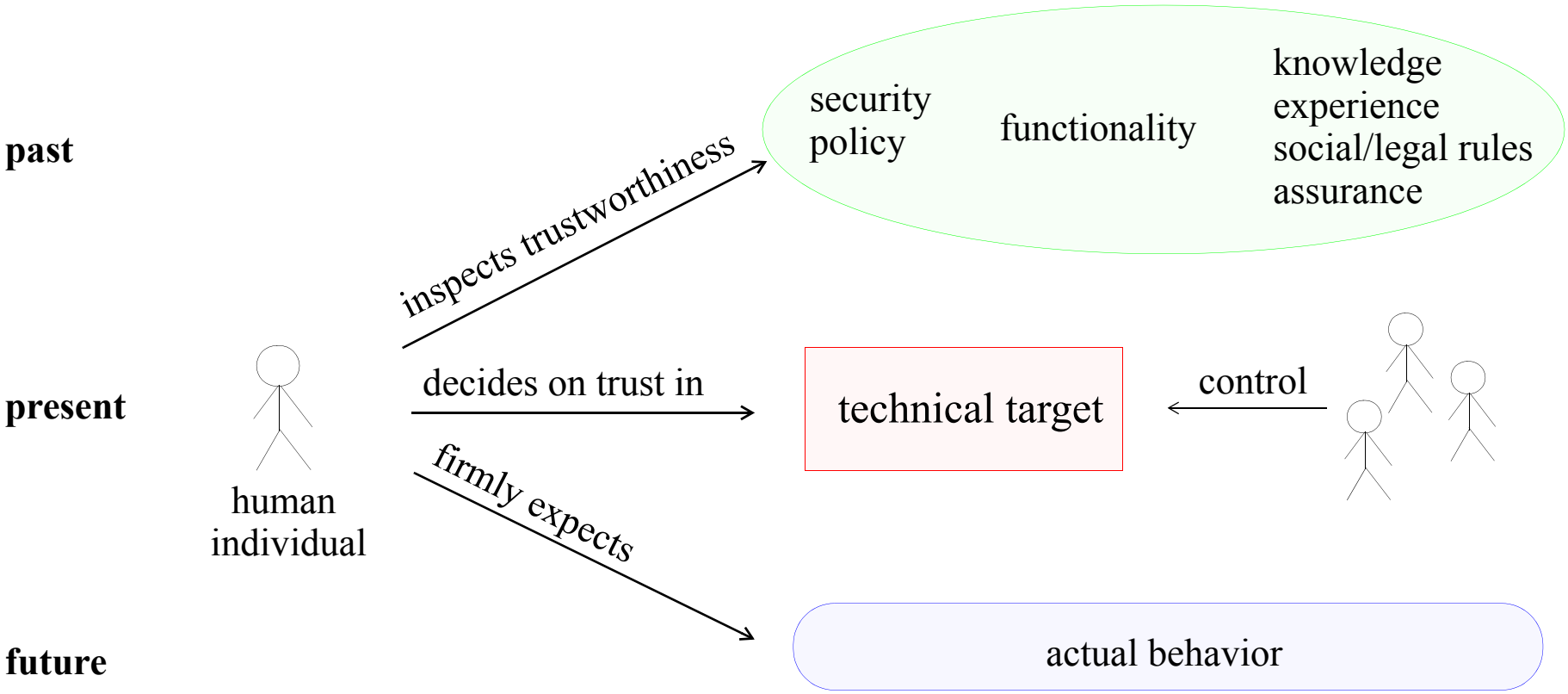
Trust and trustworthiness

- 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

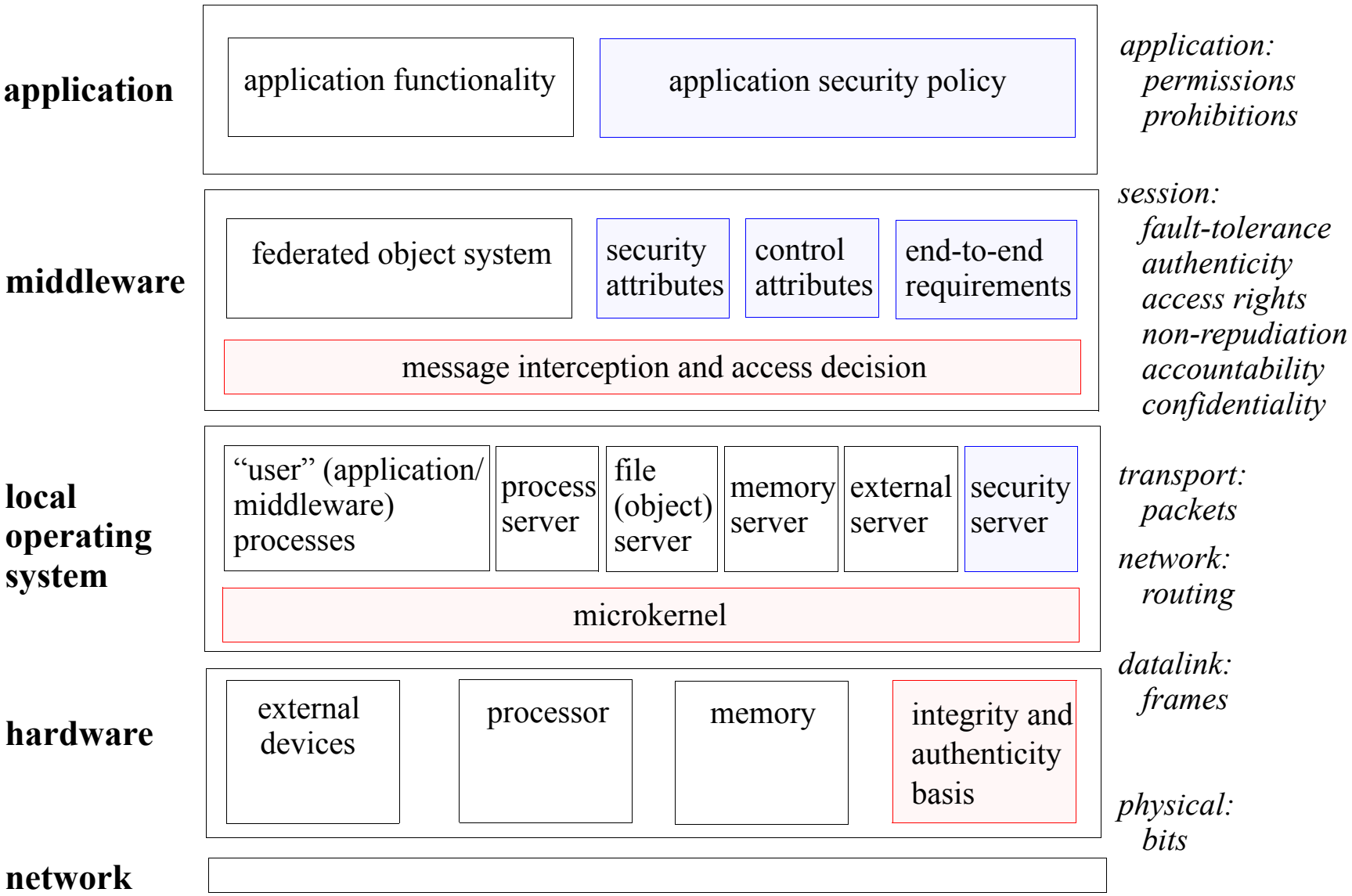
- **extended reduction to hardware support:**
  - “trusted platform modules”
    (enforcing authenticity and integrity)
  - personal computing devices
    (storing and processing cryptographic secret keys)

Trust reductions for cryptography

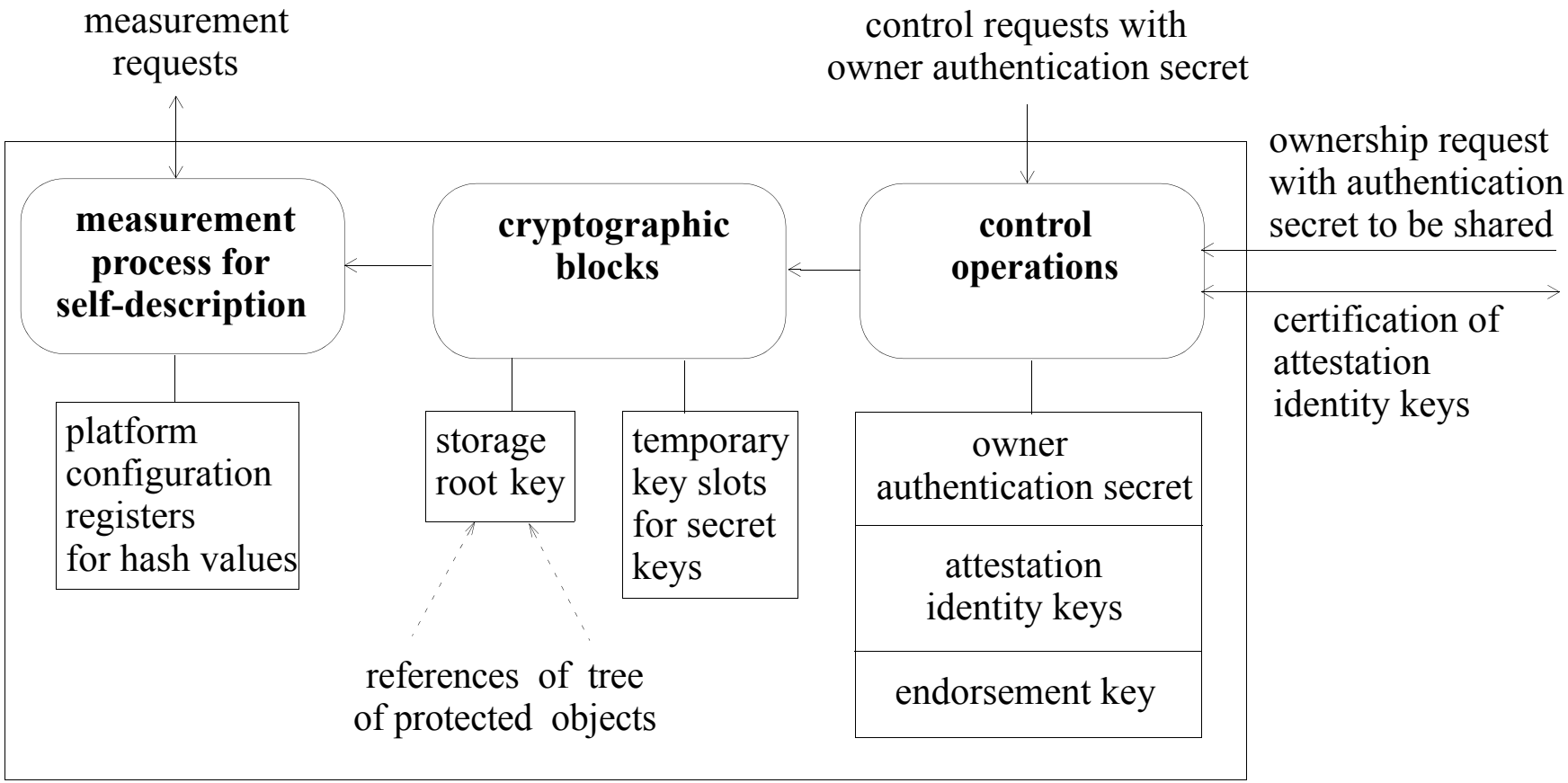
- **starting point:**
  an overall computing system consisting of
  clients, servers, networks and many other components

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

Layered design: a fictitious architecture



Integrity and authenticity basis (trusted platform module)



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

- 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*

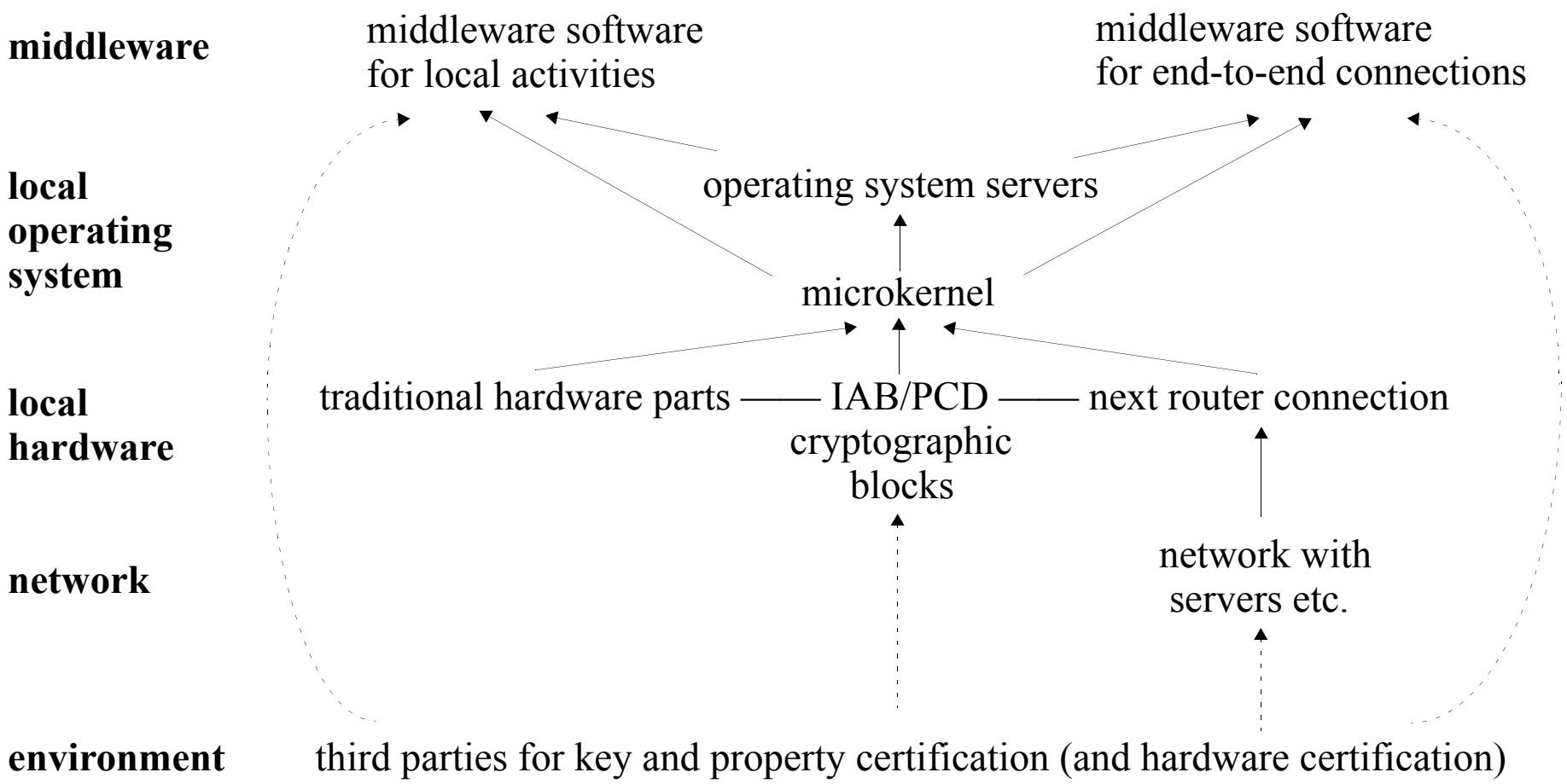
Informational infrastructure and organizational environment

- 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*

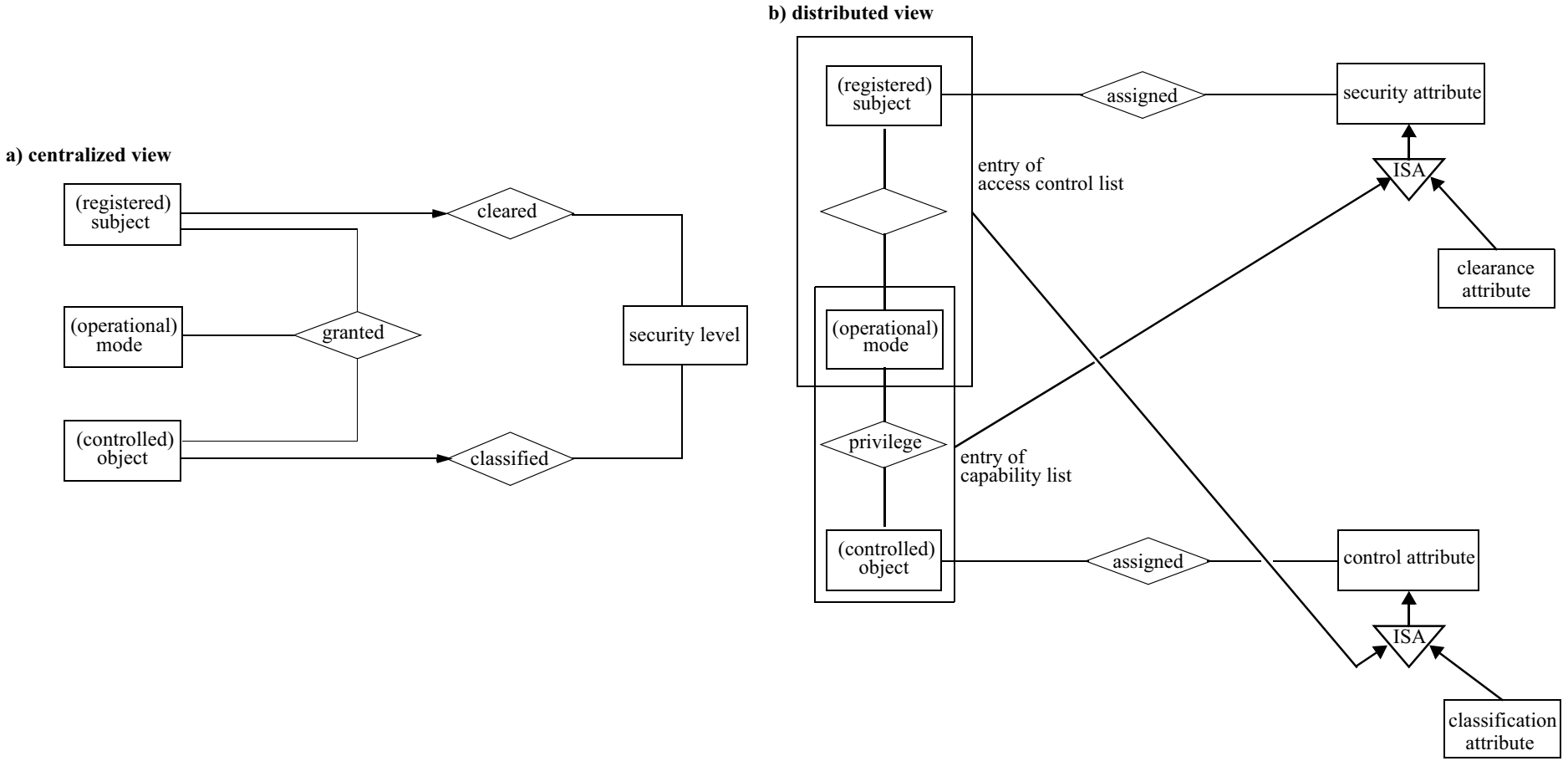
Middleware: support by underlying layers and global infrastructure



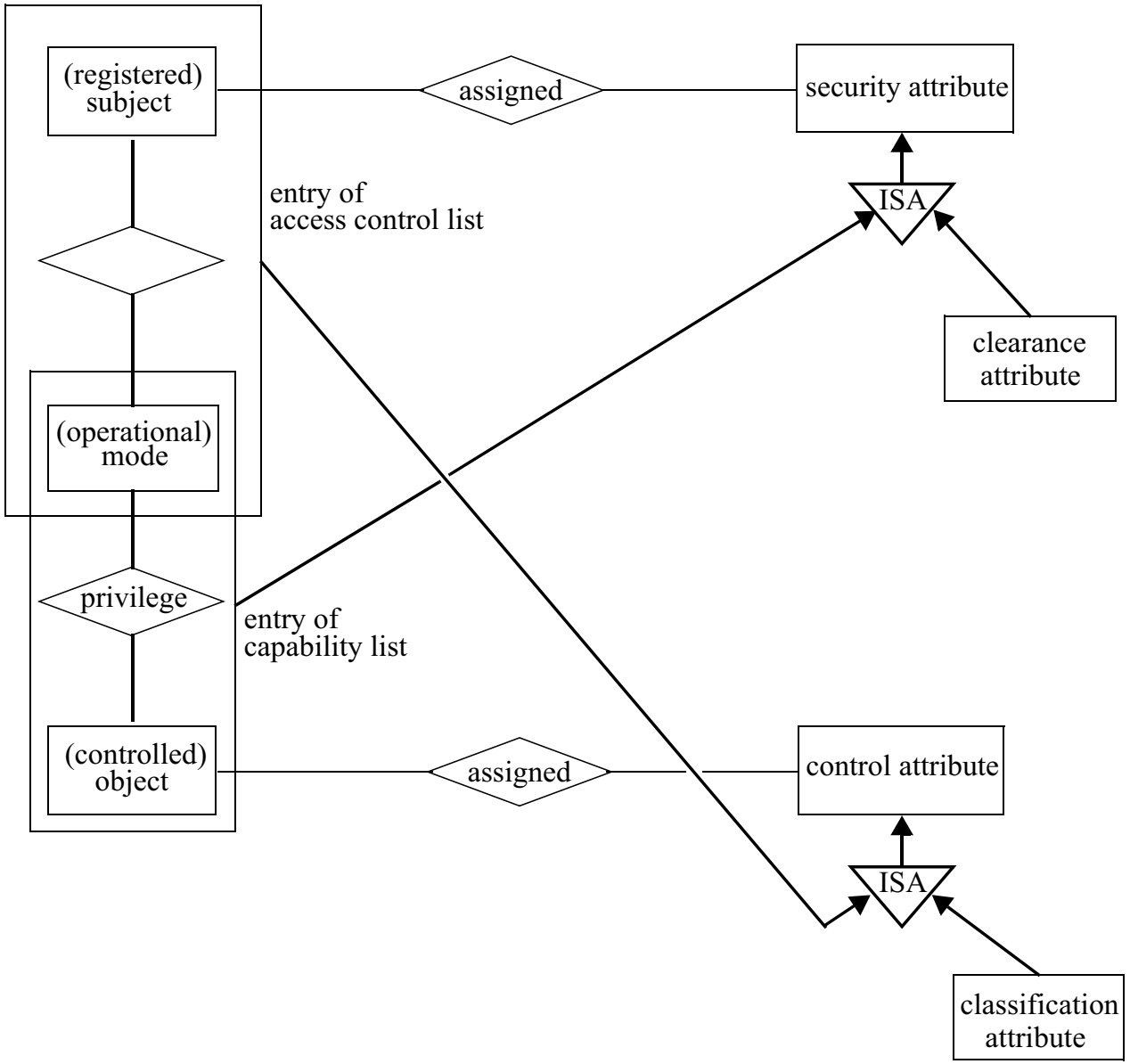
Middleware instantiation of control and monitoring

- 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

- *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

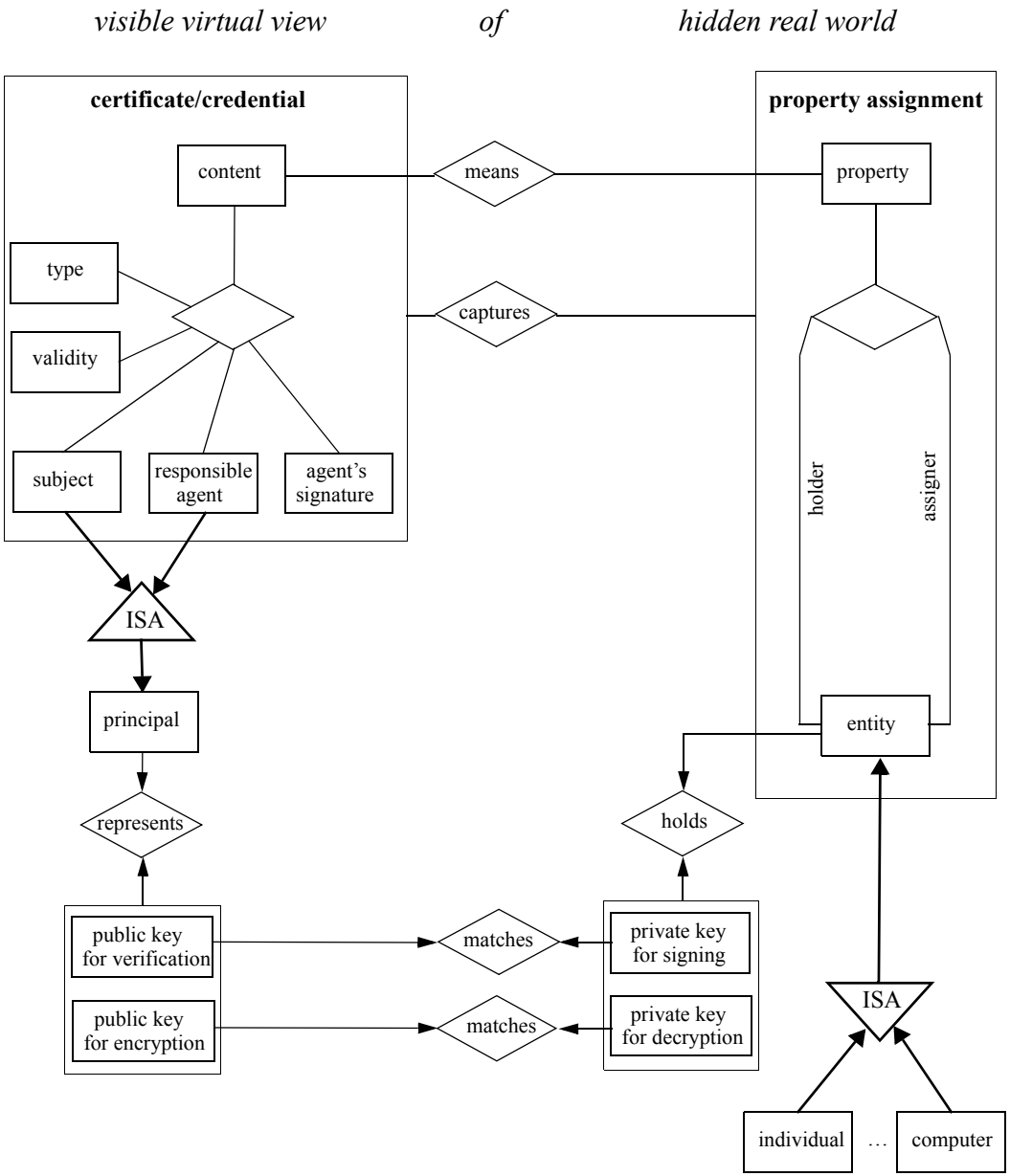
- explicitly *guarding* external input values and output values
- explicitly *guarding* values passed
  for the expected range, well-definedness or related properties
- elaborating a complete *case distinction* for guarded commands
- carefully considering *visibility* and naming conventions
- handling *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*

Distributed systems: real world and virtual view

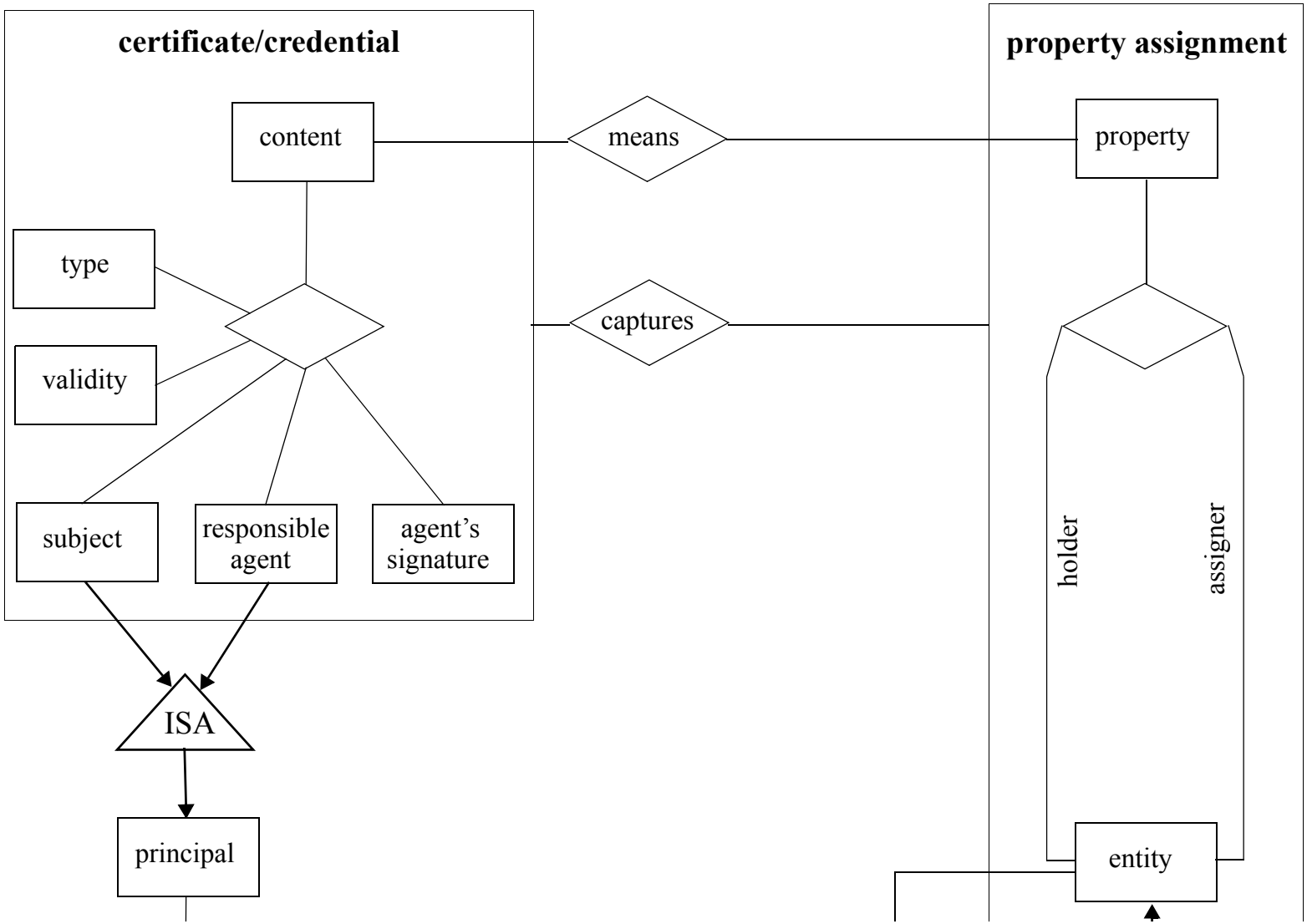
- **(real) world**
  a specific entity cannot directly see other entities:
  - other entities are hidden behind the interface to the communication lines
  - the specific entity can only send/receive messages to/from other entities

- **virtual view**
  that specific entity can produce a view on the basis of messages received:
  - security policies and permission decisions are grounded solely on the locally available visible view of the global (real) world
  - another entity may *possess* various *properties* which might be relevant to security policies and permission decisions
  - in most cases, such properties are *assigned* to an entity by a further entity
  - in general, neither the other entities themselves nor their properties are visible to the specific entity
  - we need a notifiable representation of such circumstances
  - such a representation can be based on a *public key infrastructure*

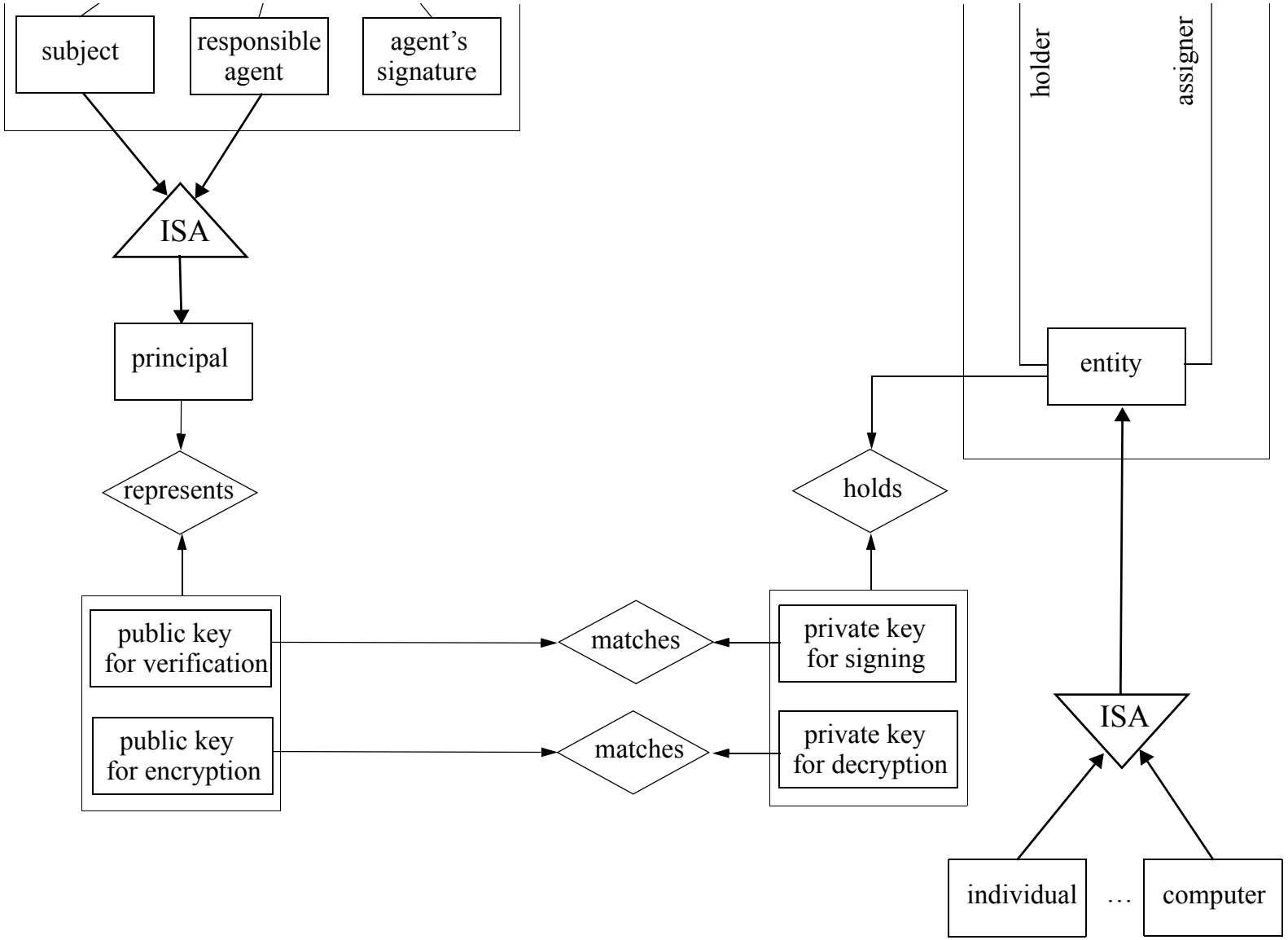
Hidden (real) world and a visible virtual view



Certificates/credentials and property assignment



Principals and entities



Digital document (certificate/credential): important fields

- ***subject***
  contains the principal that visibly represents the entity under consideration
- ***content***
  textually describes the assigned property
- ***responsible agent***
  contains the principal that visibly represents the entity
  that is responsible for the property assignment and
  has generated and digitally signed the document
- ***signature***
  contains a digital signature for the document: valid iff
  it can be verified with the responsible agent's public key for verification

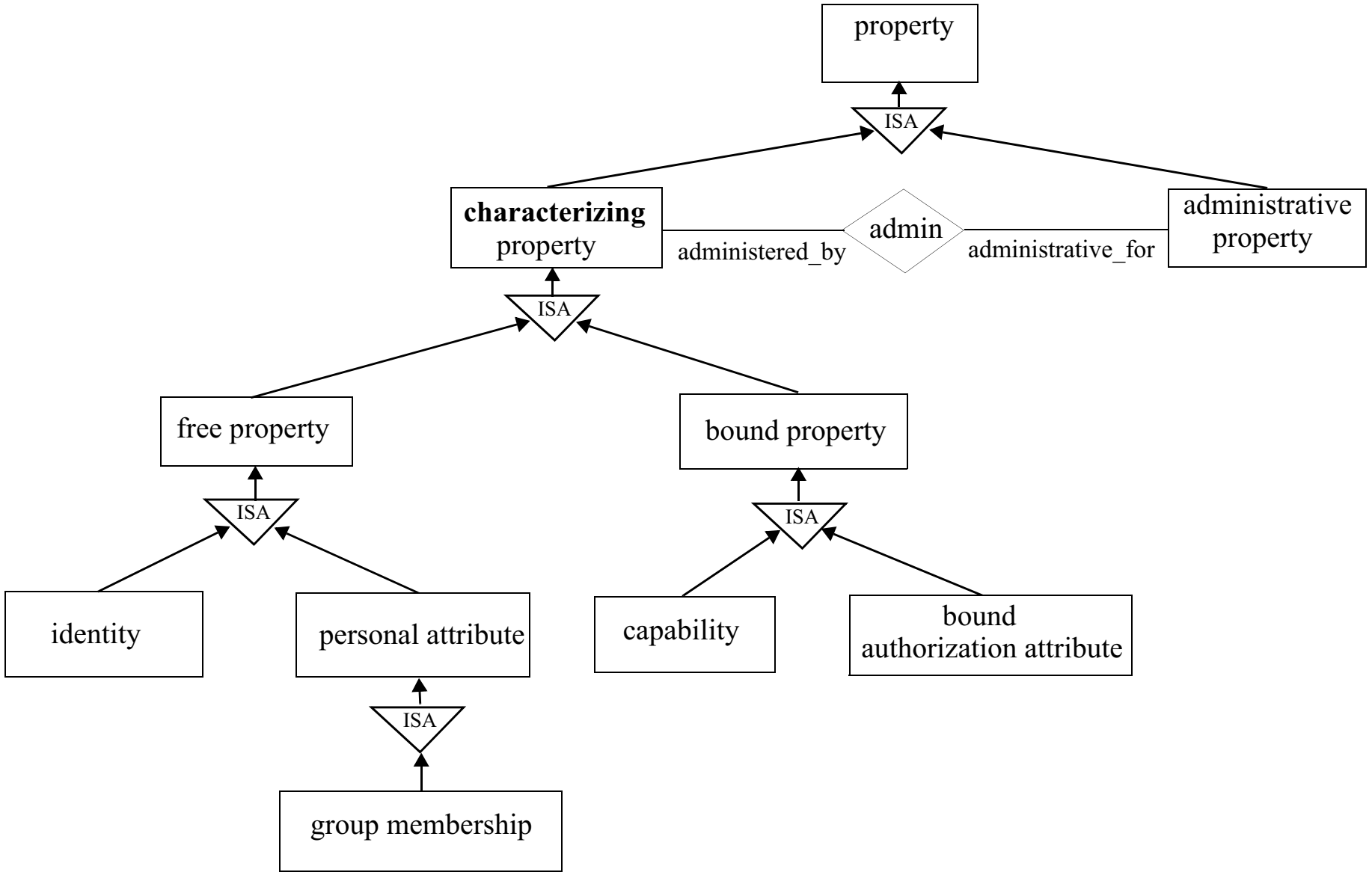
- ***type***
  indicates the meaning/provides hints on how to process the document
- ***validity***
  limits the property assignment to a certain time period or
  restricts the usability of the document otherwise

Characterizing properties: free and bound properties

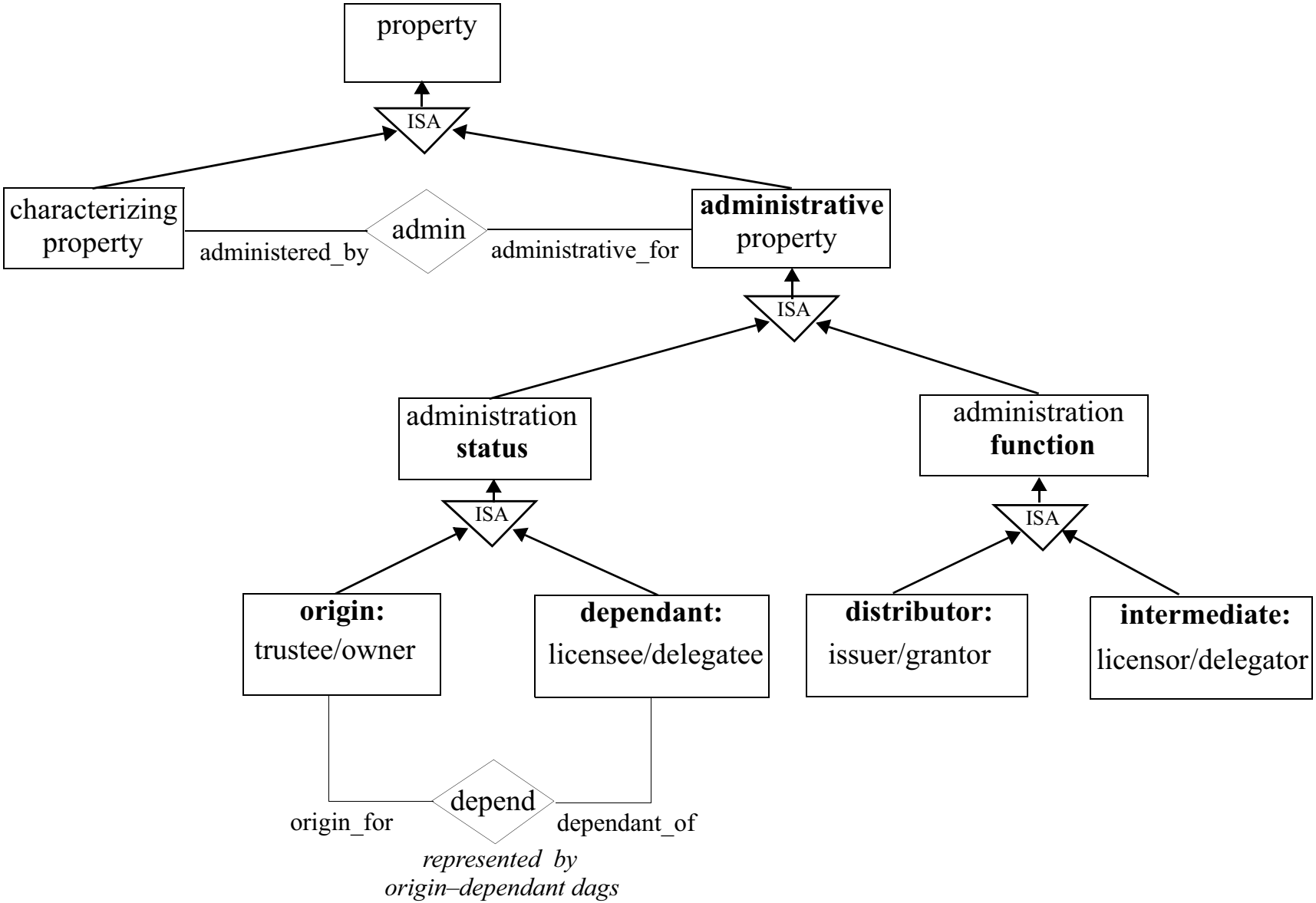
- **free property** (personal data, technical detail, skill, ability, ...)
  expresses some feature of an *entity by itself*:
  - other entities may base their security policies and permission decisions on shown free properties
  - but, in general, they will not have expressed any obligation as to whether to or how to do so

- **bound property** (a ticket, a capability, a role, ...)
  expresses some *relationship* between a client entity and another entity which might act as a server:
  - a server has declared in advance that it will recognize a shown bound property as a permission to use some of its services
  - possessing a bound property entails a promise that a specific service will be obtained

Characterizing properties



Administrative properties



Relationships and trust evaluations

- the relationships of *presumably captured by* are *ideal claims* that do not necessarily hold

- a specific entity has to evaluate its individual *trust* about such an ideal claim:
  - did the supposed assigning entity follow good practice in generating and signing the document?
  - do the principals (keys) appearing in the document represent the supposed entities?

- the very purpose of the administrative properties is just to provide a reliable foundation for such *trust evaluations*

Evaluating trust: basic situation

- ideally,
  permission decisions are intended to be based on
  characterizing properties of entities appearing as clients
- actually,
  permission decisions must be based on available, visible digital documents,
  the contents of which *mean* the respective characterizing properties
- consider any such document as a *main document*
  from the point of view of an entity entitled to take a permission decision:
  - is the literal meaning of the content indeed valid in the (real) world?
  - does the digital document *capture* a "real" property assignment?
- these questions are answered using further *supporting documents*,
  the contents of which mean appropriate administrative properties
- for each of these supporting documents, the same questions arise

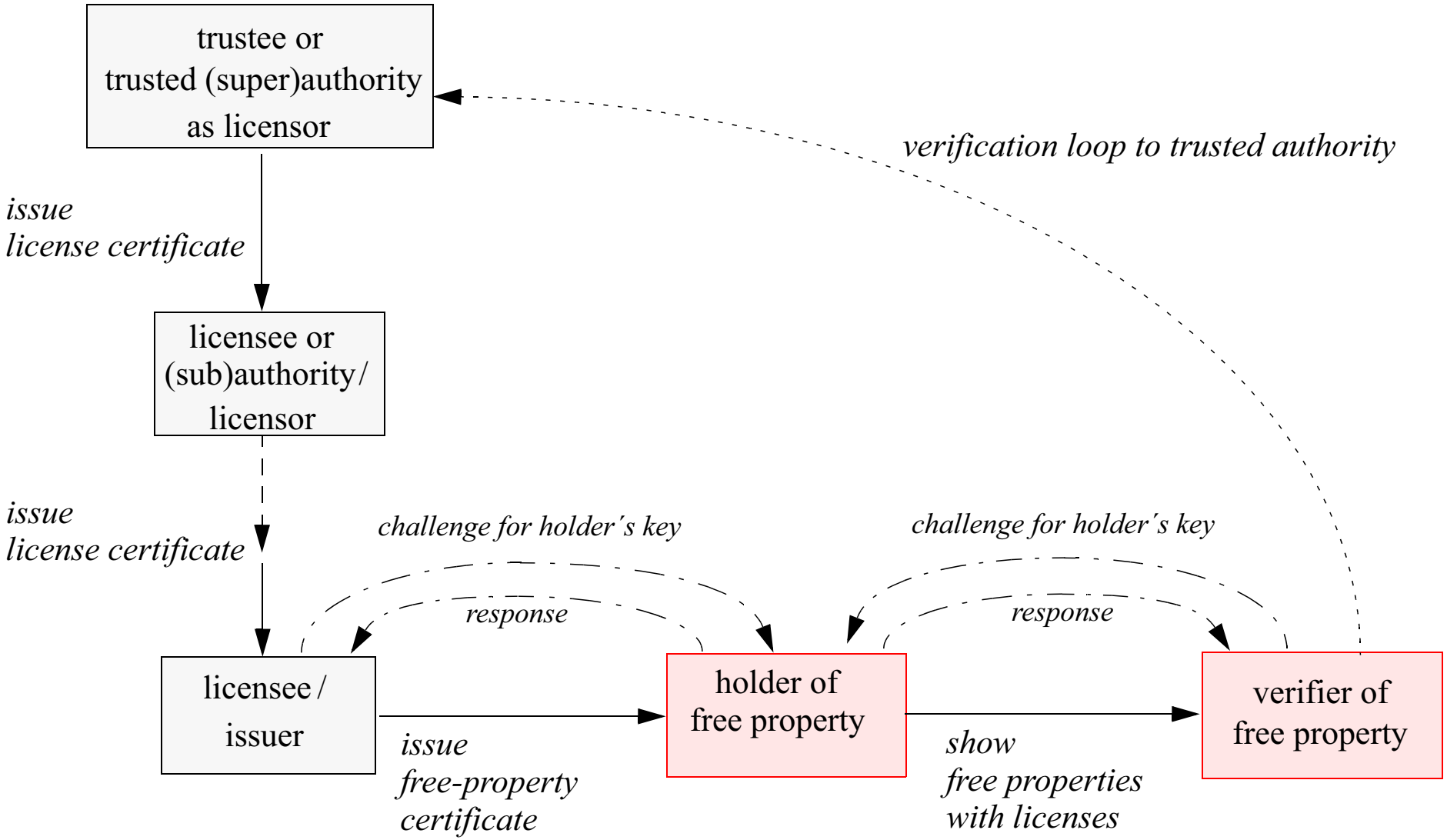
Evaluating trust recursively

- the “main document” concerning a characterizing property
  is supported by a first level of “supporting documents”
  concerning administrative properties for that characterizing property

- for each “supporting document” at the *i*-th level,
  one of the following cases holds:
  - either it is supported by further “supporting documents” at the next level,
    expressing that the responsible agent of the former document
    represents a *dependant* of the responsible agents of the latter documents
  - or it expresses that its responsible agent represents an *origin*
    for the characterizing property administered,
    expressed by the content of the “main document”

- to be helpful, the “main document” and its “supporting documents”
  should form a *directed acyclic graph* with respect to *support*
- as a special case, we may obtain just a *chain*

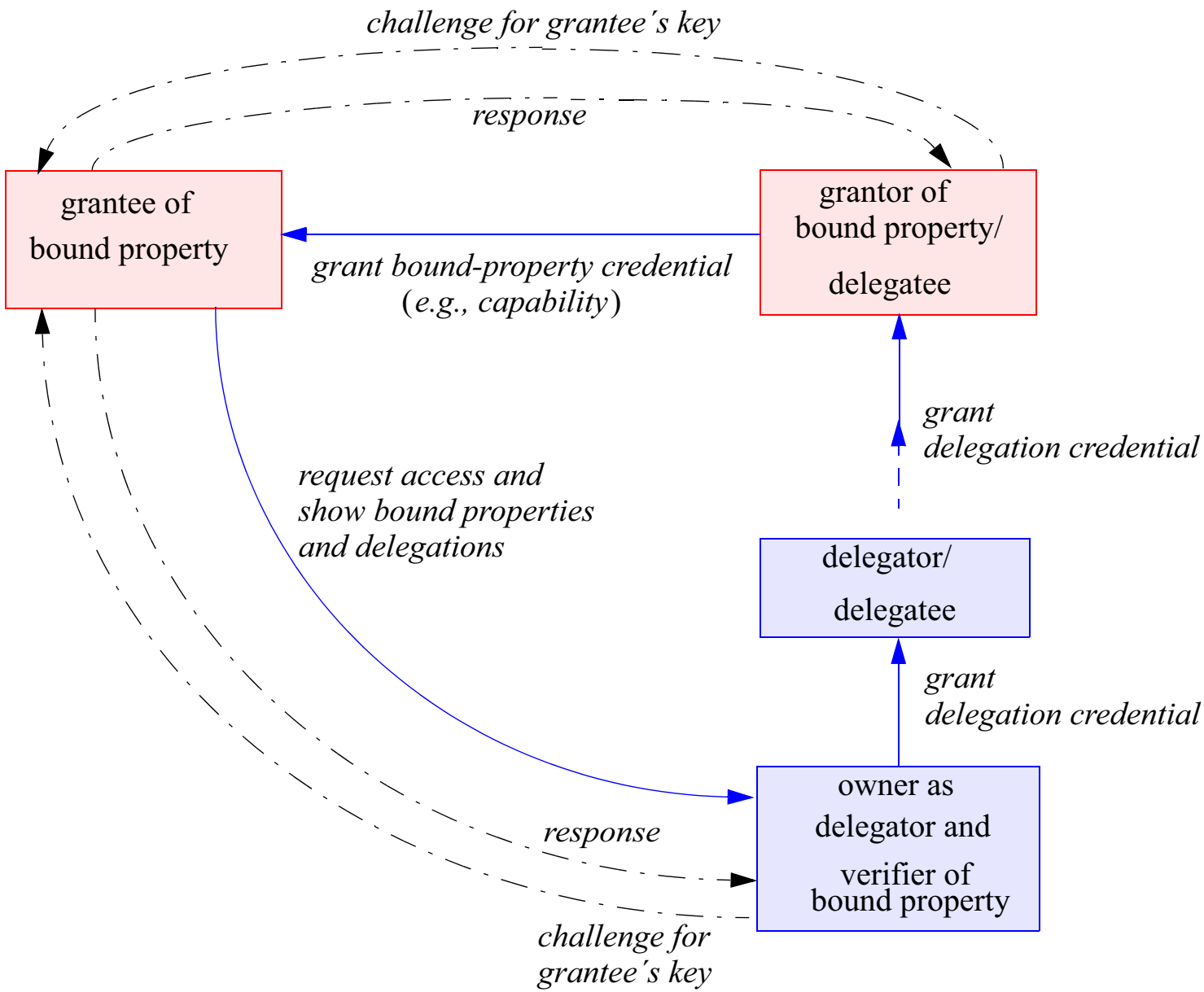
Model of trusted authorities and licensing: an instance



Certificate types in the model of trusted authorities and licensing

Certificate type	**Content**
identity certificate (X.509 term)	identifying name
attribute certificate (X.509 term)	personal attribute
accreditation certificate (mediation term)	personal attribute
private certificate (Brands' term)	personal attribute
trustee self-certificate (X.509 term: root certificate)	*administration status*: trustee
license certificate (X.509 term: certification authority certificate)	*administration function*: licensor

Model of owners and delegation: an instance

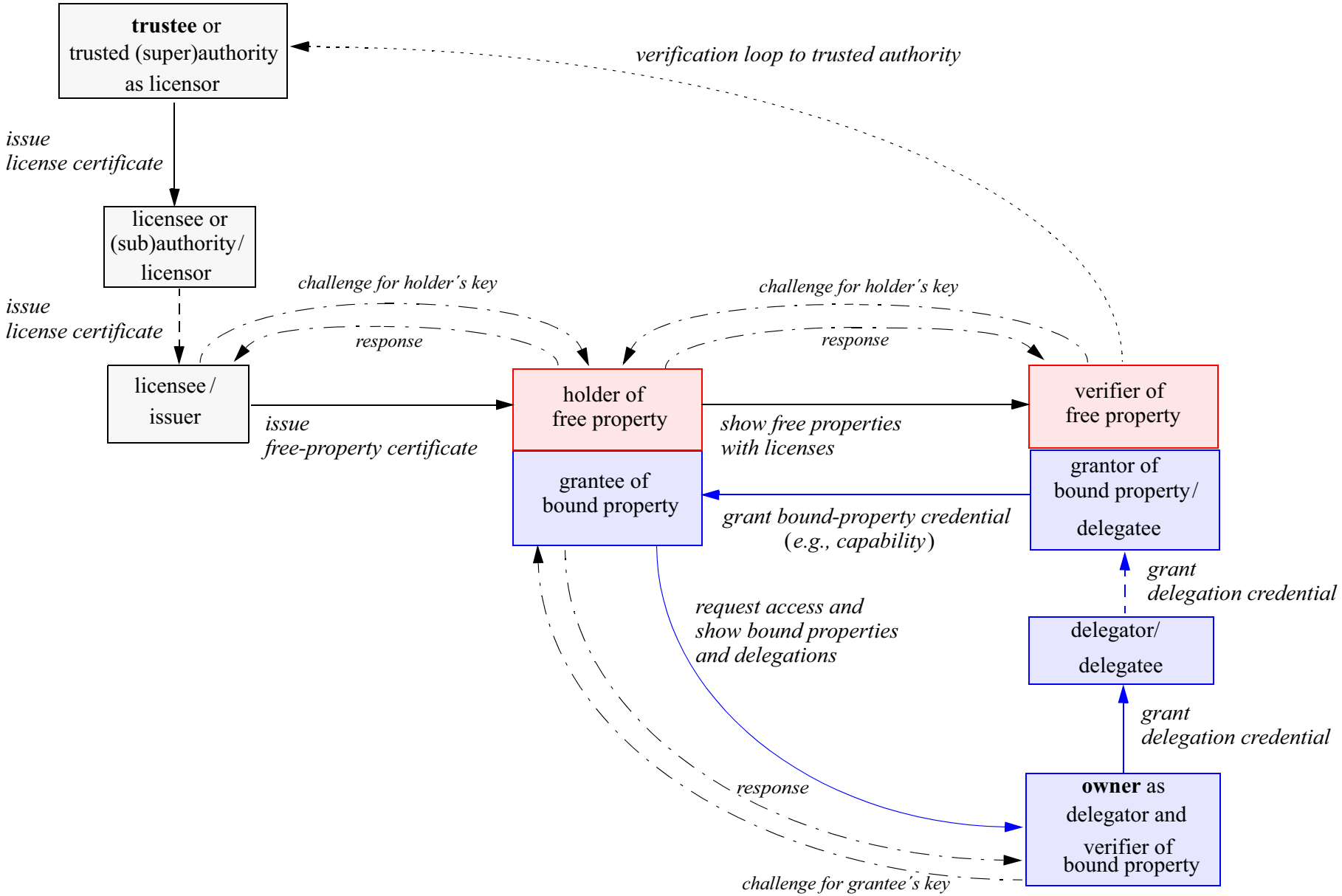


Credential types in the model of owners and delegation

Credential type	Content
capability credential (SPKI term: authorization certificate)	capability
bound-authorization-attribute credential	bound-authorization attribute
delegation credential (SPKI implementation: true delegation bit)	*administration status*: delegatee

.

Converting free properties into bound properties: an instance

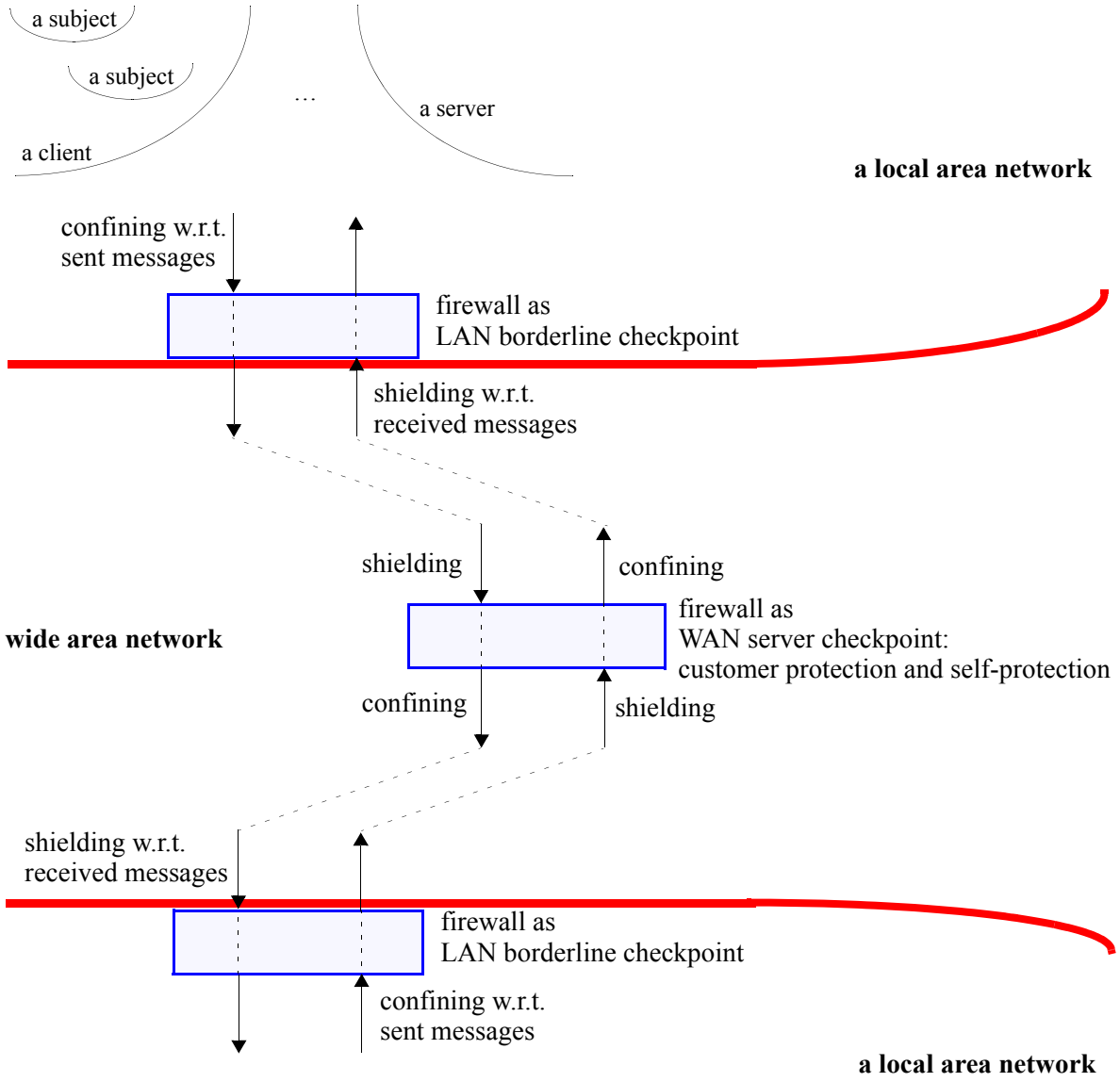


Firewalls

- computing systems in the large are composed from
  partly federated and partly nested structures built from
  - individual subjects
  - shared client computers and servers
  - local area networks
  - wide area networks

- the techniques of control and monitoring are applicable
  at the borderline of any substructure aiming at
  - *confining* the inner side with respect to *sending* messages to the outside,
    thereby restricting
    - the transfer of information to the outside
    - the requests to foreign entities
  - *shielding* the inner side with respect to *receiving* messages,
    thereby restricting
    - interference by foreign entities
    - incoming requests

Firewalls serving as LAN borderline and WAN server checkpoints



Checkpoints handling packets according to ISO/OSI model

a firewall intercepts the packets passing the checkpoint
and examines the following layers, inspecting increasingly complex data:

- network to transport layer: only the packet *headers*
- transport to session layer: *sequences* of packet headers
  (e.g., compliance with session protocols)
- session to application layer: additionally, the packet *contents*
  (e.g., intended semantics of encoded messages)

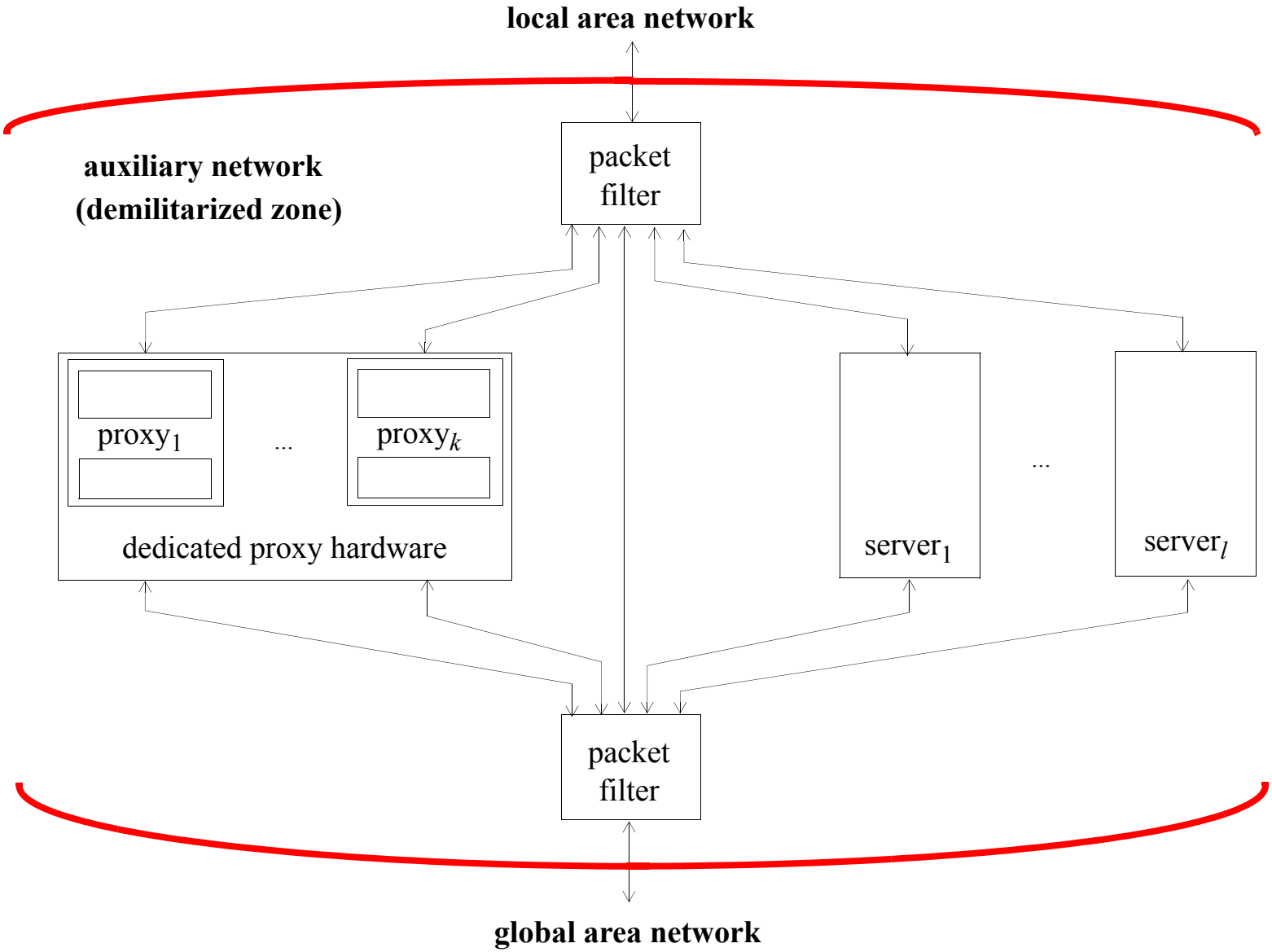
Packet filter

- placed in a layer corresponding to the *network* and *transport* layers
- inspects (statelessly) only the header of each single intercepted packet
- based on a policy as a linear list of rules:
  - `if` *event* `then` *action* — form of the event–action rules
  - *event* — expressed in terms of values of header fields
  - *action* — demands a *forwarding*, a *blocking*, some other simple option
- scans, for each packet considered, the linear list from the beginning until the first satisfied event is found; then this rule "fires" by performing the indicated action
- requires a careful arrangement of the rule ordering, to take care of the linear *first-fit search*

Proxy

- placed in a layer corresponding to the *session* and *application* layers
- simulates the complete services of a higher-layer communication *protocol*
- deals (statefully), depending on the simulated protocol,
  with both the headers and the contents of *sequences* of packets
- divided into two strictly separated parts,
  each of which deals with the functionality of one side of the borderline
- operates an inner part (for LAN: confining the subjects inside):
  - inspects the outgoing packet stream (from inside the LAN)
  - if permitted, and possibly modified,
    forwards the packets to the outer part
- operates an outer part (for LAN: supporting security interests of partners):
  - inspects the stream received from inner part
  - if permitted, and possibly modified,
    forwards the packets to the partners outside
- works correspondingly for incoming packet stream
  from outside (into the LAN)

Generic example of a LAN borderline firewall



10 Intrusion Detection and Reaction

Ideals of control and monitoring

- a security policy specifies exactly
  the wanted permissions and prohibitions
- administrators correctly and completely declare the policy
- subsequently, the policy is fully represented within the computing system
- the control and monitoring component can never be bypassed
- this component enforces the policy without any exception
- as a result, all participants are expected
  to be confined to employing the computing system

Shortcomings in reality

- the security policy is left imprecise or incomplete
- the declaration language is not expressive enough
- the internal representation contains flaws
- the enforcement does not cover all access requests
- administrators or users disable some control facilities for efficiency reasons
- intruders find a way to circumvent the control and monitoring component

Some intricate difficulties

- in general, as indicated by undecidability results,
  control privileges and information flow requirements
  are computationally difficult to manage

- for the sake of efficiency, information flow requirements can
  only be roughly approximated by access rights

- a user might need some set of specific permissions for his obligations,
  but not all possible combinations of the permissions are seen to be acceptable

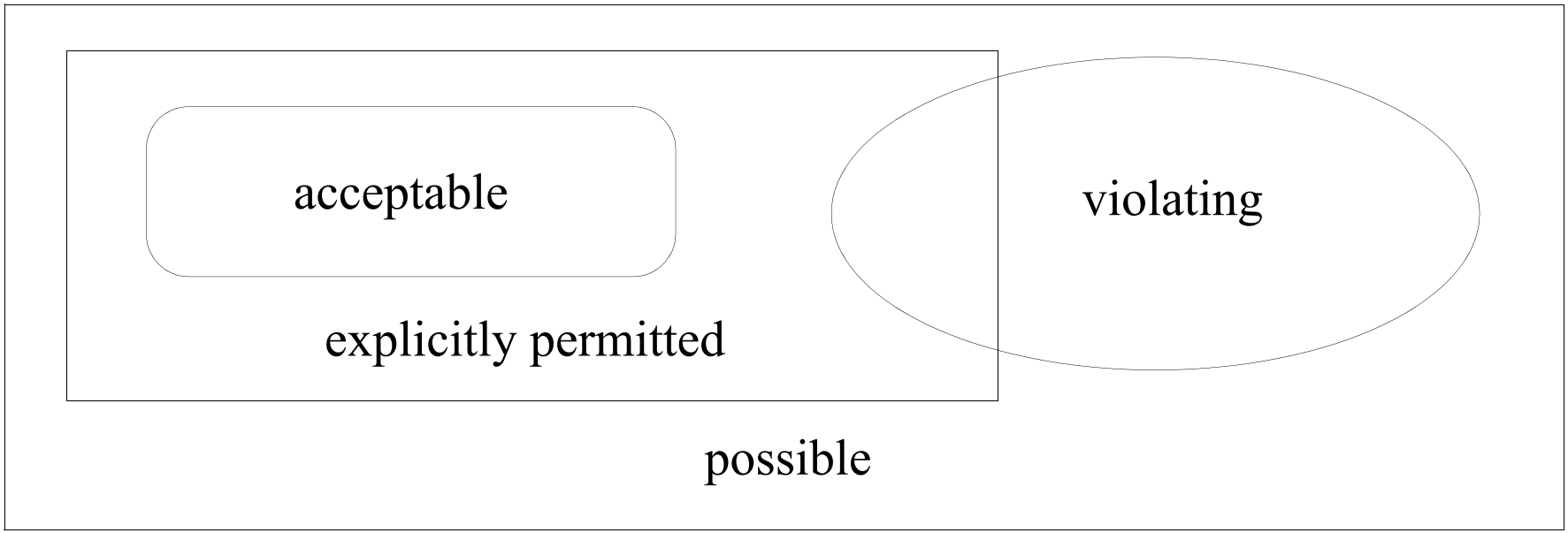
- a user might exercise his permissions excessively and
  thereby exhaust the resources of the computing system

- a user might exploit hidden operational options
  that have never been considered for acceptable usage

Additional protection mechanisms

- the generic model of local control and monitoring provides a *useful basis*
- access requests are intercepted and thus can be documented persistently
  in the *knowledge base* on the *usage history*: extend to
  logging further *useful data* about computing activities,
  including data that is only *indirectly* related to a malicious user's requests
- individual access requests are decided: extend to
  - *auditing* and *analyzing* request sequences/other recorded activities;
  - searching for *intrusions* (patterns of unexpected or unwanted behavior)
  - *reacting* as far as is possible or convenient
- clearly, such additional secondary mechanisms
  cannot achieve perfection either;
  they should be designed to work *complementarily*,
  aiming at narrowing the gap left by the primary mechanisms

Classifying behaviors or states



- *possible* behaviors:
  captures all operational options of the computing system considered
- *explicitly permitted* behaviors:
  enforced according to privileges granted
- (*semantically*) *acceptable* behaviors:
  described by the “intended usage” of the system/security defense policy
- *violating* behaviors:
  described by the “unwanted usage” of the system/security defense policy

Classification and monitoring task

- checking
  whether behaviors are remaining within the “acceptable behaviors” or
  whether they are going to approach a “violating behavior”

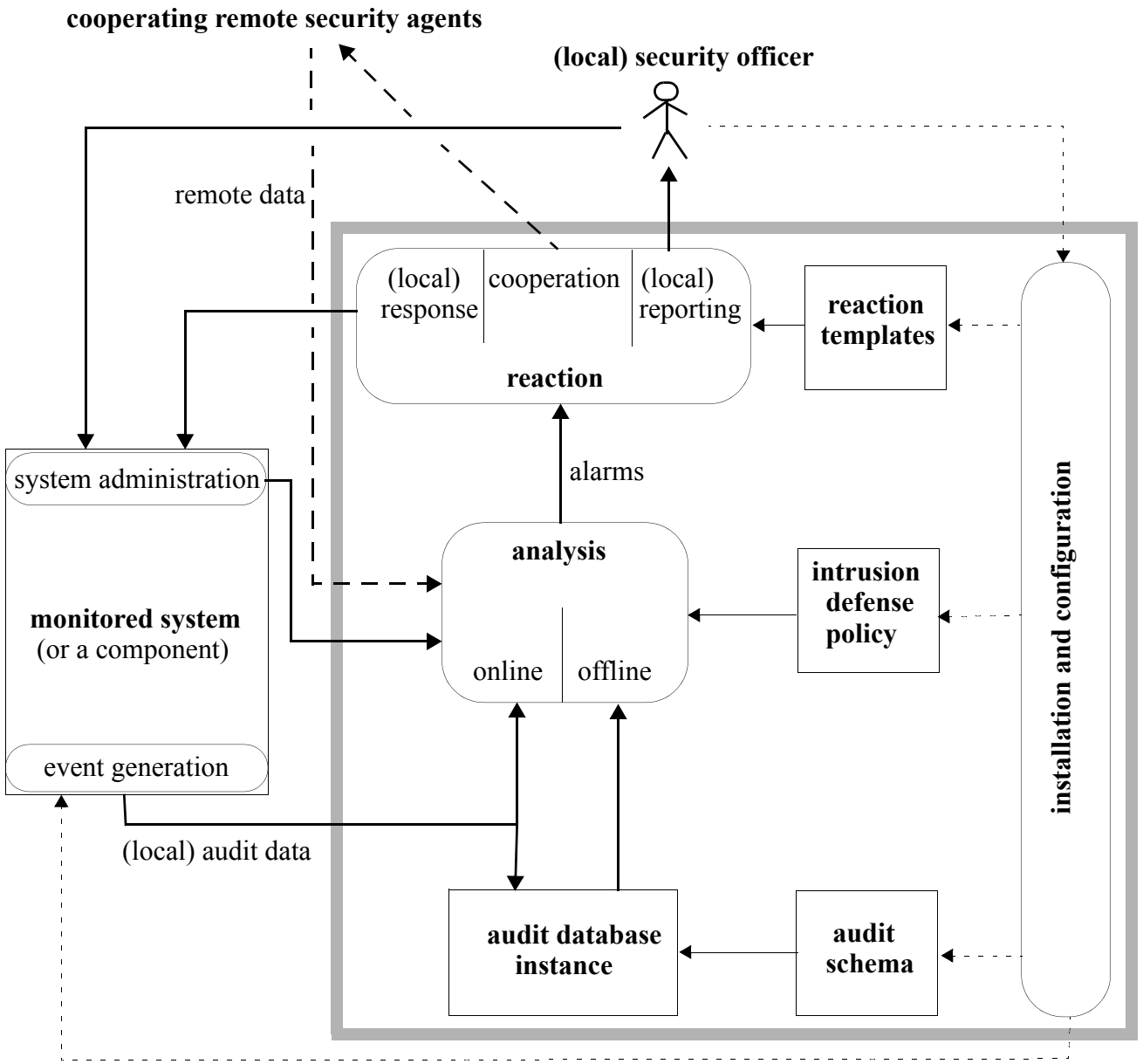
- *separating* “acceptable” behaviors from “violating” ones

- keeping track of histories

- exploring whether an inspected state transition could possibly be
  a *dangerous step towards* reaching a “violating behavior”

- investigating whether a “violating behavior” has *already been reached*

A simple model



Basic components

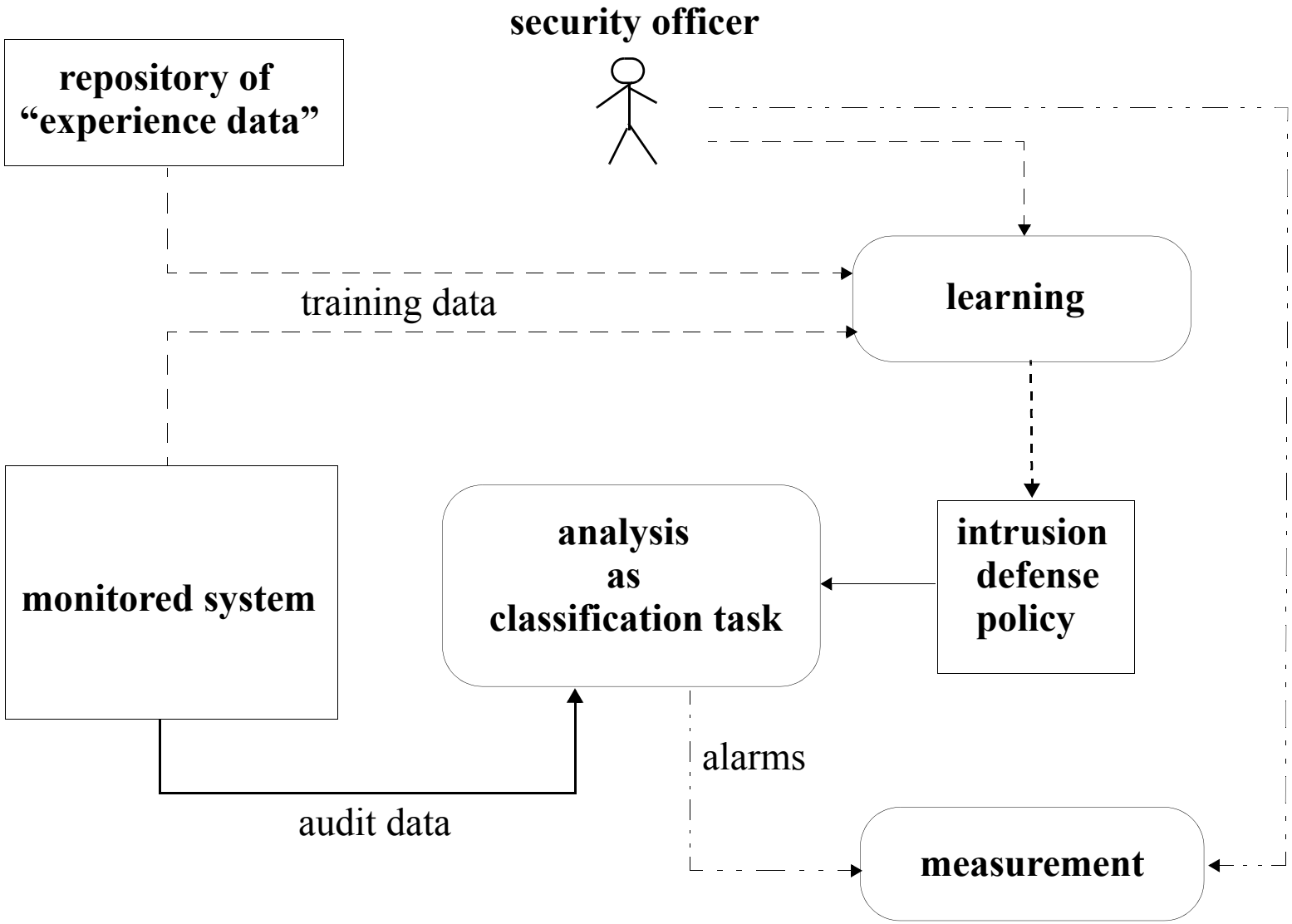
- ***event generation***
  appropriately implanted in the *monitored computing system*,
  delivers local *audit data* to the monitoring system

- ***audit database instance***
  constitutes the intermediately stored audit data gathered for offline analysis

- ***analysis***
  - directly inspects the currently delivered audit data in an *online* mode, or
  - examines a larger amount of audit data *offline*
  - raises *alarms* if suspicious behaviors or states are detected

- ***reaction***
  deals with alarms in basically three ways:
  - purely algorithmically generating a *local response*
    that intervenes in the monitored system
  - *local reporting* to a human *security officer*
  - *sending* appropriate messages to *cooperating remote security agents*

Learning, operation and measurement for a policy



Effectiveness of an analysis component: four possibilities

- component raises no alarm (it classifies the behavior as “acceptable”)
  and
  the “real status” of the behavior is indeed acceptable
- component raises an alarm (it classifies the behavior as “violating”)
  and
  the “real status” of the behavior is indeed violating
- component raises an alarm (it classifies the behavior as “violating”),
  but
  the “real status” of the behavior is actually acceptable:
  – component raises a *false* alarm
  – classification result is said to be a *false positive*
- component raises no alarm (it classifies the behavior as “acceptable”),
  but
  the “real status” of the behavior is actually violating:
  – component fails to generate a *correct* alarm
  – classification result is said to be a *false negative*

Signature-based approach: outline

- contributes to representing *violating behaviors* and constructing a corresponding recognizer
- long-term observation and evaluation of violating behaviors have led to a large collection of samples of known attacks
- a *signature* is a formal representation of a known *attack pattern*, preferably including its already seen or merely anticipated variations, in terms of generic *events*
- instances of events are recognized by the *event generation* and reported as *audit data*

Signature-based approach: overly simplified case

- a *signature* σ is given as a finite time-ordered sequence of abstract events, taken from a finite event space Σ: $\sigma \in \Sigma^*$

- the *event space* Σ is determined by the layer of the event generation, e.g.:
  - operating system: *system calls* to the kernel
  - network system: *packet moves*
  - application, system: *method invocations*

- depending on the actual location, intrusion detection systems are sometimes classified as *host-based* or *network-based*

- inputs of *analysis component*:
  - $\sigma \in \Sigma^*$: fully known signature, the intrusion defense policy
  - $\beta \in \Sigma^\infty$: an eventwise supplied behavior, (ongoing) recorded activities

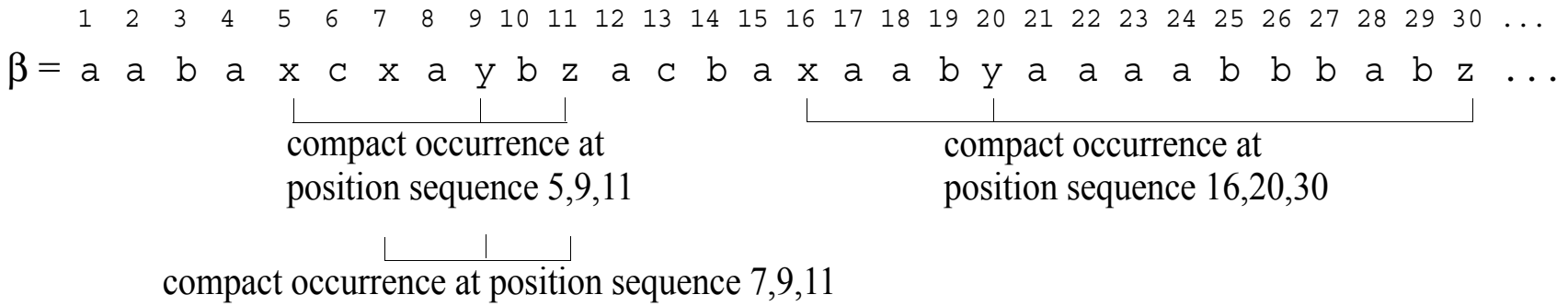
- basic *classification task*:
  determine whether and where
  "the signature σ *compactly occurs* in the behavior β", i.e.,
  find *all* position sequences for β that give the signature σ
  such that each prefix cannot be completed earlier
  (or some similar property holds)

- analysis component must provide a corresponding *recognizer*,
  which should raise an alarm for *each* such compact occurrence of σ in β

- abstract example:

signature: σ = x y z

supplied behavior:



Analysis component: some more sophisticated features

- a compact occurrence of the signature σ can be spread widely:
  while checking whether "σ compactly occurs in β",
  the recognizer must memorize and handle
  each detected occurrence of a prefix of σ in β
  until the prefix has been completed

- one would like to "forget" non-completed prefixes after a while:
  - declare explicit *escape conditions*
  - employ a *sliding window* of some appropriate length l for the behavior β

- a *parameterized event* $e[\ldots, A_i : v_i, \ldots]$ might consist of
  - an event type e
  - a list of specific attribute–value pairs $A_i : v_i$

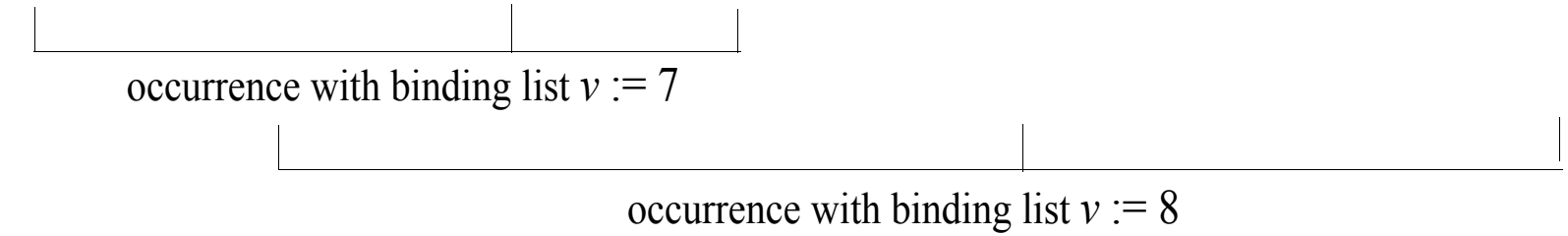
- a *parameterized signature* would be a sequence of parameterized events,
  where some or all values might be replaced by variables:
  while searching for occurrences of the signature in the supplied behavior,
  the values in the signature have to *match* the audited values,
  whereas the variables in the signature are *bound* to audited values

- thus each detected occurrence of a prefix of the parameterized signature is linked to a *binding list* for variables:
  once a variable is bound for a detected prefix,
  the binding also applies to the tail of the signature
- the recognizer must maintain a partially instantiated signature instance for each detected occurrence of a prefix, e.g.:

signature: `x[ID:v] y[ID:v] z[AR:loc]`

supplied behavior:

`x[ID:7]...x[ID:8]...y[ID:7]...z[AR:loc]...y[ID:8]...z[LOC:net]...z[AR:loc]`



- to capture variations of an attack,
  several closely related event sequences might be represented concisely
  as a directed acyclic graph (dag) built from events:
  the recognizer has to search for a compact occurrence
  for any path from some start event to some end event
  within the supplied behavior

- “violating” behavior is described by hundreds of known attacks,
  and thus by a large number of signatures:
  the analysis component has to handle them in parallel

Signature-based approach: basic steps:

- ***learning phase*:** the administrator, assisted by a tool
  - models the known attacks by an intrusion defense policy, specified as a set of parameterized dag-like signatures
  - transforms the specified policy into an integrated collection of recognizers

- ***operation phase*:** recognizers
  - instantiate the given signatures according to the prefixes and their bindings for variables, as detected in the supplied behavior within a sliding window
  - raise an alarm whenever an instantiation has been completed

- ***measurement phase*:** the administrator
  - revises or refines the policy
  - enlarging the length of the sliding window
  - optimizes the recognizers (diminishing/adapting dynamically the length of the sliding window, ...)

Anomaly-based approach: outline

- contributes to representing *acceptable behaviors* and constructing a corresponding recognizer for non-acceptable behaviors

- with some precautions, a large collection $N \subset \Sigma^*$ of actual behaviors, i.e., sufficiently long event sequences generated as audit data in the past, is supposed to constitute a representative sample of "acceptable" behaviors

- a recognizer is constructed, trained to
  - let each collected behavior $\sigma \in N$ pass (seen as supposedly *normal*)
  - let sufficiently similar behaviors pass as well (still seen as supposedly *normal*)
  - raise an alarm for all other behaviors (seen as *anomalous*)

Anomaly-based approach: basic steps

- ***learning phase*:** the administrator
  - gathers a sample set *N* of supposedly normal behaviors
  - selects a length *l* of a *sliding window* on the behaviors
  - employs a suitable tool for *machine learning*
    to construct an efficient finite-automaton-like recognizer
    for anomalous parts of behaviors

- ***operation phase*:** the recognizer
  - searches for anomalous parts in the supplied behavior
    within the sliding window
  - raises an alarm whenever such a part has been detected

- ***measurement phase*:** the administrator
  - adapts the sample set *N* or enlarges the length of the sliding window
  - reconstructs the recognizer
  - optimizes or even smooths the recognizer
    (letting additional behaviors pass,
    diminishing the length of the sliding window, ...)

Cooperation

- *normalization*
  maps local alarms to a common format with common semantics
- *fusion*
  discards obvious duplicate alarms generated by different sites
- *verification*
  identifies irrelevant alarms and false positive alarms
- *thread reconstruction*
  gathers together alarms describing attacks with same origin and/or target
- *session reconstruction*
  correlates alarms that describe events on the network and in a host
- *focus recognition*
  integrates alarms describing attacks with many targets and/or sources
- *multistep correlation*
  combines alarms suspected to constitute a complex attack
- *impact analysis* and *alarm prioritization*
  determine the suspected effect of an attack to prioritize the respective alarm

Part IV

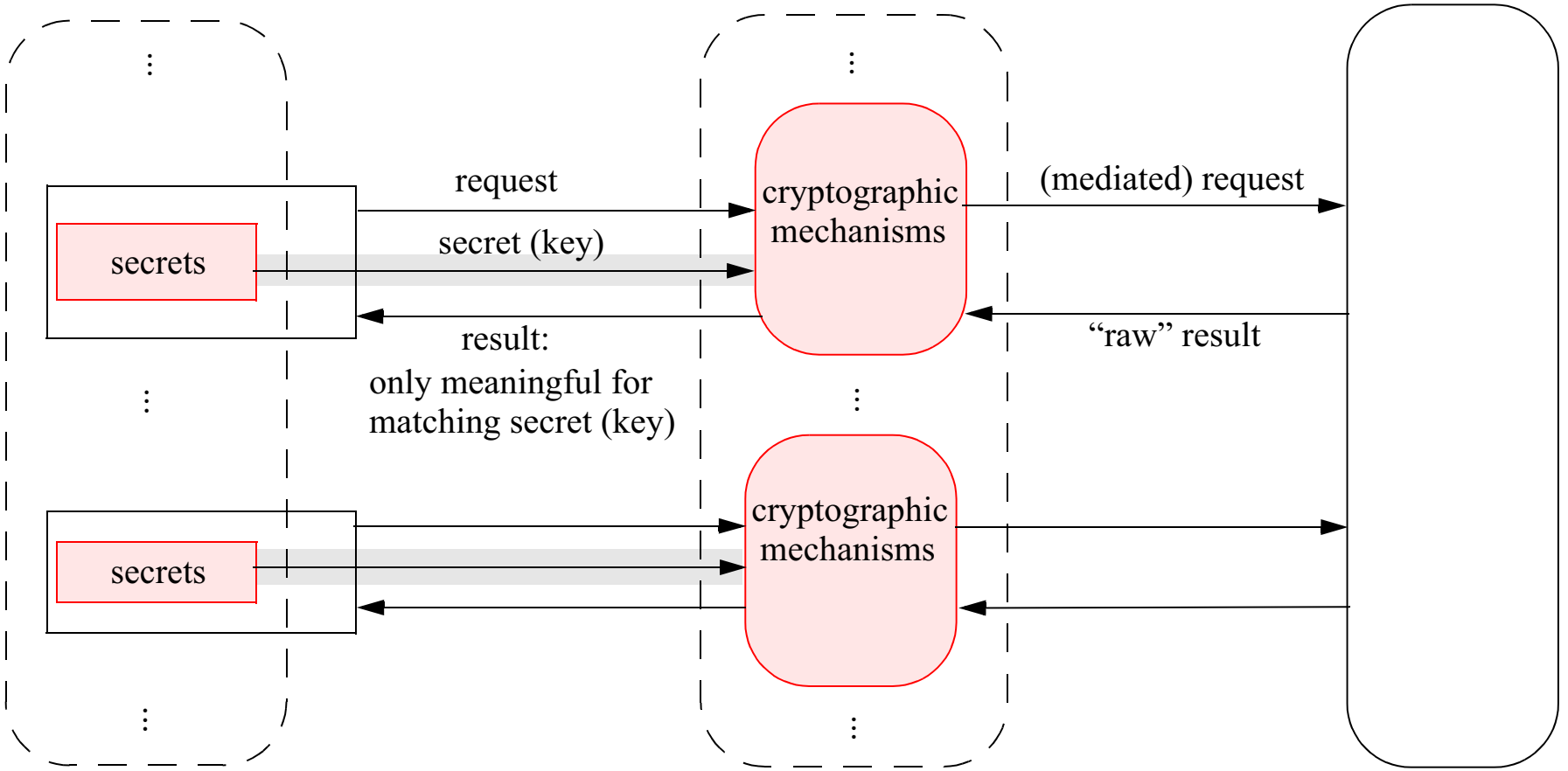
Cryptography

11 Fundamentals of Cryptography

Cryptography

conceptually: (cryptographic)
knowledge base on permissions
(and prohibitions)

conceptually: (cryptographic)
control component



participating subjects

- generate, store and employ secrets
- exploit physical isolation
  (indicated by the gray areas)

controlled objects

Cryptography

- is usually closely intertwined with control and monitoring
- binds a successful and meaningful execution of an operation or interaction to providing a suitable *secret key* as input
- achieves *virtual isolation* between participants: participants that share a cryptographic key are virtually isolated from those that do not
- enables *cooperation in the presence of threats* based on limited trust: participants that autonomously generate and secretly keep appropriate cryptographic keys can enforce their security *interests* by themselves

Basic cryptographic blocks

- encryption
- authentication
- anonymization
- randomness and pseudorandomness
- one-way hash functions
- timestamps

Encryption: functionality

- the sender *S* transforms the original bit string *m* to be transmitted into another bit string m°
  such that *only* the designated receiver *R* (and possibly the sender) is enabled to recover the original bit string

- (probabilistic) *key generation* algorithm *Gen* (one parameter and one result):
  - l — *security parameter* (key length, ...)
  - (ek_R , dk_R) — matching *key pair*

- (probabilistic) *encryption* algorithm *Enc* (two parameters and one result):
  - ek_R — *encryption key*
  - m — *plaintext* (original message)
  - $m^\circ = Enc(ek_R, m)$ — *ciphertext* (transformed bit string)

- (probabilistic) *decryption* algorithm *Dec* (two parameters and one result):
  - dk_R — *decryption key*
  - m° — *ciphertext*
  - $m^{\circ\circ} = Dec(dk_R, m^\circ)$ — (hopefully) recovered plaintext

Encryption: correctness property

- encryption algorithm *Enc* and decryption algorithm *Dec* should be inverse whenever a *matching key pair* (ek_R, dk_R) generated by *Gen* has been employed:

  for all plaintexts m, $Dec(dk_R, Enc(ek_R, m)) = m$

Encryption: secrecy property

- ***naive version***
  for all plaintexts m, without a knowledge of the decryption key dk_R,
  m cannot be "determined" from the ciphertext $m°$

- **(*informal*) *semantic version***
  an unauthorized observer of a ciphertext cannot infer
  anything new about the corresponding plaintext, i.e.,

  for all plaintexts m, without a knowledge of the decryption key dk_R,
  any *property* of m that can be "determined" from the ciphertext $m°$
  could also be "determined" without knowing $m°$ at all

- **(*informal*) *operational version***
  an unauthorized observer of ciphertexts
  cannot separate apart any pair of ciphertexts, and thus

  cannot solve the problem of
  assigning a specific plaintext to a ciphertext

Operational secrecy as indistinguishability

for a *probabilistic* setting, considering
sequences of plaintexts and of matching key pairs
of *increasing* length (taken as a *security parameter*),
we have *indistinguishability of ciphertexts*:

for any pair of plaintext sequences
$(m_1', m_1'', m_1''', \ldots)$
and
$(m_2', m_2'', m_2''', \ldots)$,

without a knowledge of the sequence of decryption keys employed,

the resulting sequences of ciphertexts are
"computationally indistinguishable"

Basic assumptions

- approved algorithms *Gen* and *Dec* and *Enc* are publicly known
- decryption keys are strictly kept secret
- given approved algorithms and seen from the perspective of the endusers, enforcing the *confidentiality* of messages by encryption basically relies *only* on
  - selecting appropriate keys (as determined by the security parameter)
  - actually hiding the decryption keys

Relationship between the encryption key and the decryption key

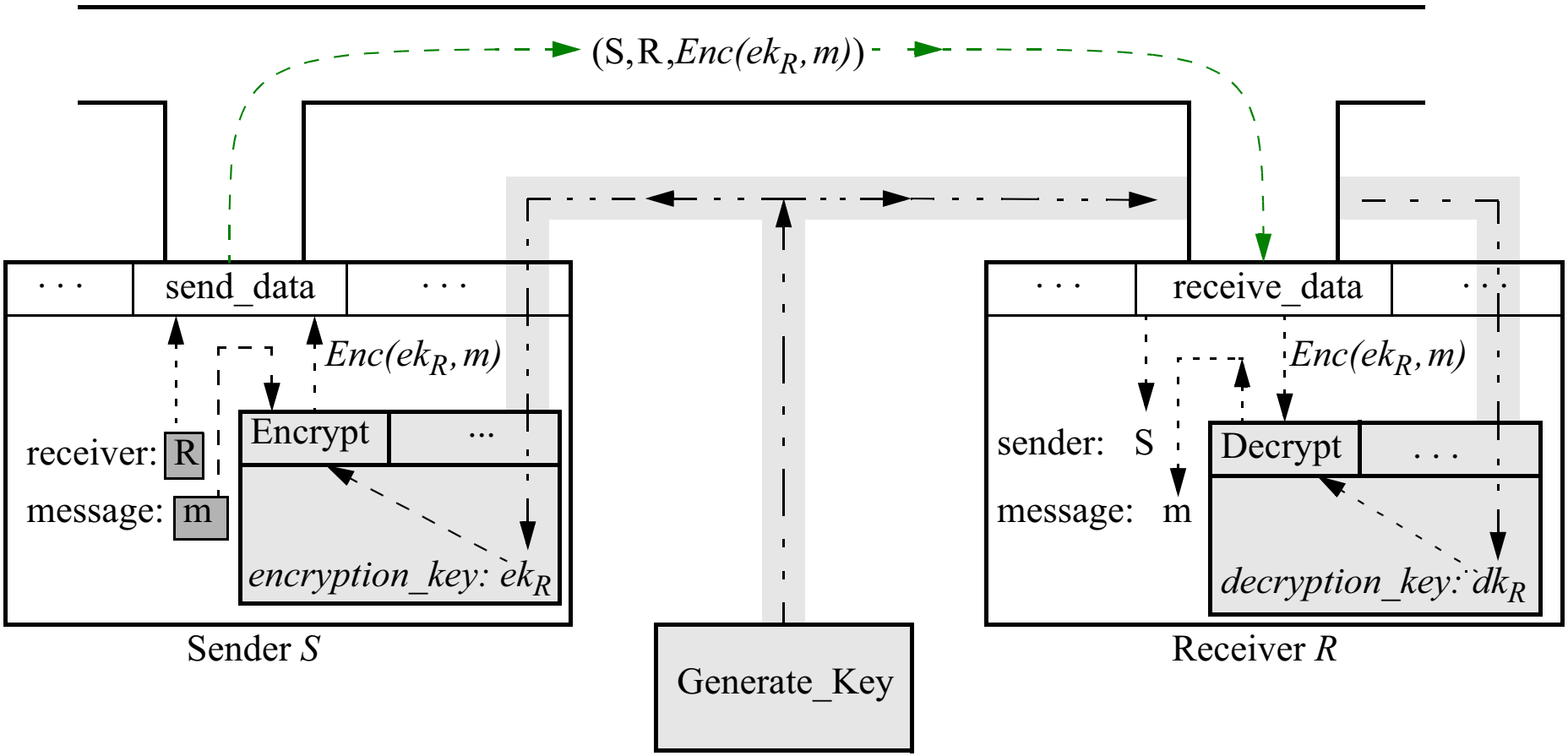
- ***symmetric* (or *secret-key*) mechanism:**
  - the encryption key is (basically) *equal* to the decryption key

- ***asymmetric* (or *public-key*) mechanism:**
  - the encryption key is essentially *different* from the decryption key
  - an additional *secrecy property* (*naive* version) is required:

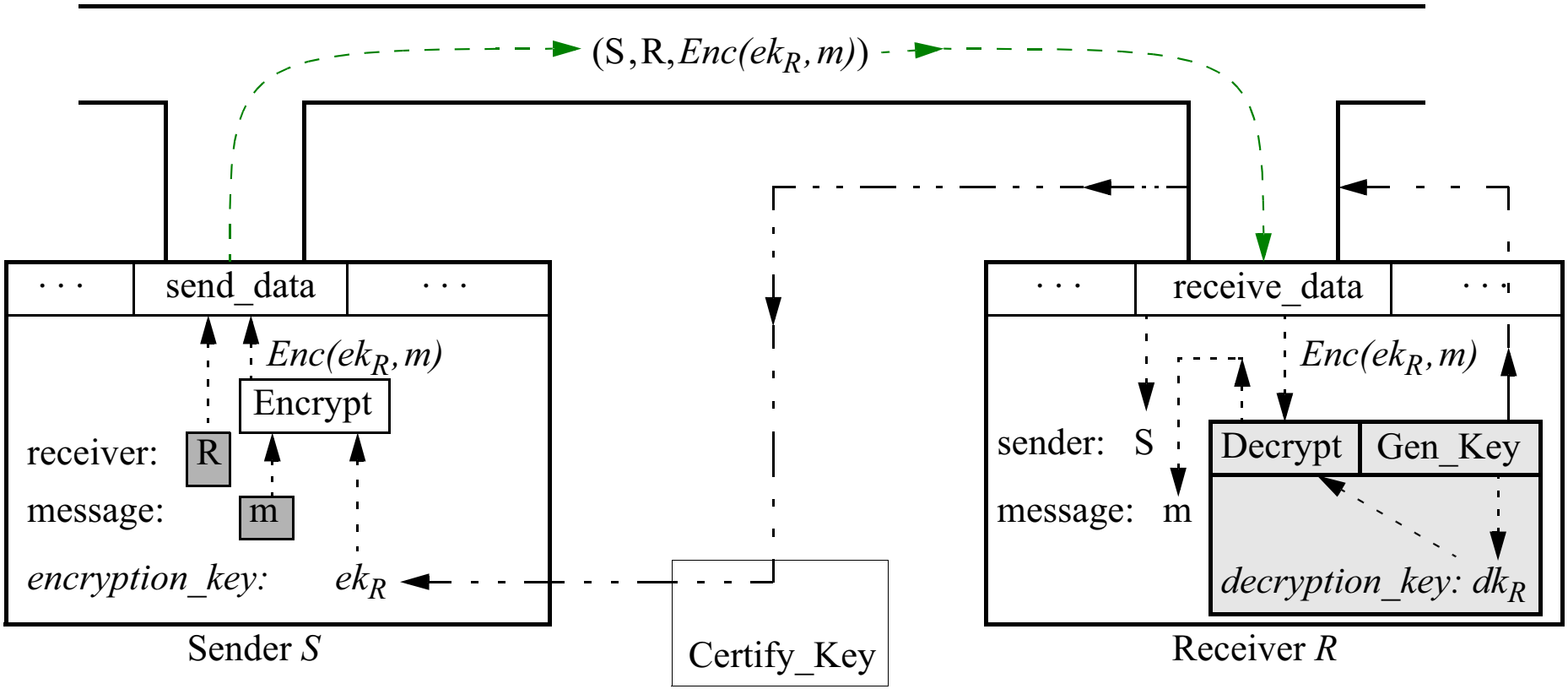
    the (private) decryption key dk_R

    cannot be "determined" from the (public) encryption key ek_R

Symmetric encryption



Asymmetric encryption



Symmetric and asymmetric encryption mechanisms

Feature	Symmetric	Asymmetric
generating and *distributing* keys	both partners are *equally* involved	designated *receiver* has a *distinguished* role
protection requirements	key generation/ communication of the secret key and storage of the secret key must be protected on *both sides*	key generation and storage of the private key must be protected on the side of the *receiver only*
contributions of the *trusted third parties*	generate and distribute secret keys	certify public keys

Authentication: basic approach

designated sender *S*

- prepares for transmitting a bit string m as a message
  by computing another bit string $red_{S,m}$
  as a *cryptographic piece of evidence*
  (*cryptographic exhibit* or *cryptographic check redundancy*)
- forwards the compound $(S, m, red_{S,m})$
  - S sender identification
  - m original bit string
  - $red_{S,m}$ computed bit string

receiver

- receives such a compound of the form $(S°, m°, red_{S,m}°)$
- checks whether the message part originates from the claimed sender without modification by inspecting the included cryptographic exhibit (must depend on both the designated sender and the message)
- either accepts (as authentic) or rejects the received message

Authentication: functionality

- (probabilistic) *key generation* algorithm *Gen* (one parameter and one result):
  - l — *security parameter* (key length, ...)
  - (tk_S , ak_S) — matching *key pair*

- (probabilistic) *authentication* algorithm *Aut* (two parameters and one result):
  - ak_S — *authentication key*
  - m — message
  - $red_{S,m} = Aut(ak_S, m)$ — *cryptographic exhibit*

- (probabilistic) Boolean-valued *authenticity verification* algorithm *Test* (three parameters and Boolean result):
  - tk_S — *test key/verification key*
  - m — received message
  - red — cryptographic exhibit

Authentication: (weak) correctness property

- authentication algorithm *Aut* and authenticity verification algorithm *Test* should be complementary
  whenever a *matching key pair* (tk_S, ak_S) generated by *Gen*
  has been employed:

  for all messages m, $Test(tk_S, m, Aut(ak_S, m)) = true$

Authentication: unforgeability

- **(*naive*) *unforgeability property***
  for all messages m,
  without a knowledge of the authentication key ak_S,
  one cannot "determine" a bit string *red* such that $Test(tk_S, m, red) = true$

- **(*naive*) *strong correctness property*,**
  complemented by a ***weak unforgeability property***

  for all messages m and for all bit strings *red*,
  $Test(tk_S, m, red) = true$ iff $red = Aut(ak_S, m)$

  and

  without a knowledge of the authentication key ak_S,
  one cannot "determine" this solely accepted cryptographic exhibit

Basic assumptions

- approved algorithms *Gen*, *Aut* and *Test* are publicly known
- authentication keys are strictly kept secret
- given approved algorithms and seen from the perspective of the endusers, enforcing the *integrity* and *authenticity* of messages (in the sense of detection of violations) by authentication basically relies *only* on
  - selecting appropriate keys (as determined by the security parameter)
  - actually hiding the authentication keys

Relationship between the test key and the authentication key

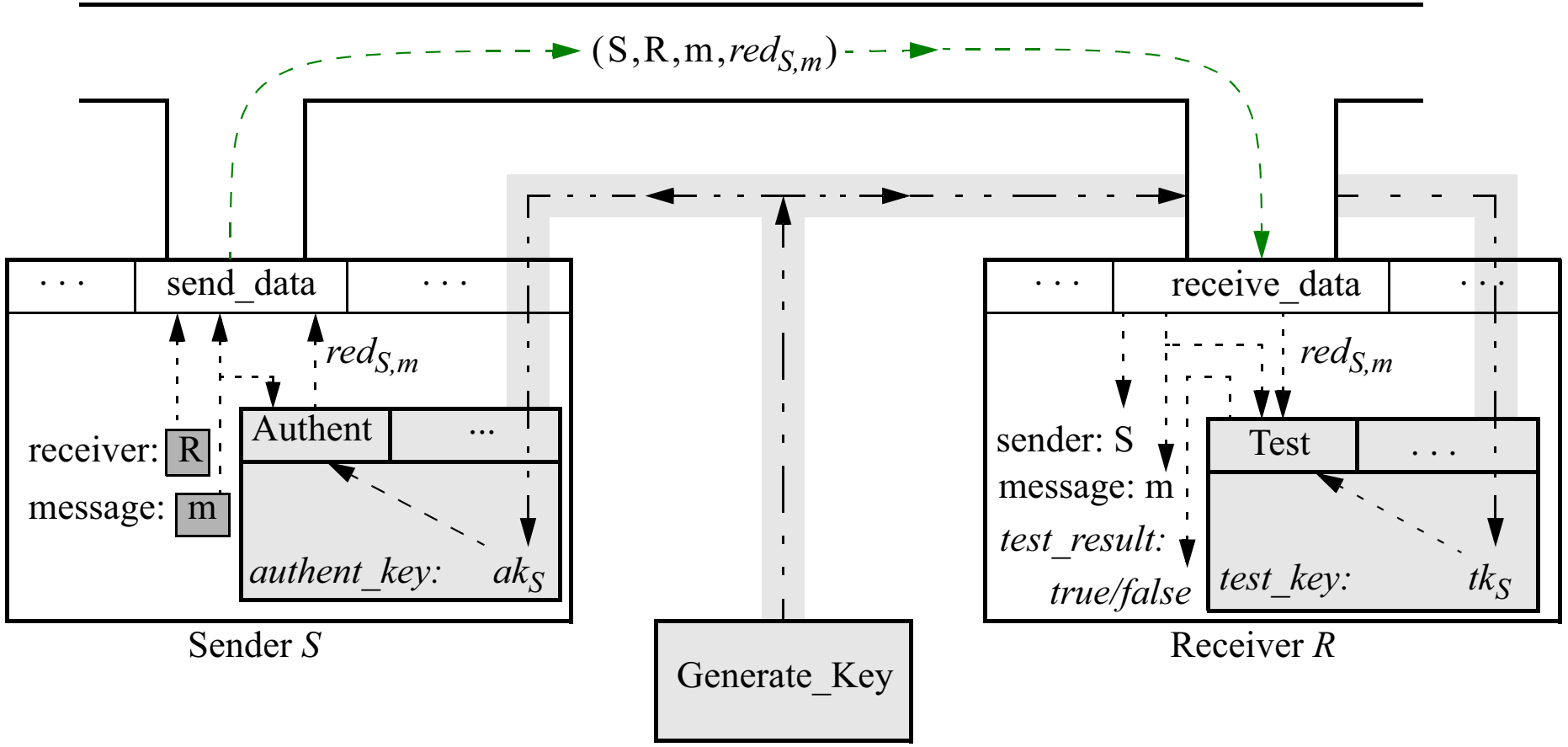
- ***symmetric* (or *secret-key*) mechanism:**
  - the test key is (basically) *equal* to the authentication key

- ***asymmetric* (or *public-key*) mechanism:**
  - the test key is essentially *different* from the authentication key
  - an additional *secrecy property* (*naive* version) is required:

    the (private) authentication key ak_S

    cannot be "determined" from the (public) test key tk_S

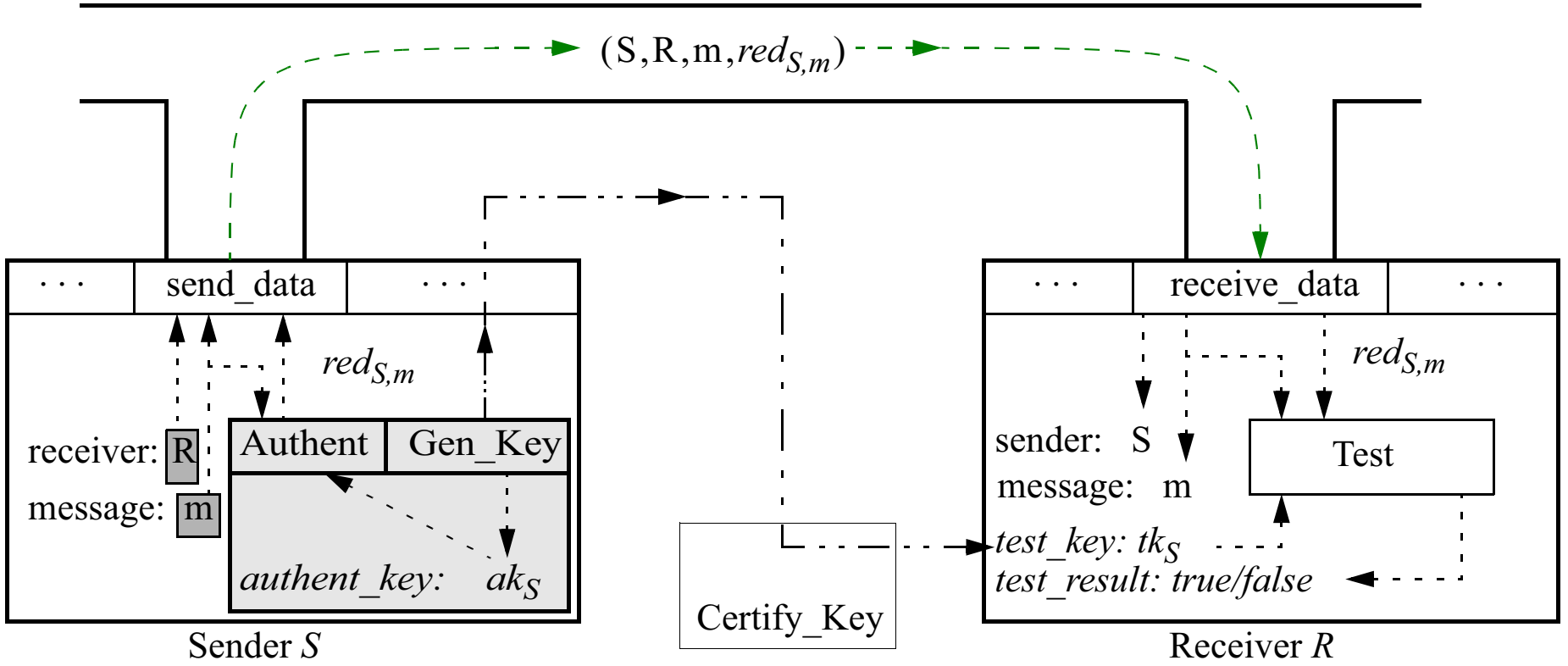
Symmetric authentication



authenticity verification algorithm:

- recompute the cryptographic exhibit for the received message
- compare the result with the received exhibit
- the verification is seen as *successful* iff both exhibits are equal

Asymmetric authentication (digital signing)



Symmetric and asymmetric authentication mechanisms

Feature	Symmetric	Asymmetric
generating and *distributing* keys	both partners are *equally* involved	designated *sender* has a *distinguished* role
protection requirements	key generation/ communication of the secret key and storage of the secret key must be protected on both sides	key generation and storage of the private key must be protected on the side of the *sender only*
contributions of the *trusted third parties*	generate and distribute secret keys	certify public keys
non-repudiation/ digital signatures	no	yes

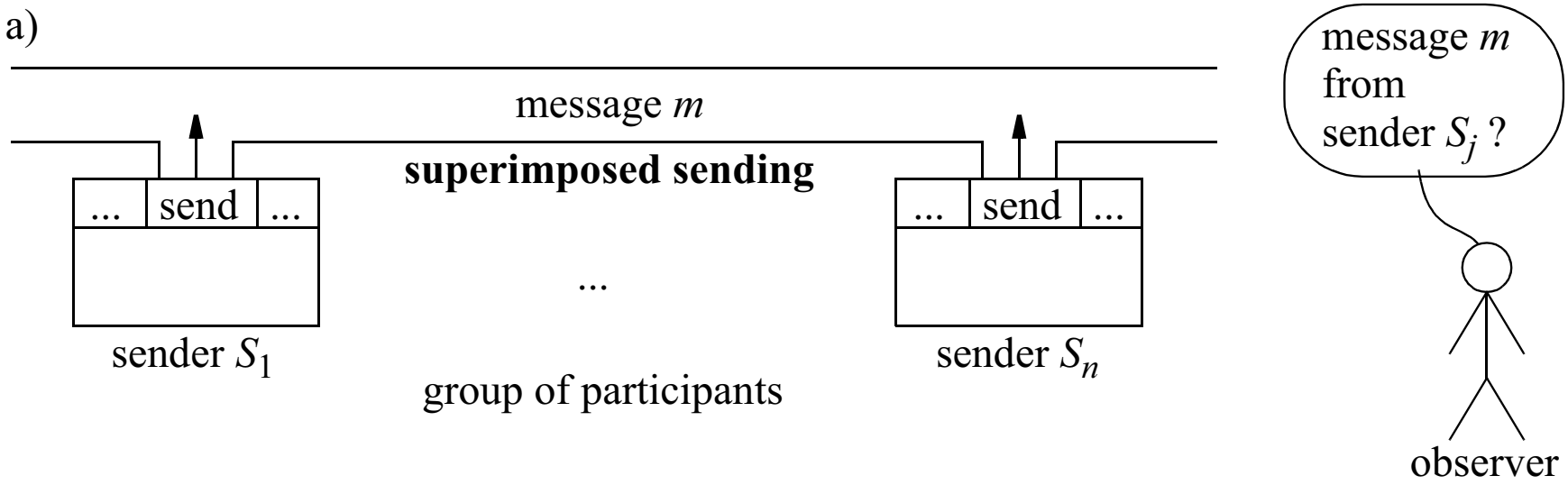
Anonymization

- the interest in *anonymity* or, more generally, in *non-observability* can be seen as strengthened forms of (message) *confidentiality*:
  - not only the message itself should be kept secret
  - but also the full *activity* of a message transmission

- from the point of view of an observer who is not designated to learn about an activity or a sequence of activities:
  any actually occurring activity
  is *indistinguishable* from
  any other activity in a preferably large activity domain
  from which the actually occurring activity has been selected

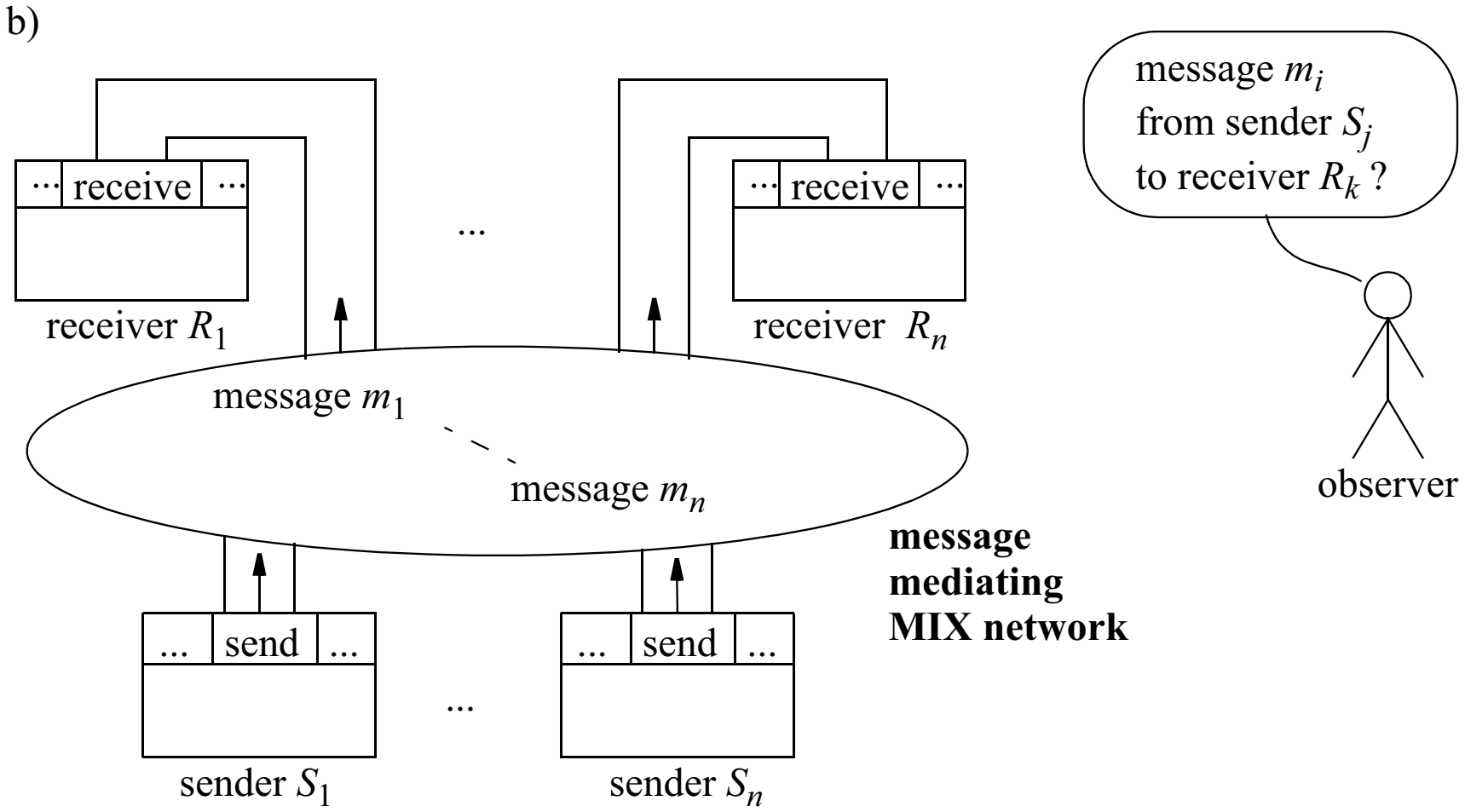
- the actual activity is *indistinguishably hidden* in a preferably large domain of other possibilities, often called an *anonymity class*

Sender anonymity



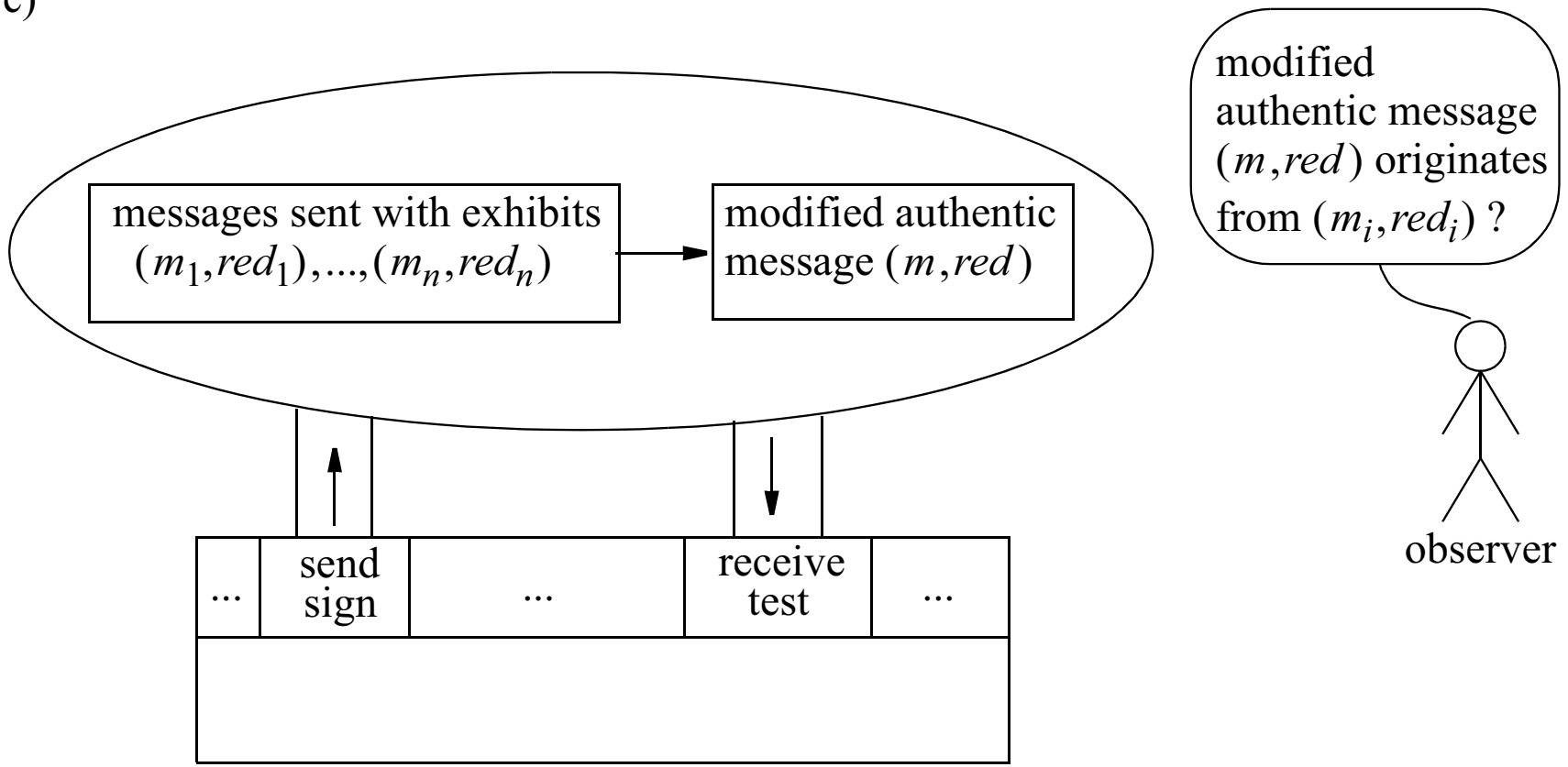
- **activity domain:**
  participants $S_1, \ldots, S_n$ sending and receiving messages
- **anonymity property:**
  by observing an actual message m,
  a non-designated observer cannot "determine" the actual sender S_j
- **mechanism:**
  superimposed sending

Sender–receiver anonymity



Anonymity by unlinkability

c)



“blindly signing” participant

Unlinkability and blind signatures

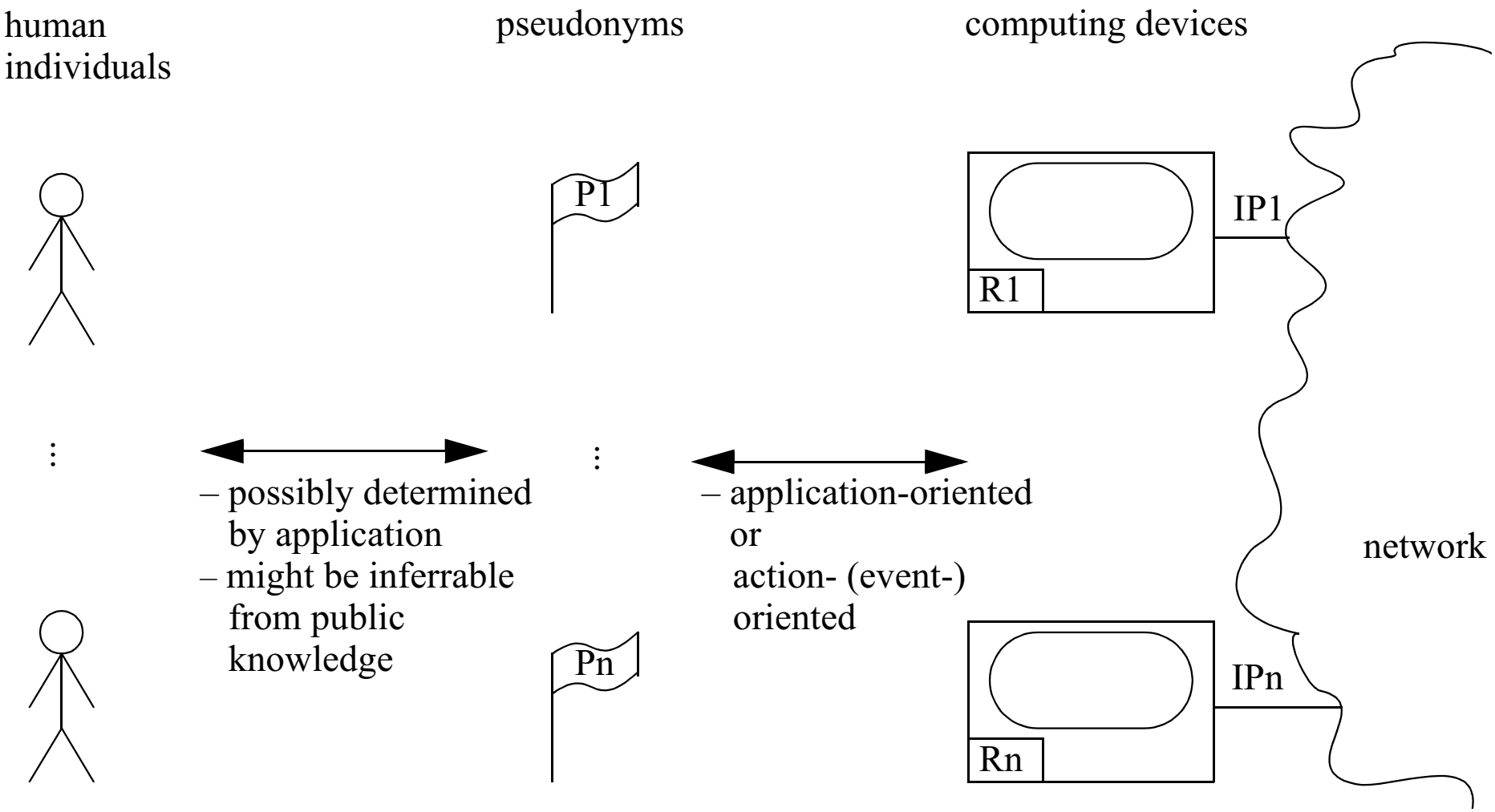
- **activity domain:**
  - one distinguished participant *issues* (sends) digital documents (digitally signed messages) expressing some *obligation* to receivers
  - receivers/holders *present* digital documents as a *credential* (*digital legitimation*) to be redeemed to the distinguished participant
- ***unlinkability property*:**
  knowing the issued documents $\{ (m_1, red_1), \ldots, (m_n, red_n) \}$ and seeing a presented modified document (m, red) with a verified signature *red*, a non-designated observer
  cannot "determine" the link from the presented document to the corresponding issued document
- **mechanism:**
  blind signatures

A classification of pseudonyms

- regarding the *dissemination of knowledge* about the relationship between the pseudonym and the substituted subject, a pseudonym can be seen as
  - *public* (e.g., a phone number of an employee)
  - *confidential* (e.g., a bank account of a citizen)
  - *secret* (also called an *anonym*)

- regarding the intended *potentials for multiple use* and the resulting linkability, there are
  - *subject pseudonyms* for a broad range of activities
  - *role pseudonyms* for specific activities
  - *relationship pseudonyms* for activities addressing specific partners
  - combined *role–relationship pseudonyms* for specific activities addressing specific partners
  - *transaction pseudonyms* (*event pseudonyms*) for *single use* only

Meanings of the notion of “participant” and their relationships



Sufficient randomness

- to achieve the indistinguishability goals of *cryptographic mechanisms*, *sufficient randomness* is needed

- a cryptographic mechanism *superimposes*
  the randomness of a secretly selected key, and possibly further inputs,
  on the returned items of interest such that
  the output items (ciphertexts, exhibits, …)
  again appear to be randomly taken

- making "sufficient randomness" algorithmically available
  is an outstanding open problem in computer science

- in fact, precisely defining the notion of "sufficient randomness"
  has already turned out to be a great challenge
  that has raised various proposals for an answer

Pseudorandom generator

- is a deterministic polynomial-time algorithm

- stretches a seed, a short and supposedly random input,
  into a much larger output sequence
  appearing again to be “sufficiently random”

- delivers outputs that should be *computationally indistinguishable* from
  a family of (ideal) uniformly distributed sequences:
  - there is no probabilistic polynomial-time algorithm
    that can distinguish the algorithmic outputs
    from the abstract ideal sequences
    with a non-negligible probability without knowing the seeds

Guidelines for generating and employing pseudorandom sequences

- use some *physical source* for supplying (supposedly) "truly random" *seeds* of short length
- use a *pseudorandom generator* for stretching a supposedly random input seed into a much larger output sequence appearing again to be "sufficiently random"
- design a cryptographic mechanism (for encryption, authentication, etc.)
  - to take a "truly random" input
  - to superimpose the randomness of this input on the returned items (to be proven to comply with pertinent indistinguishability as well)
- for an actual *implementation*, however, replace the (ideal) "truly random" input by an actually available pseudorandom sequence
- verify a compositionality property of the *indistinguishability properties*, to ensure that the replacement does not affect the quality of the returned items

Goals of random input: examples

- to *generate* a *secret key* for some cryptographic mechanism:
  to designate its holder(s) as distinguished from all other participants

- to employ a random input as a *nonce*:
  to mark a message within some cryptographic protocol as unique and personal

- to *pad* a value from some (too small) domain with a random input:
  to define a modified domain sufficiently large to prevent successful guessing

- to *blind* some data with a random input using a reversible algebraic operation:
  to present that data to somebody else without revealing the actual value

- most generally, to *randomize* some algorithm of a cryptographic mechanism:
  to achieve a wanted indistinguishability property

One-way hash functions

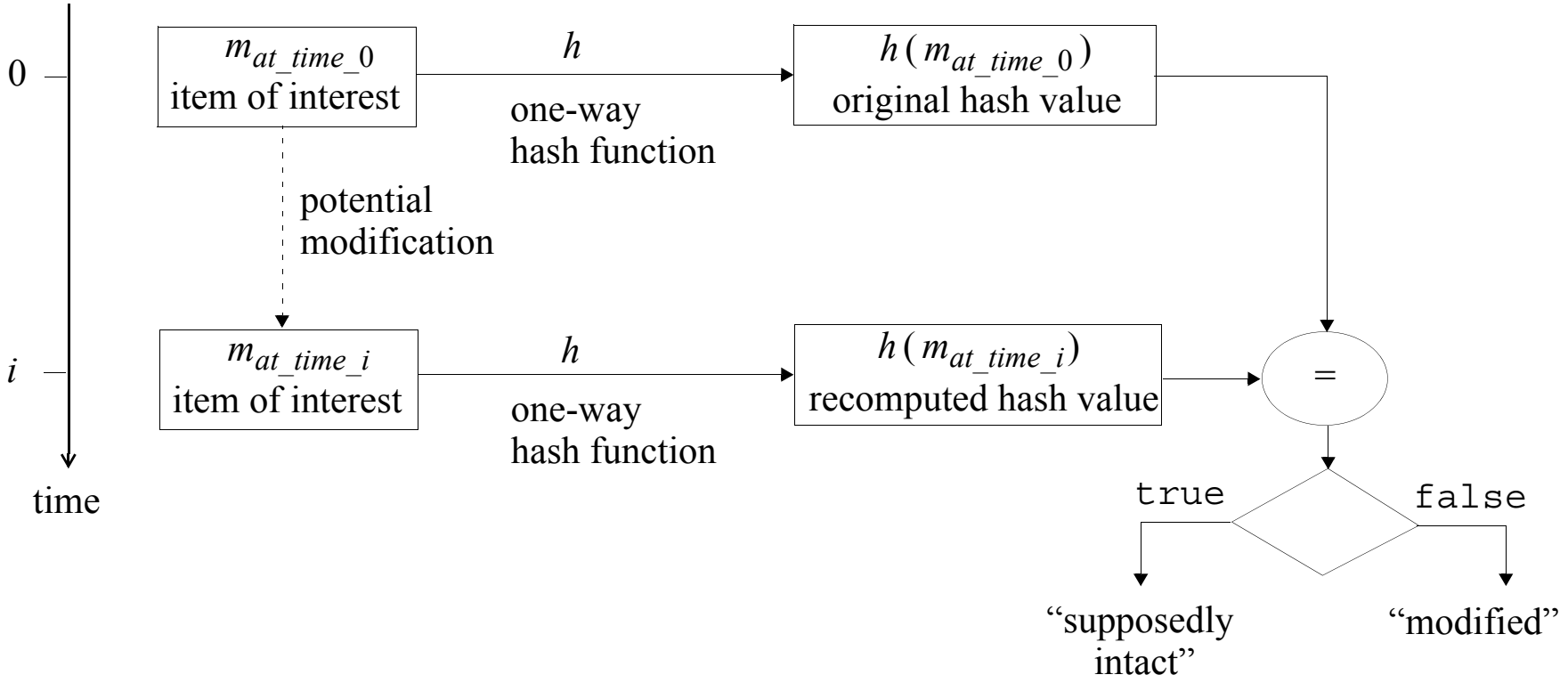
- some item of interest is often represented in a concise, disguised and unforgeable form, called a *fingerprint*, a *digest* or a *hash value*
- *concise*:
  - representation consists of a suitably short bit string of an agreed format
  - a large domain of items is mapped onto a small domain of representations: there must be collisions
- *disguised*:
  a represented item cannot be "determined" from its representation
- *unforgeable*:
  nobody can "determine" a representation of an item
  without a knowledge of that item
- *collision resistant*:
  nobody can "determine" pairs of items that share a representation

Application: representations with fixed short format

- a cryptographic protocol might demand an argument complying with a fixed short format for further processing, but the items of interest might vary or even be of arbitrary length

- example:
  some authentication protocols digitally sign the representations instead of the represented items

Application: enforcing integrity (detection of modification)



- **at time 0:**
  - map the item onto its representation (original hash value)
  - store the item and its representation in different locations
- **at a later time *i*:**
  - compare the retrieved representation (original hash value) with a recomputed representation of the retrieved item (recomputed hash value)

One-way hash functions: functionality and properties

- function h maps any element m of a (large) domain D (might be infinite) onto a bit string of a (short) fixed length l, i.e.,
  onto an element of $\{0,1\}^l$

- an assigned value $h(m)$ is called the *hash value* of m

- the function h must be efficiently computable, i.e.,
  there is an efficient algorithm H that computes $h(m)$ on input of m

- the *inversion* of h must be computationally infeasible, i.e.,
  the following roughly circumscribed *one-way property* is required:
  for all values $z \in \{0,1\}^l$,
  one cannot "determine" a domain element $m \in D$ such that $h(m) = z$

- regarding the inevitable collisions (for large domain and short length),
  the function h must be *collision-resistant*

Weak collision-resistance property

- should protect against a fraud
  where a given message m is exchanged for another one:

- for all domain elements $m \in D$,
  one cannot "determine" a different domain element $m' \in D$
  such that $h(m) = h(m')$

Strong collision-resistance property

- should totally block any attempt at a fraudulent exchange

- one cannot “determine”
  two different domain elements $m \in D$ and $m' \in D$
  such that $h(m) = h(m')$

- equivalent to requiring
  that one cannot “determine”
  an element $m \in D$
  that violates the weak version

Timestamps

- sometimes, *integrity* as *temporal correctness* should be supported

- in a proof of authenticity, the receiver should be able to evaluate
  - not only *who* has formed and sent a message
  - but also *when* these two events happened

- to prevent replay attacks or to achieve related goals,
  before authenticating a message, the sender can include a current *timestamp*

- considering the time span between
  - when the message was *formed* and
  - when it was *received*,

  the receiver can decide whether he is willing
  to accept the message as authentic or not

- all participants involved must share *synchronized clocks*;
  the receiver should take tolerable discrepancies in local times into account

- combined *temporal correctness and unforgeability property* is desired:

  for all messages *m* with an included timestamp *ts*
  and suitably authenticated by the sender,
  from the perspective of a receiver,
  the actual *forming time* of the message
  coincides with the included *timestamp*

- participants might prefer to employ weaker
  but more readily manageable means than timestamps

- if only *relative* forming times are important,
  the sender might include *serial numbers* (instead of timestamps)

- a receiver not willing to rely on synchronized clocks might
  ask a sender to follow a *challenge–response procedure*
  in order to obtain evidence for the freshness of a received message

Quality in terms of attacks

- cryptography aims at enabling participants to autonomously enforce their security interests even in the presence of threats
- a *threat* is instantiated by somebody/something performing a specific *attack*
- attack in theoretical investigations:
  an execution of a polynomially time-bounded probabilistic Turing machine
- attack in more practical investigations:
  exploiting a concrete attacking strategy
- *security requirements*:
  to be specified in terms of attacks
- *evaluating* a cryptographic mechanism:
  includes an analysis of the mechanism's robustness against attacks
- *classification framework* for attacks (on encryption mechanisms):
  here, from the point of view of attackers, describing their options for success

A classification framework for attacks against encryption

- ***kind* of success**
  - *exact*: exact new knowledge
  - *probability-theoretic*: improved probability distribution

- ***extent* of success**
  - *universal*: functional equivalence with decryption algorithm
  - *complete*: gain of secret key
  - *message-selective*: plaintexts of selected ciphertexts
  - *message-existential*: plaintext of some ciphertext

- ***target* of attack**
  - affect *human individuals*
  - exploit *computing system*
  - affect individuals and the system in *coordination*

- ***time* of attack/attacked *part***
  - subvert *overall system*
  - subvert *key generation*
  - subvert *key distribution*
  - exploit *message transmissions*

- ***method* of attack (against message transmissions)**
  - *passive*: observe messages [ciphertext/plaintext pairs]
  - *active*: observe plaintexts [ciphertexts] of chosen ciphertexts [plaintexts]

- ***planning* of active attack**
  - *non-adaptive*: choose statically at the beginning
  - *adaptive*: choose dynamically depending on progress

- ***expectation* of success**
  - *probability-theoretic*: upper bound for success probability
  - *complexity-theoretic*: lower bound for needed resources
  - *combined*: upper bound for success probability with limited resources

Cryptographic security

- a participant designated to hold some secret or private keys
  must be able to secretly generate, store and use these keys;
  best if the participant controls a *personal tamper-resistant computing device*

- secret and private keys and possibly further items have to appear as random,
  and, accordingly, some *source of randomness* should be available;
  best possibility being a *truly random* physical source

- items to appear as random must have *sufficient length*
  to resist attacks based on *exhaustive search and trials*

- some assistance of a *trusted third party* is normally required

- various further *external participants* contribute
  to an application of a cryptographic mechanism;
  assigning *trust* to them should be based on
  open design and *informational assurances*

12 Case Studies: PGP and Kerberos

Pretty Good Privacy (PGP)

- supports participants of a distributed computing system
  in autonomously enforcing their security interests
  (*confidentiality*, *integrity* as *detection of modification*,
  authenticity, *non-repudiation*)

- provides a user-friendly interface
  to *encryption* and *authentication* (*digital signatures*) to be employed
  - explicitly by means of a simple command language
  - transparently embedded into some appropriate application software

- may serve
  - to protect *files* on a local computer
  - to ensure *end-to-end security* in a global environment

- assists participants with the necessary *key management*, including assessment
  - of claims that a public key belongs to a specific partner
  - of the *trust* in the respective issuers of such claims

Basic blocks

- *symmetric encryption* by a *block cipher* (IDEA, Triple-DES, AES, ...), extended into a *stream cipher* with *cipher block chaining* (*CBC*) mode:
  applied to
  - plaintexts (files to be stored or messages to be sent)
  - private asymmetric (decryption or signature) keys

- *asymmetric encryption* (RSA, ElGamal, ...) within *hybrid encryption*:
  applied to secret session keys for symmetric encryption

- *authentication* by *digital signatures* (RSA, ElGamal, ...)

- *one-way hash function* (MD5, ...):
  to generate
  - a message digest from an original message
  - a symmetric key from a passphrase

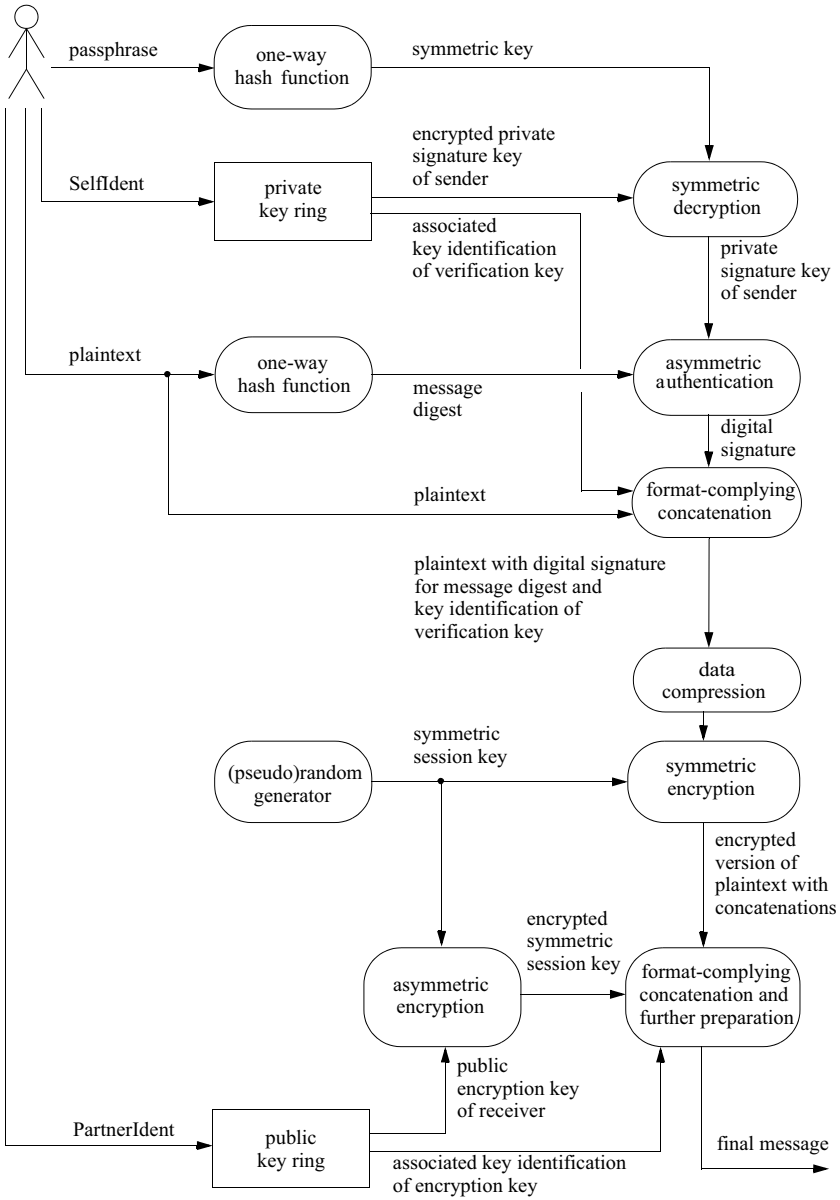
- *random generator* or *pseudorandom generator*:
  for generating symmetric session keys

- *data compression*:
  for reducing the redundancy of plaintexts

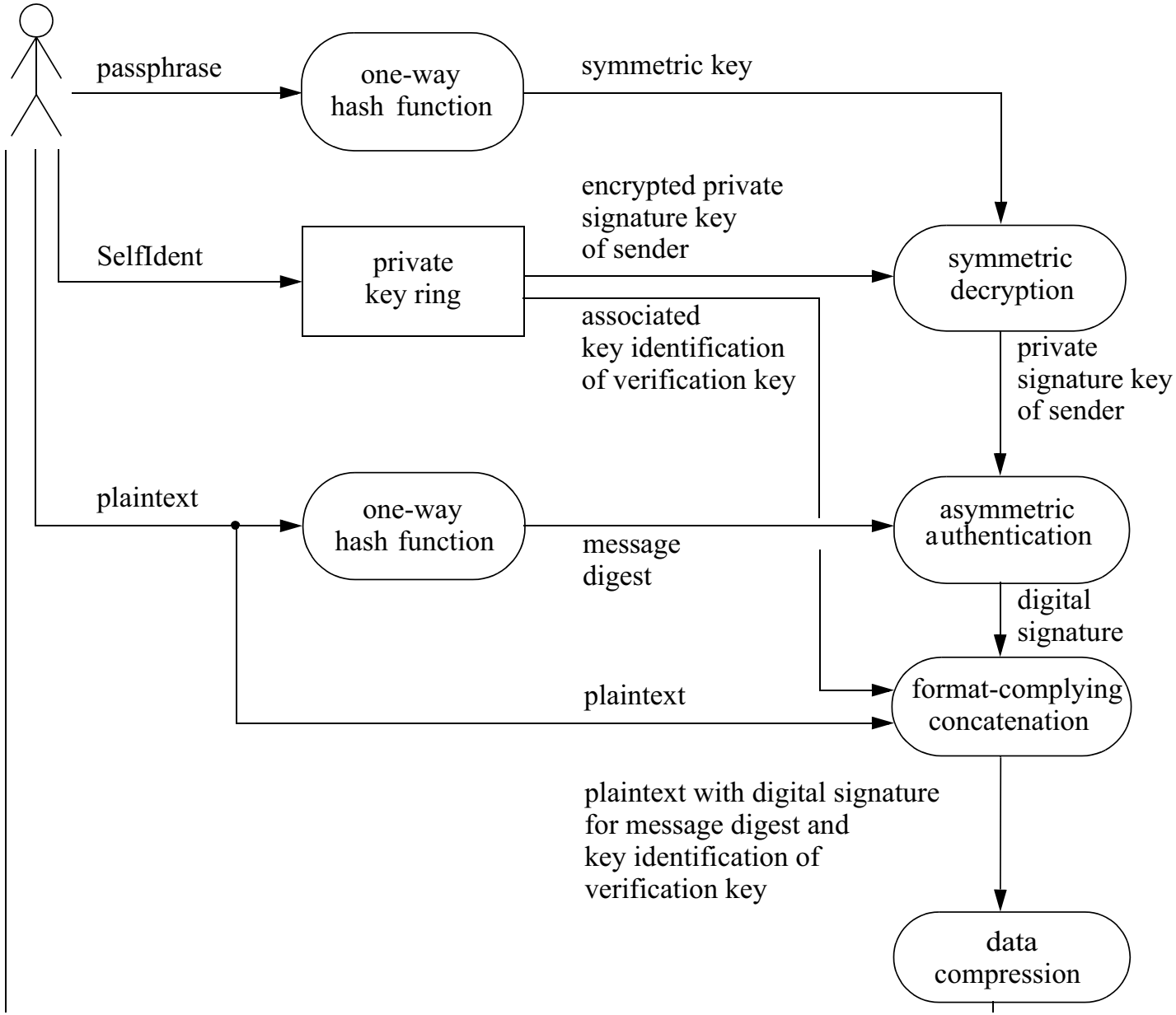
- *passphrases*:
  for generating symmetric keys
  - to protect private asymmetric (decryption or signature) keys
  - for secure end-to-end connections
  - to protect the user's own files

- *key management* by means of a *private key ring* and a *public key ring*:
  - for storing the user's own private asymmetric keys
  - for storing, assessing and selecting the public asymmetric keys of the user's partners

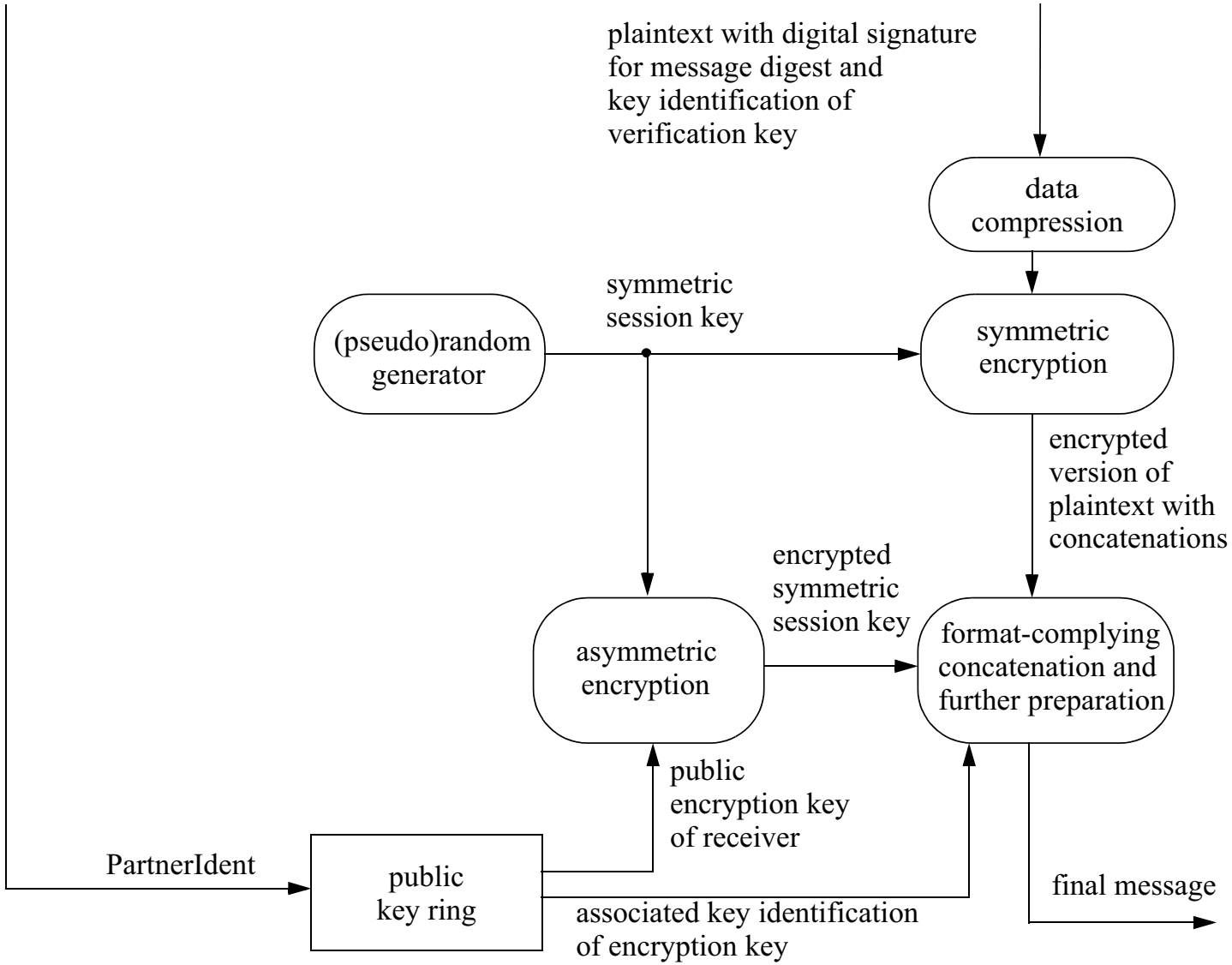
Conceptual design of secure message transmission



Secure message transmission: preparations



Secure message transmission: encryption and finalization



PGP parameters

- *SelfIdent*,
  denoting the participant acting as a sender

- *passphrase*,
  as an exhibit for a proof of authenticity of the sender

- *PartnerIdent*,
  denoting the intended receiver

- *plaintext*,
  to be communicated from the sender to the receiver

Key management

- only the keys for the asymmetric mechanisms are stored persistently

- a secret key for any symmetric mechanism employed is
  - generated or recovered only when it is actually needed
  - afterwards immediately destroyed

Using a symmetric secret key for securing an asymmetric private key

- authentication is strongly needed
  (owner is distinguished among all other participants):
  - authentication by demanding a passphrase
  - from which the secret key is directly derived by a one-way hash function

- the secret key is never stored persistently
  but is always dynamically regenerated whenever it is required

- the task of keeping secret information is reduced to
  the burden of handling the passphrases,
  and thus is mainly shifted to the users of PGP in diminished form

Using a symmetric secret key as a session key for the hybrid method

- the symmetric secret key is generated on the fly
  by a (pseudo)random generator,

  used only once for encrypting content data
  by means of the block cipher employed,

  and then itself asymmetrically encrypted for later use
  when the content data must be recovered

- on the side of the participant acting as the encryptor,
  there is no need to keep the secret key

- on the side of the participant acting as the later decryptor,
  the secret key is held in encrypted form:
  when the non-encrypted form of the secret key is recovered,
  the first case applies, since authentication is strongly needed

Private key ring

- this ring contains the user's own key pairs, each of which consists of
  - a *private signature key* and
  - the matching *public verification key*

  or
  - a *private decryption key* and
  - the matching *public encryption key*

- each private key is stored in encrypted form

- each private key is stored together with
  - a *timestamp*
  - a derived *key identification* for referencing the key pair
  - an *identification* of the owner
  - some further administrative data

- the access to a private key is secured by a passphrase that the owner selected when he issued the PGP command to generate and store a key pair

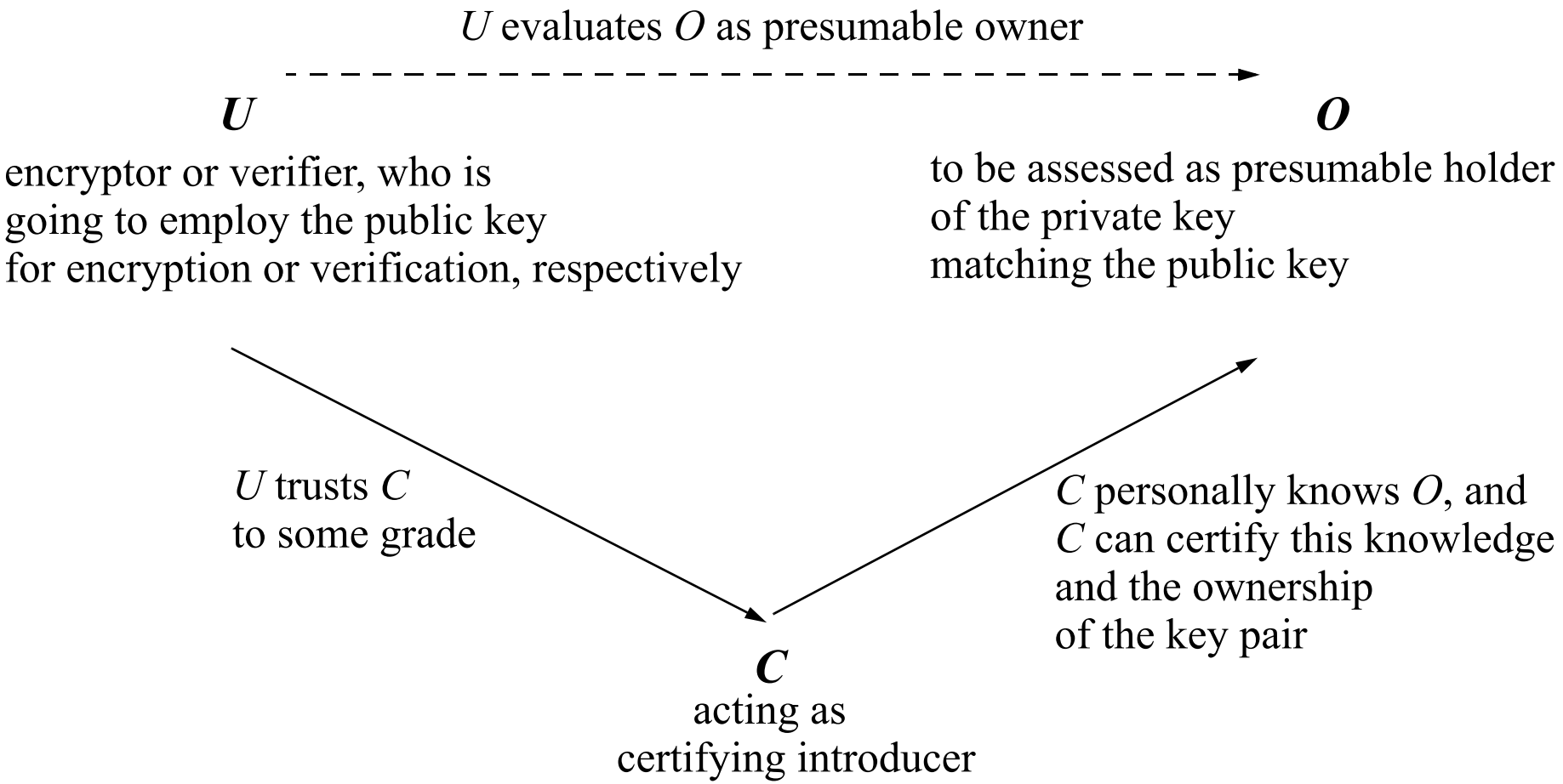
Public key ring

- this ring contains the
  - *public verification keys* and
  - *public encryption keys*

  of the owner's communication partners

- a key is complemented by
  - a *timestamp*
  - a derived *key identification*
  - an *identification* of the partner
  - further administrative data
  - some further entries to be used to assess the public key

Assessment of public keys



Two basic relationships

- one participant *C(ertifier) personally knows* another participant *O(wner)* such that *C* can certify that a public key *k* belongs to *O*:

  the participant *O* is the legitimate owner of the pertinent key pair and thus the actual holder of the matching private key;

  the participant *C* (perceived as the *introducer* of *O*)
  confirms such an ownership by issuing and digitally signing
  a *key certificate*, also known as an *identity certificate*,
  basically consisting of
  - an identification *OIdent*
  - the public key *k* together with the pertinent digital signature

- one participant *U(ser)*, willing to encrypt or to verify a message, may *trust* another participant *C(ertifier)* to various degrees to issue correct key certificates;

  PGP suggests four *trust grades* (more sophisticated grades could be used):
  unknown, *untrusted*, *marginally_trusted*,*completely_trusted*

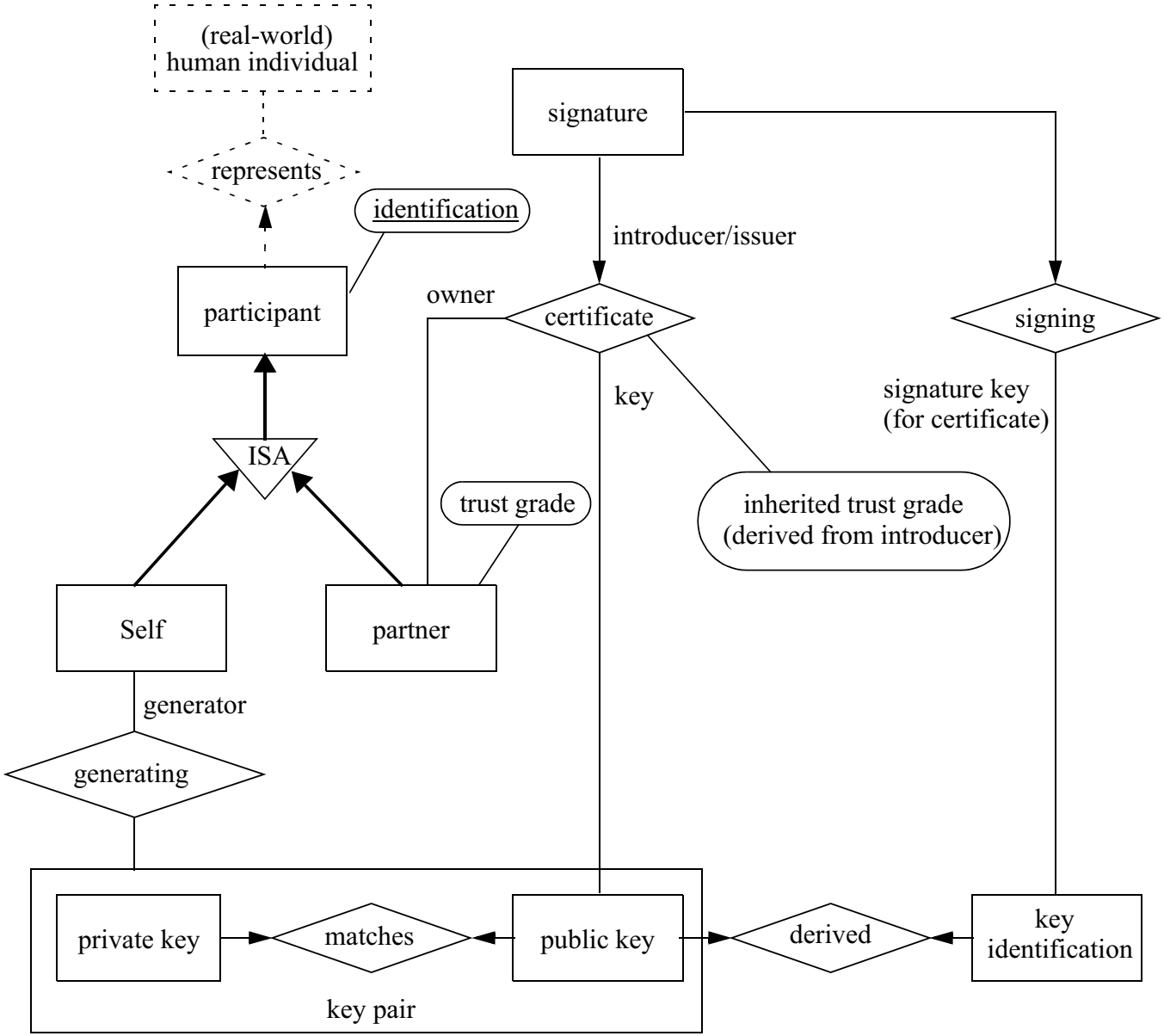
A derived relationship

- the participant *U(ser) evaluates* another participant *O(wner)* as the presumable owner of a public key *k*,

  on the basis of successfully verifying
  the digital signature of a key certificate
  of the form $(OIdent, k)_{signature}$,
  issued and digitally signed by some introducer *C(ertifier)*

- the grade of the evaluation of *O* is derived from the grade of the trust in the introducer *C*

Participants, asymmetric keys, signatures and their relationships



Kerberos

- supports participants,
  may be unknown to each other before interacting,
  who are acting in a *distributed* computing system,
  - as a (functional) server
  - as a client

- enables servers to specify and enforce a *security policy*
  that describes the permissions of potential clients

- initializes and maintains secure *end-to-end connections*
  that achieve mutual *authenticity* and enforce *confidentiality*

- proposes the use of a *trusted third party*, known as a Kerberos server,
  to dynamically act as a *mediator* on a request from of a client,
  on the basis of statically agreed relationships
  between the participants and the Kerberos server

Overall security achievements and trust

- participants assign *trust* to the Kerberos server:
  - each of the participants and the Kerberos server have to initially exchange a *secret* (*key*) for enabling symmetric authentication
  - a server has to permanently delegate the granting of permissions to the Kerberos server
  - however, within Kerberos, permission granting is degenerated to allow accesses whenever proper authentication has been achieved

Basic blocks

- *symmetric encryption*,
  - for evaluating the authenticity of messages
    on the basis of the possession of a secret symmetric *key*
  - for enforcing the confidentiality and integrity of messages

- *passwords*,
  used as substitutes for the secret symmetric key
  agreed between a particular participant and the Kerberos server

- *one-way hash function*
  for dynamically regenerating a key from the substituting password

- *random generator*
  to generate symmetric *session keys*,
  to be used for a secure *end-to-end connection*
  during a client–server interaction

- *timestamps*,
  used as indications of the *freshness* of messages

- *nonces* (random bit strings),
  used as *challenges* to be included in responses

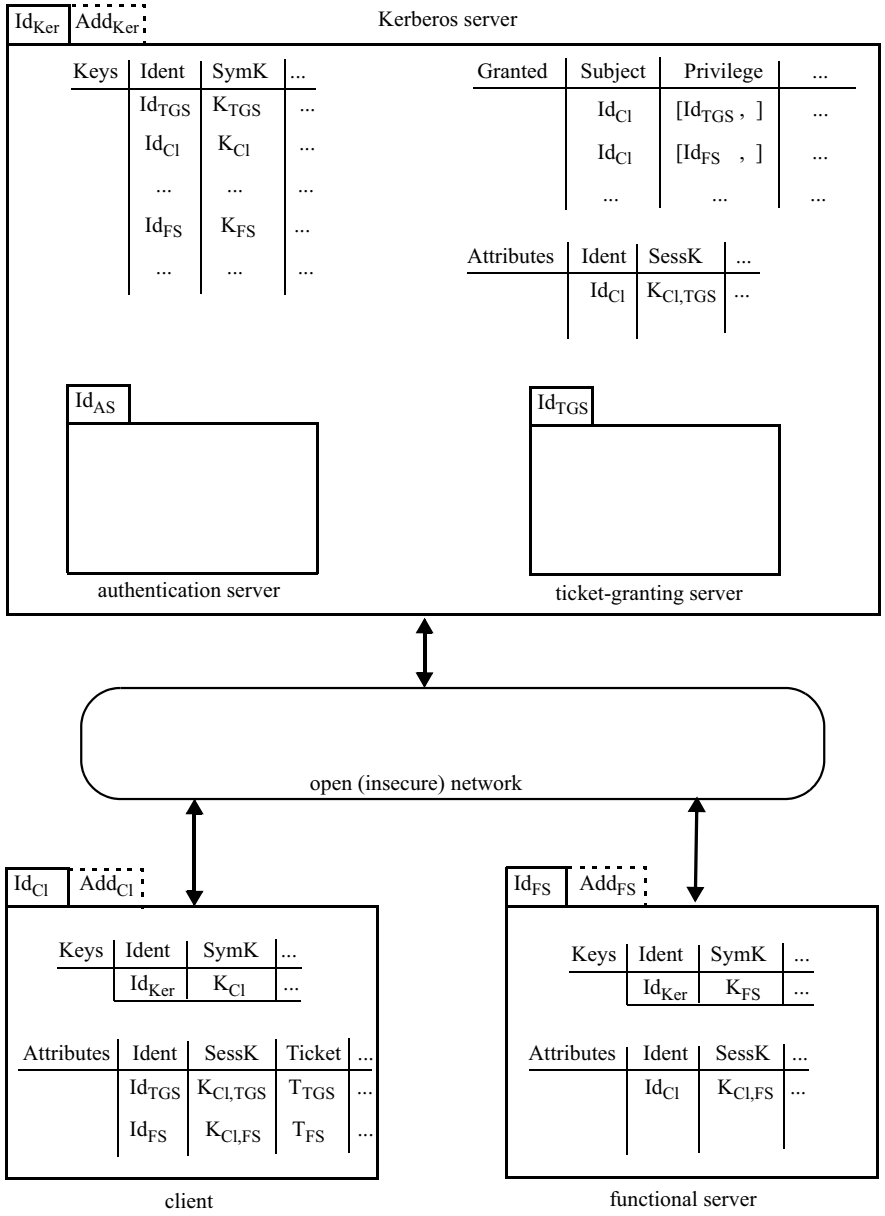
- *tickets*,
  used as a special kind of *credential* that
  – encode *privileges* granted to a client as a grantee
  – are shown to a server as a (self-protecting) controlled object

- *validity* specifications for tickets

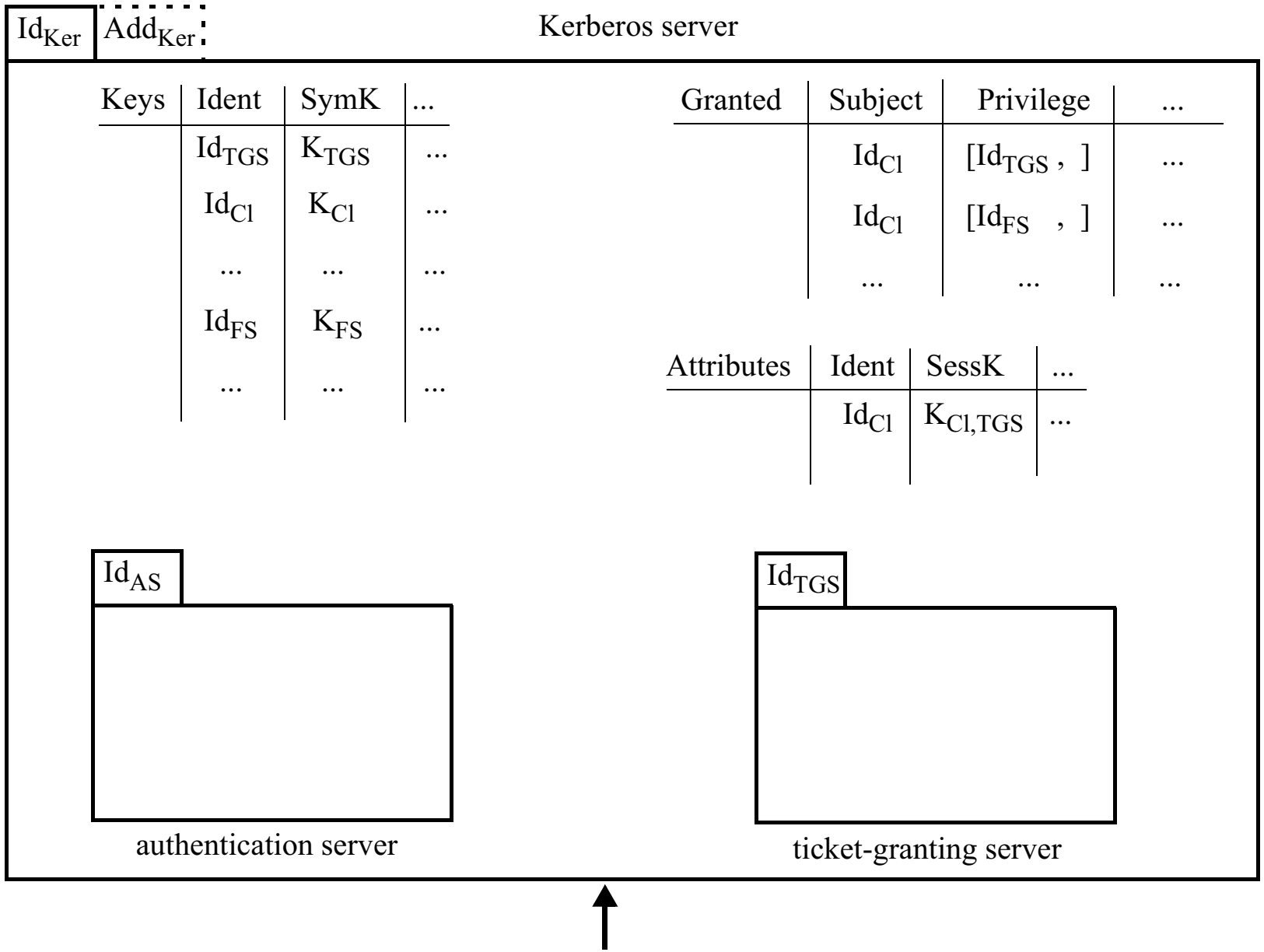
- *access decisions*,
  taken by a server on the basis of shown tickets

- *delegation*
  of the issuing of tickets by the Kerberos server on behalf of a server

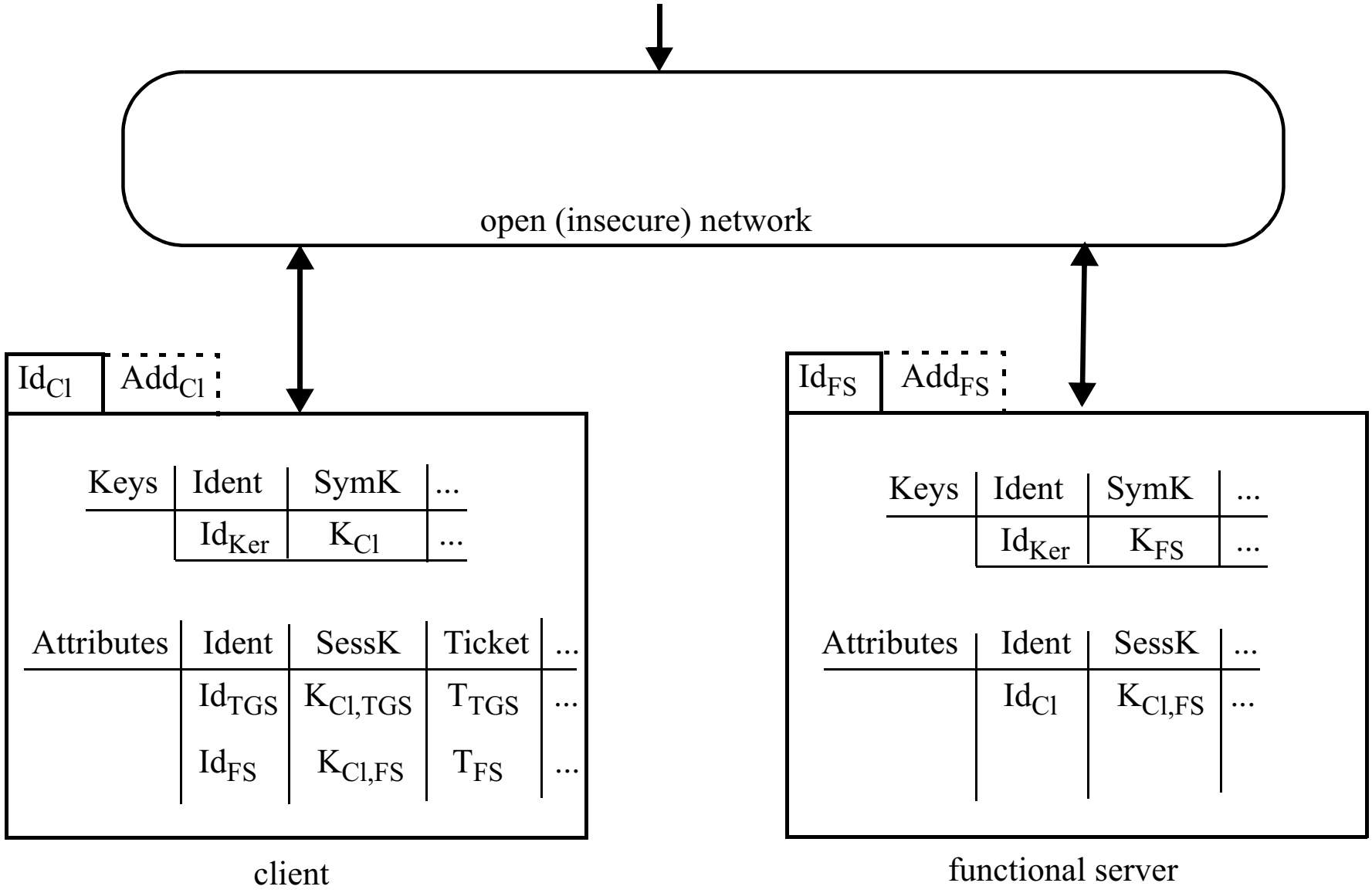
Conceptual design: structures



Structure of a Kerberos server



Structures of a client and a functional server



Names, identifiers, addresses and keys

- Kerberos server
  - *AS* *authentication server*
  - *TGS* *ticket-granting server*

- participant *P* (client *Cl*, Kerberos server *Ker* with components *AS* and *TGS*)
  - Id_P unique identifier
  - Add_P network address
  - K_P secret symmetric key for a symmetric encryption method

Kerberos server

- *Keys* local table with:
  - *Ident(ifier)* column for identifier Id_P
  - *Sym(metric)K(ey)* column for key K_P of each registered participant P
  - ... columns for further administrative data

- *Granted* local table with columns
  - *Subject*
  - *Privilege*

  to represent the *permissions* of clients to access services:
  - (Subject: Id_{Cl}, Privilege: [Id_{FS},]):
    the participant identified by Id_{Cl} is permitted, as a client,
    to access the services offered by the functional server identified by Id_{FS}
  - (Subject: Id_{Cl}, Privilege: [Id_{TGS},]):
    the participant identified by Id_{Cl} is permitted, as a client,
    to access the service of the ticket-granting server,
    which is identified by Id_{TGS} and is a component of the Kerberos server

A client

- *Keys* local table referring to
  the identifier Id_{Ker} of the Kerberos server

- however, for a human individual acting as a client,
  the secret symmetric key is *not* permanently stored:

  instead, the individual can choose a secret *password*,
  from which the symmetric key can be repeatedly computed
  by use of a *one-way hash function*

Rounds of the Kerberos protocol

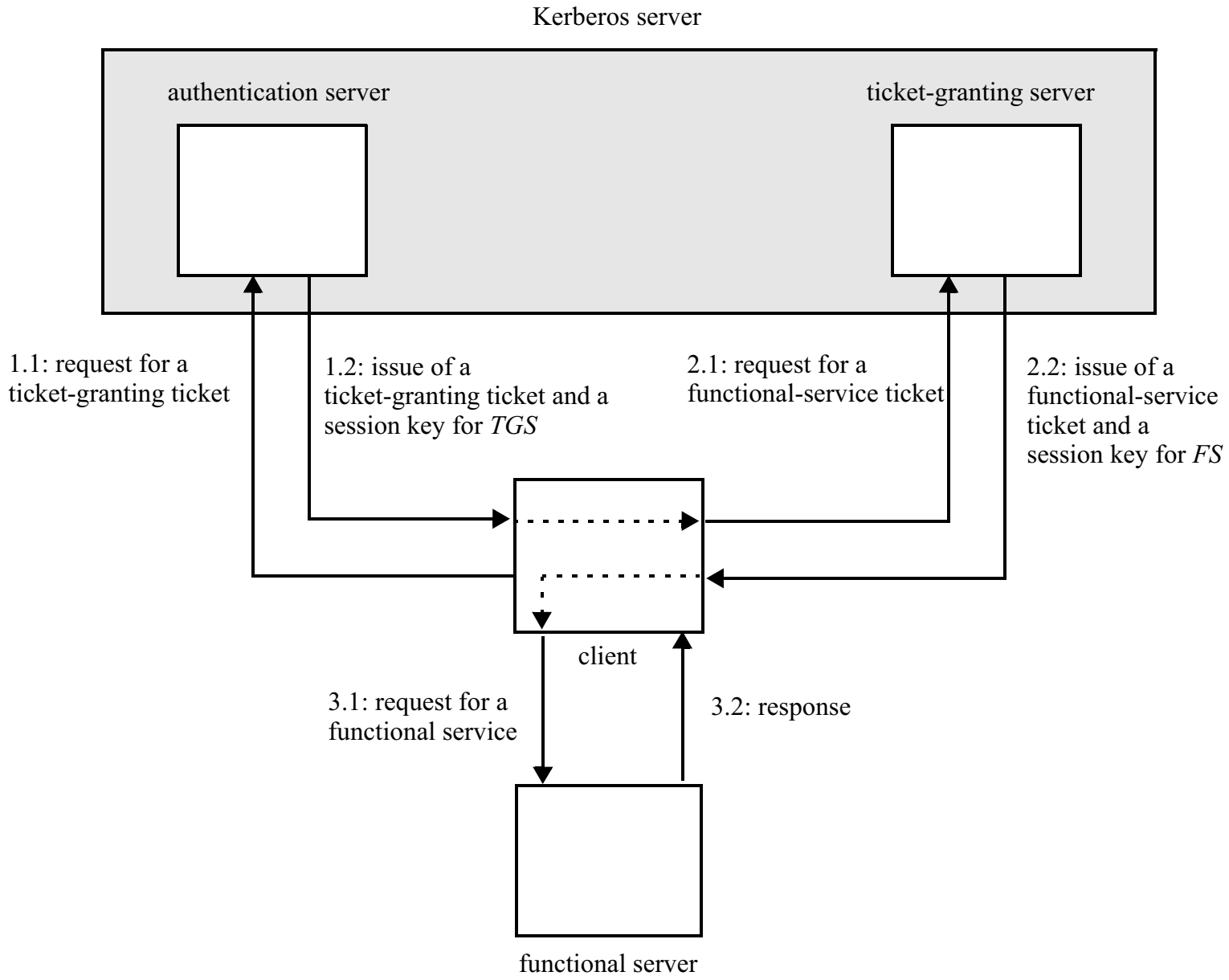
- each round is initialized by a client and has two messages

- **first round,**
  executed once per client session (can be integrated within a login procedure):
  to authenticate the client for the later process of
  obtaining and exploiting a reusable *ticket*
  that expresses a *privilege* for a service

- **second round,**
  performed once for each functional server
  that is contacted during a client session:
  to actually grant the privilege to the client

- **third round,**
  repeatedly called for each actual *service invocation*:
  to exploit the granted privilege

Messages between a client, a Kerberos server and a functional server



Rough meanings of the six different Kerberos messages

- 1.1: a client requests a ticket-granting ticket from the authentication server
- 1.2: the authentication server issues a ticket-granting ticket for the client, together with a session key for a secure end-to-end connection between the client and the ticket-granting server
- 2.1: a client requests a functional-service ticket from the ticket-granting server
- 2.2: the ticket-granting server issues a functional-service ticket for the client, together with a session key for a secure end-to-end connection between the client and the functional server
- 3.1: a client requests a service invocation from the functional server
- 3.2: the functional server responds to the client

Simplified message 1.1

the client *Cl*

- requests a ticket-granting ticket from the authentication server *AS*, to be shown to the ticket-granting server *TGS*
- adds the wanted validity specification $Validity_1$
- includes a nonce $Nonce_1$

$$Id_{Cl}\,,\, Id_{TGS}\,,\, Validity_1\,,\, Nonce_1$$

Simplified message 1.2

the authentication server *AS*

- issues a ticket-granting ticket $Ticket_{TGS}$ to the client Cl,
  to be shown to the ticket-granting server TGS

- attaches
  - a session key $K_{Cl,TGS}$ for a secure end-to-end connection
    between the client Cl and the ticket-granting server TGS
  - the wanted $Validity_1$
  - the received $Nonce_1$

  where the attachments are encrypted with the client's secret key K_{Cl}

$$Id_{Cl}\,,\, Ticket_{TGS}\,,\, Enc(\, K_{Cl}\,,\, [\, K_{Cl,TGS}\,,\, Validity_1\,,\, Nonce_1\,,\, Id_{TGS}\,]\,)$$

Ticket-granting ticket

the ticket-granting ticket $Ticket_{TGS}$ contains

- the session key $K_{Cl,TGS}$ for a secure end-to-end connection between the client Cl and the ticket-granting server TGS
- the client's identifier Id_{Cl}
- the client's network address Add_{Cl}
- the wanted $Validity_1$

and is encrypted with the ticket-granting server's secret key K_{TGS}

$$Ticket_{TGS} = Enc(\, K_{TGS}\,, [\, K_{Cl,TGS}\,, Id_{Cl}\,, Add_{Cl}\,, Validity_1\,]\,)$$

Simplified message 2.1

showing the ticket $Ticket_{TGS}$, **the client *Cl***

- requests a functional-service ticket from the ticket-granting server *TGS*, to be shown to the functional server *FS*
- adds the wanted validity specification $Validity_2$
- includes a nonce $Nonce_2$
- attaches
  - an authentificator $Auth_{Cl,TGS}$ that encrypts the client's identifier Id_{Cl}
  - a timestamp TS_3

  where the authentificator is encrypted with the session key $K_{Cl,TGS}$ (which is made available to the ticket-granting server by the ticket $Ticket_{TGS}$)

$$Id_{FS}\,,\ Validity_2\,,\ Nonce_2\,,\ Ticket_{TGS}\,,\ Auth_{Cl,TGS}$$

where

$$Auth_{Cl,TGS} = Enc(\ K_{Cl,TGS}\,,\ [\ Id_{Cl},\ TS_3\ ]\ )$$

Simplified message 2.2

the ticket-granting server

- issues a functional-service ticket $Ticket_{FS}$ to the client Cl,
  to be shown to the functional server FS

- attaches
  - a session key $K_{Cl,FS}$ for a secure end-to-end connection
    between the client Cl and the functional server FS
  - the wanted $Validity_2$
  - the received $Nonce_2$

  where the attachments are encrypted with the session key $K_{Cl,TGS}$

$$Id_{Cl}\,,\, Ticket_{FS}\,,\, Enc(\, K_{Cl,TGS}\,,\, [K_{Cl,FS}\,,\, Validity_2\,,\, Nonce_2\,,\, Id_{FS}\,]\,)$$

Functional-service ticket

the functional-service ticket $Ticket_{FS}$ contains

- the session key $K_{Cl,FS}$ for a secure end-to-end connection between the client Cl and the functional server FS
- the client's identifier Id_{Cl}
- the client's network address Add_{Cl}
- the wanted $Validity_2$

and is encrypted with the functional server's secret key K_{FS}

$$Ticket_{FS} = Enc(\, K_{FS}\,, [K_{Cl,FS}\,, Id_{Cl}\,, Add_{Cl}\,, Validity_2\,]\,)$$

Simplified message 3.1

showing the ticket $Ticket_{FS}$, **the client *Cl***

- requests a service invocation from the functional server *FS*

- includes
  - an authentificator $Auth_{Cl,FS}$
    that encrypts the client's identifier Id_{Cl}
  - a timestamp TS_4

  where the authentificator is encrypted with the session key $K_{Cl,FS}$
  (which is made available to the functional server by the ticket $Ticket_{FS}$)

$$Ticket_{FS}\,,\, Auth_{Cl,FS}$$

where

$$Auth_{Cl,FS} = Enc(\, K_{Cl,FS}\,,\, [\, Id_{Cl},\, TS_4\,]\,)$$

Simplified message 3.2

the functional server *FS*

- responds to the client
  by sending back the received timestamp TS_4,

  encrypted with the session key $K_{Cl,FS}$

$$Enc(\, K_{Cl,FS}\, ,\, TS_4)$$

13 Symmetric Encryption

Encryption mechanism: functionality

- underlying sets:
  - D domain set of (possible) *plaintexts*
  - R range set of (possible) *ciphertexts*
  - $K = EK \times DK$ set K of (possible) *keys*, each of which comprises
    - $ek \in EK$ *encryption key*
    - $dk \in DK$ *decryption key*

- $Gen: \rightarrow K$ *key generation* algorithm,
  might take a natural number l as a *security parameter*

- $Enc: EK \times D \rightarrow R$ *encryption* algorithm,
  transforms a plaintext $x \in D$
  into a ciphertext $y = Enc(ek, x) \in R$
  using an encryption key $ek \in EK$

- $Dec: DK \times R \rightarrow D$ *decryption* algorithm,
  transforms a ciphertext $y \in R$
  into a plaintext $x = Dec(dk, y) \in D$
  using a decryption key $dk \in DK$

Encryption mechanism: properties

- ***correctness***
  using a generated key pair,
  any encryption can be reversed by the corresponding decryption, i.e.,
  for all keys $(ek, dk) \in EK \times DK$ generated by *Gen*,
  for all plaintexts $x \in D$:

  $$Dec(dk, Enc(ek, x)) = x$$

- ***secrecy* (*naive* version)**
  without knowing the pertinent decryption key *dk*,
  an (unauthorized) observer of a ciphertext $y = Enc(ek, x)$
  cannot "determine" the corresponding plaintext x

  (*semantic* version: such an observer
  can "determine" only those properties of the corresponding plaintext x
  that he could "determine" without knowing the ciphertext y at all)

- ***efficiency***
  algorithms *Gen*, *Enc* and *Dec* are efficiently computable

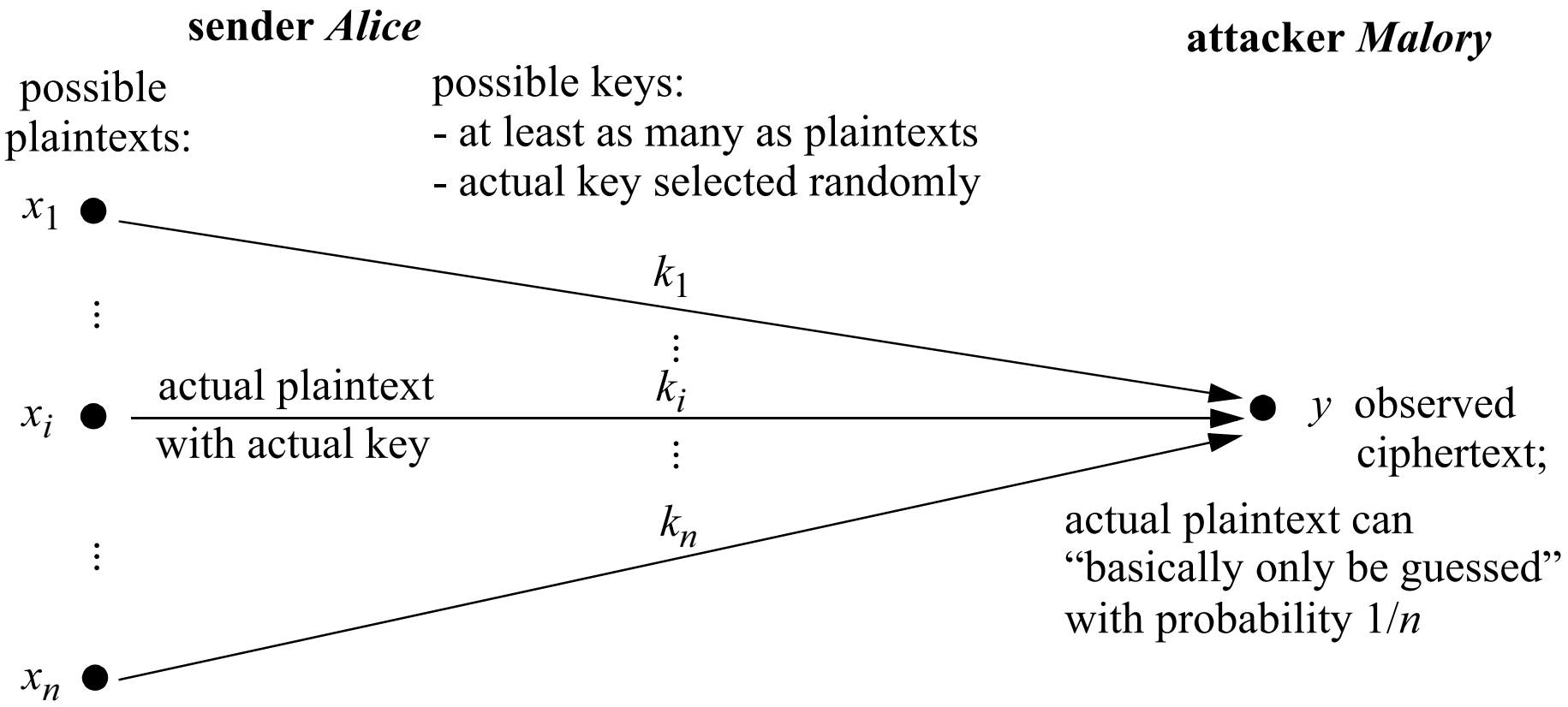
Classification

- ***mode of operation*:**
  blockwise or *streamwise*

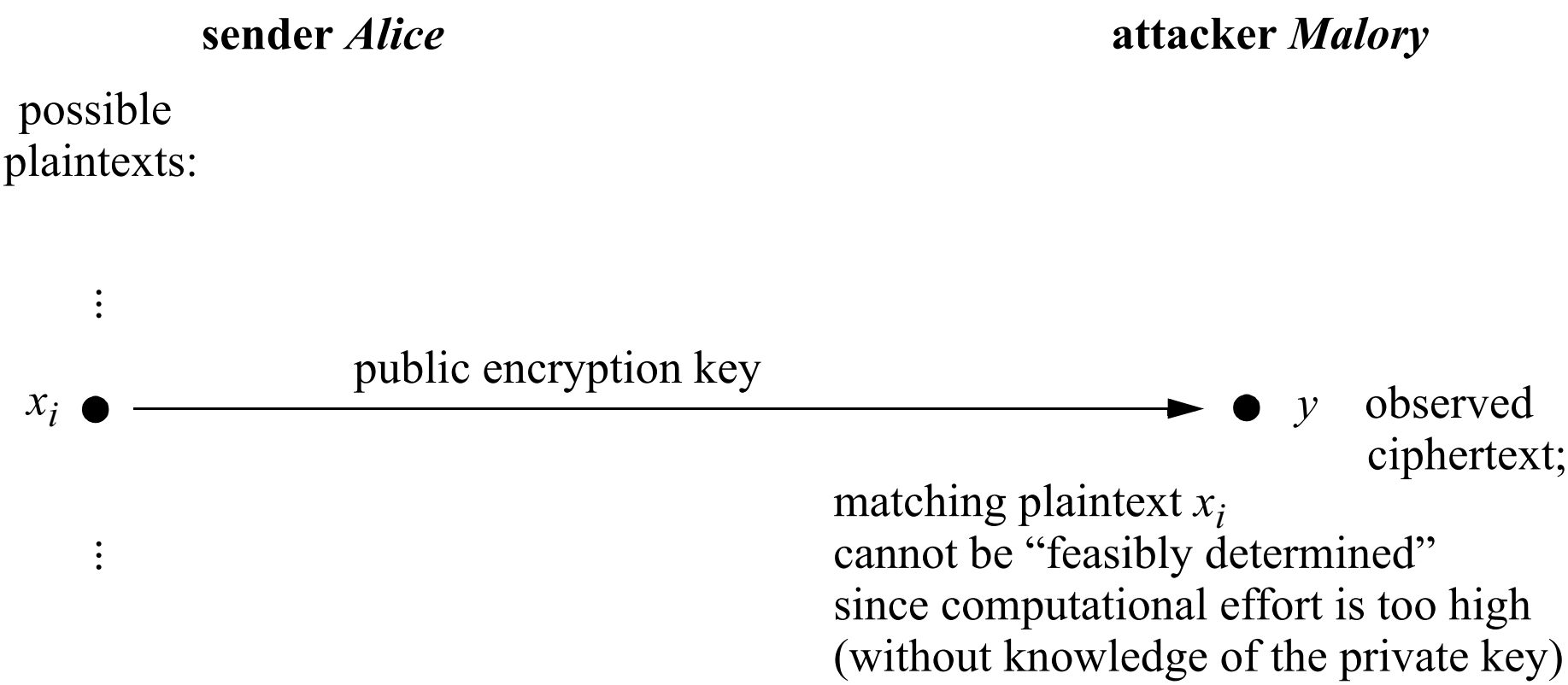
- ***relationship between keys*:**
  symmetric or *asymmetric*

- ***justification of a secrecy property*:**
  one-time key or *one-way function* or *chaos*

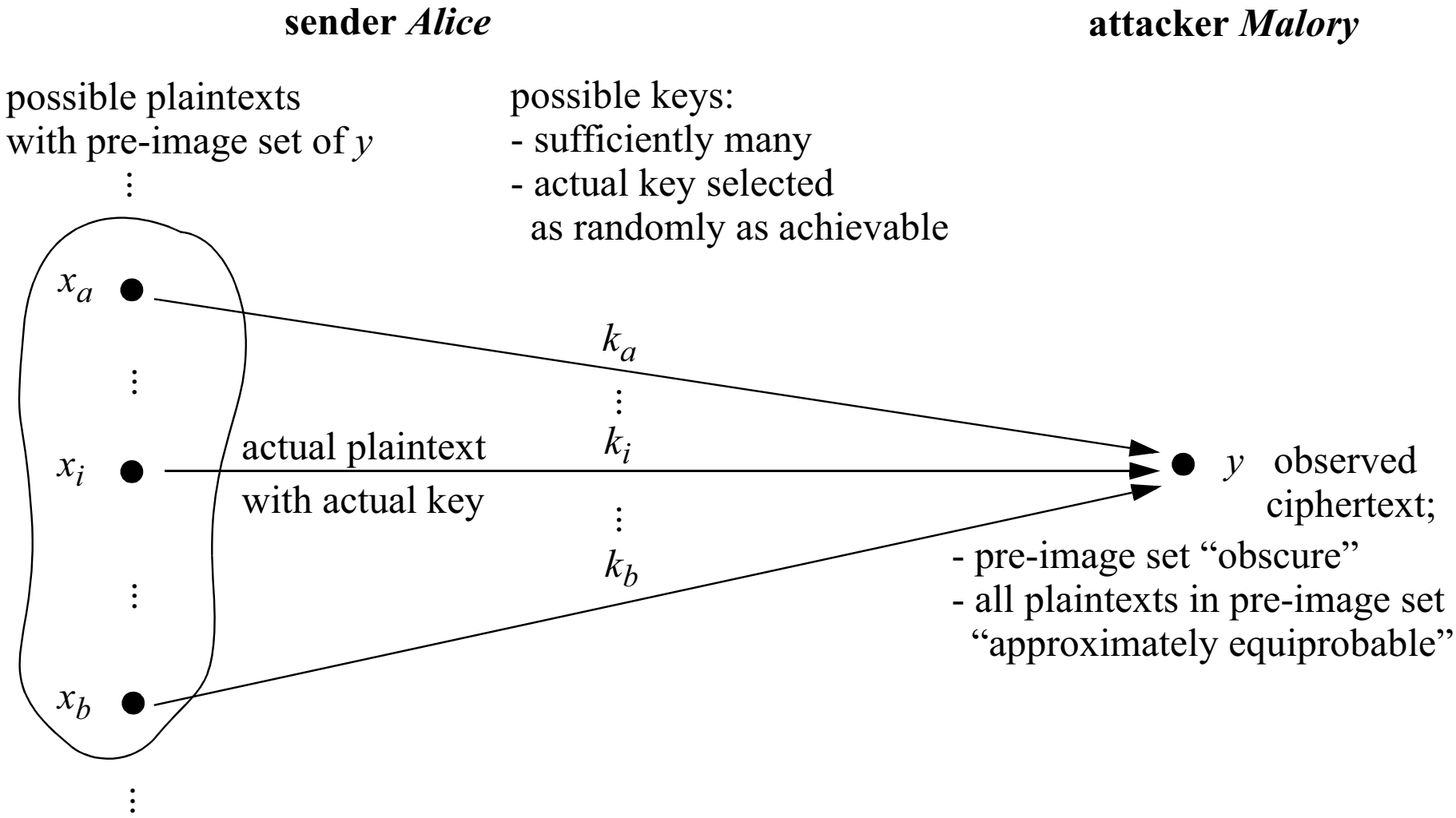
Probability-theoretic secrecy property (one-time key approach)



Complexity-theoretic secrecy property (one-way function approach)



Empirical secrecy property (chaos approach/confusion and diffusion)

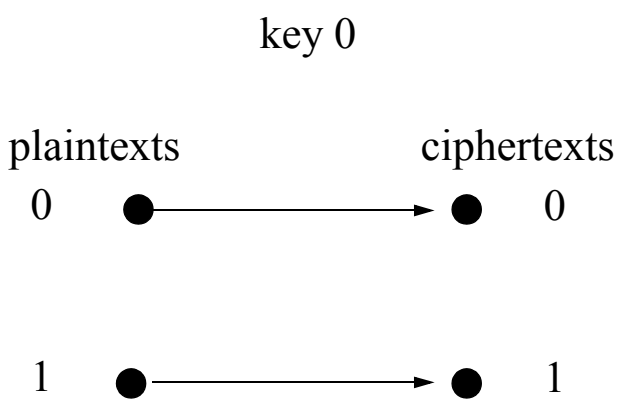


One-time keys and perfect ciphers (Vernam)

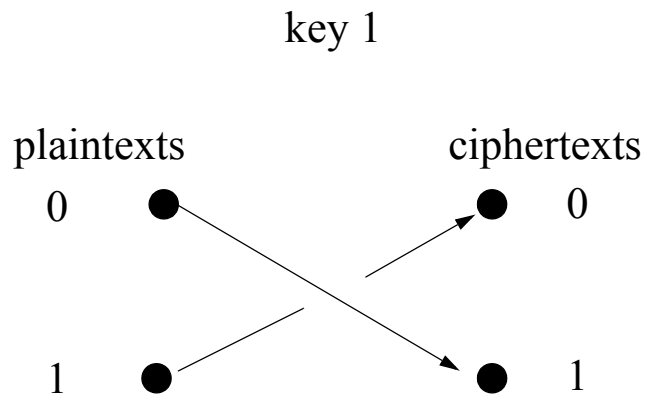
- are based on
  - a sufficient (and „nearly necessary“) condition for *perfectness*, achieving *probability-theoretic secrecy*
  - the resulting group-based construction
- are *symmetric*,
  having *identical* encryption key and decryption key
- are restricted to a *single* key usage
- operate *streamwise* by considering a plaintext as a sequence of bits, each of which is treated separately

One-time keys: treating a single bit

- *plaintext* domain, *ciphertext* range and *key* set are chosen as $\{0,1\}$
- set $\{0,1\}$ is seen as the carrier of the *group* $(\mathbf{Z}_2,+,0)$ of *residue classes* modulo 2, where the residue classes are identified with their representatives 0 and 1
- *group operation* of addition modulo 2 is identical to the Boolean operation XOR (exclusive or, denoted by the operator $\oplus$)



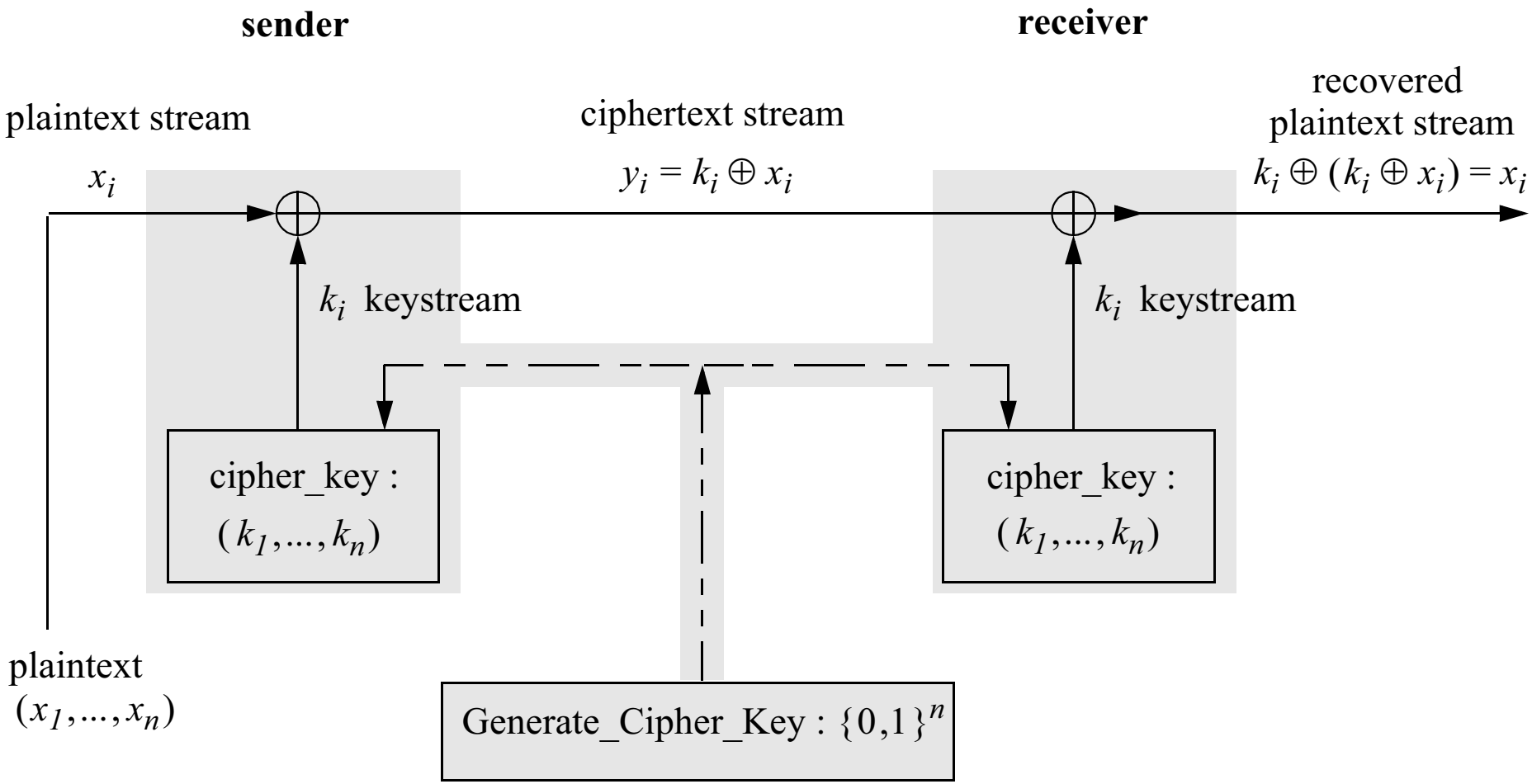
$Enc(0,x) = 0 \oplus x = x$
delivers the identity permutation



$Enc(1,x) = 1 \oplus x = 1 - x$
delivers the exchanging permutation

One-time keys: handling bit strings of length n

- employ the corresponding *product group*:
  - take the group $(Z_2, +, 0)$ n times
  - define the *group operation* componentwise



One-time keys: underlying sets

- plaintexts: bit strings of length n, i.e.,
  "streams" $(x_1, \ldots, x_n)$ of length n over the set $\{0,1\}$
- ciphertexts: bit strings of the same length n, i.e.,
  "streams" $(y_1, \ldots, y_n)$ of length n over the set $\{0,1\}$
- keys: bit strings of the same length n, i.e.,
  "streams" $(k_1, \ldots, k_n)$ of length n over the set $\{0,1\}$

One-time keys: algorithms

- ***key generation*** algorithm *Gen(erate_Cipher_Key)*
  selects a "truly random" *cipher key* $(k_1,\ldots,k_n)$

- ***encryption*** algorithm *Enc*
  handles the plaintext $(x_1,\ldots,x_n)$ and the cipher key $(k_1,\ldots,k_n)$ as streams;
  treats each corresponding pair of a plaintext bit x_i and a cipher key bit k_i
  as input for a XOR operation, yielding a ciphertext bit
  $y_i = k_i \oplus x_i$

- ***decryption*** algorithm *Dec*
  handles the ciphertext $(y_1,\ldots,y_n)$ and the cipher key $(k_1,\ldots,k_n)$ as streams;
  treats each corresponding pair of a ciphertext bit y_i and a cipher key bit k_i
  as input for a XOR operation,
  yielding the original plaintext bit x_i *correctly*:

  $k_i \oplus y_i = k_i \oplus (k_i \oplus x_i) = (k_i \oplus k_i) \oplus x_i = 0 \oplus x_i = x_i$

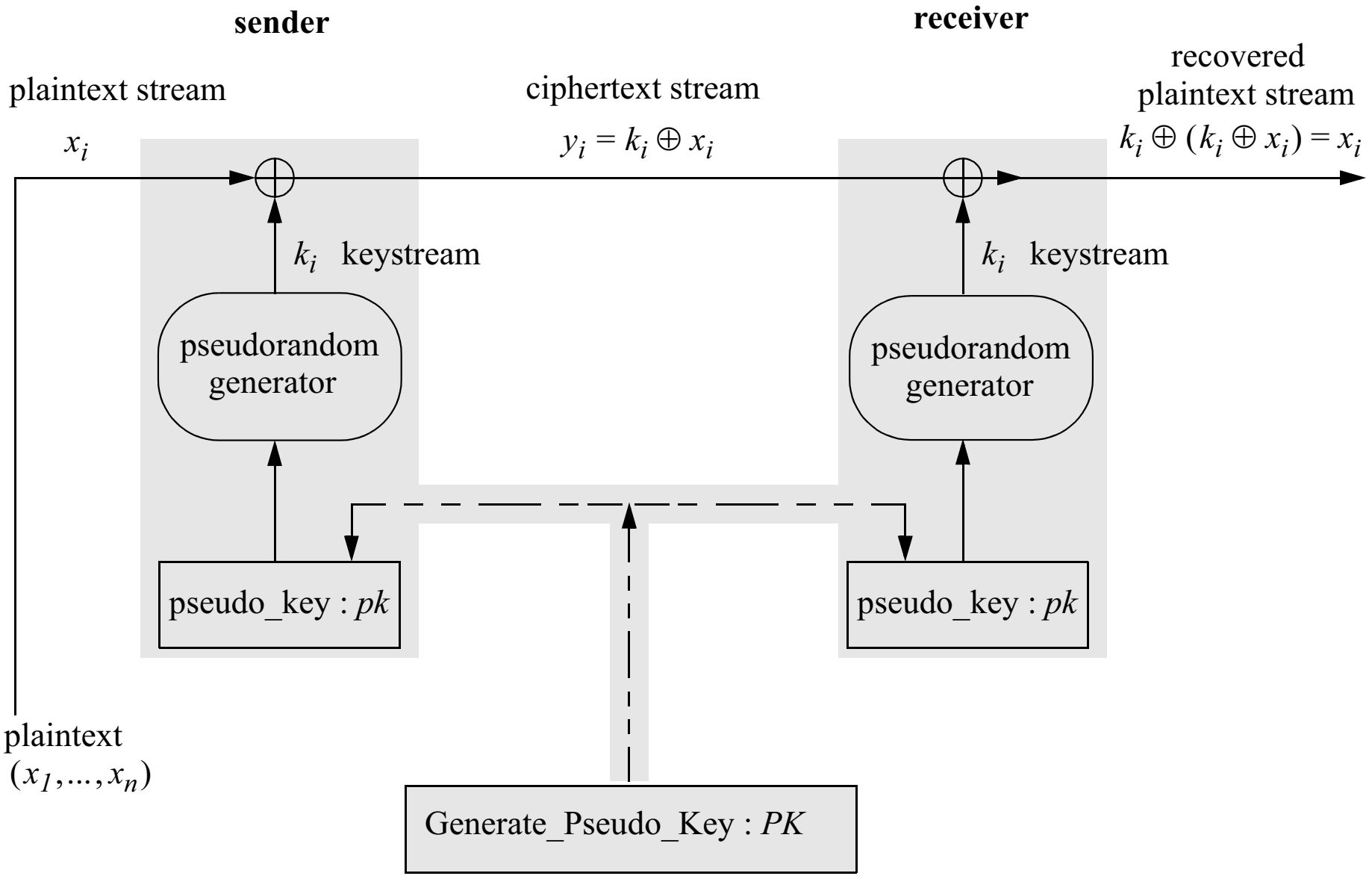
One-time keys: applications

- restriction to using a key *only once* is crucial:
  observing a ciphertext/plaintext pair, an attacker achieves complete success:
  solve, for each position i, the equation $y_i = k_i \oplus x_i$
  regarding the secret key bit as
  $k_i = y_i \oplus x_i$
- considering the transmission of a *single* message:
  qualified to the best possible extent regarding *secrecy* and *efficiency*
- as a trade-off for the best secrecy – proved to be inevitable:
  - secret cipher key can be used only once
  - secret cipher key must be as long as the anticipated plaintext
- as a *stand-alone* mechanism,
  pure one-time key encryption is practically employed
  only in dedicated applications with extremely high secrecy requirements
- however, basic approach is widely exploited in
  - variants
  - subparts of other mechanisms

Stream ciphers with pseudorandom sequences (Vigenère)

- are a variant of the *one-time key* encryption mechanism
- are obtained by replacing the "truly random" *cipher key*
  by a *pseudorandom* one
  that is determined by a short(er) *pseudo-key*
- are *symmetric*
- operate *streamwise* by considering a plaintext as a sequence of bits,
  each of which is treated separately
- cannot be *perfect* or *probability-theoretically secure* in practice,
  since the pseudo-key is often substantially shorter than
  the generated cipher key

Vigenère: overall structure



DES (Data Encryption Standard)

- has been a most influential example of the *chaos* approach, used worldwide
- designed by IBM and the National Security Agency (NSA) of the USA
- standardized by the National Bureau of Standards (NBS)
  in 1976/77 for “unclassified government communication”
- adopted by the American National Standards Institute (ANSI)
  in 1981 for commercial and private applications
- is a *symmetric* mechanism, admitting *multiple* key usage
- operates *blockwise*, where the block length is 64 bits
- has a key length of 56 bits:
  today, the pure form of this mechanism is considered to be outdated,
  as it suffers from a too short key length
- has a still useful variant: *Triple-DES*

Triple-DES

inputs:

- a plaintext x / a ciphertext y
- three different keys k_1, k_2, k_3

encryption algorithm: successively perform

- an encryption with k_1,
- a decryption with k_2
- another encryption with k_3

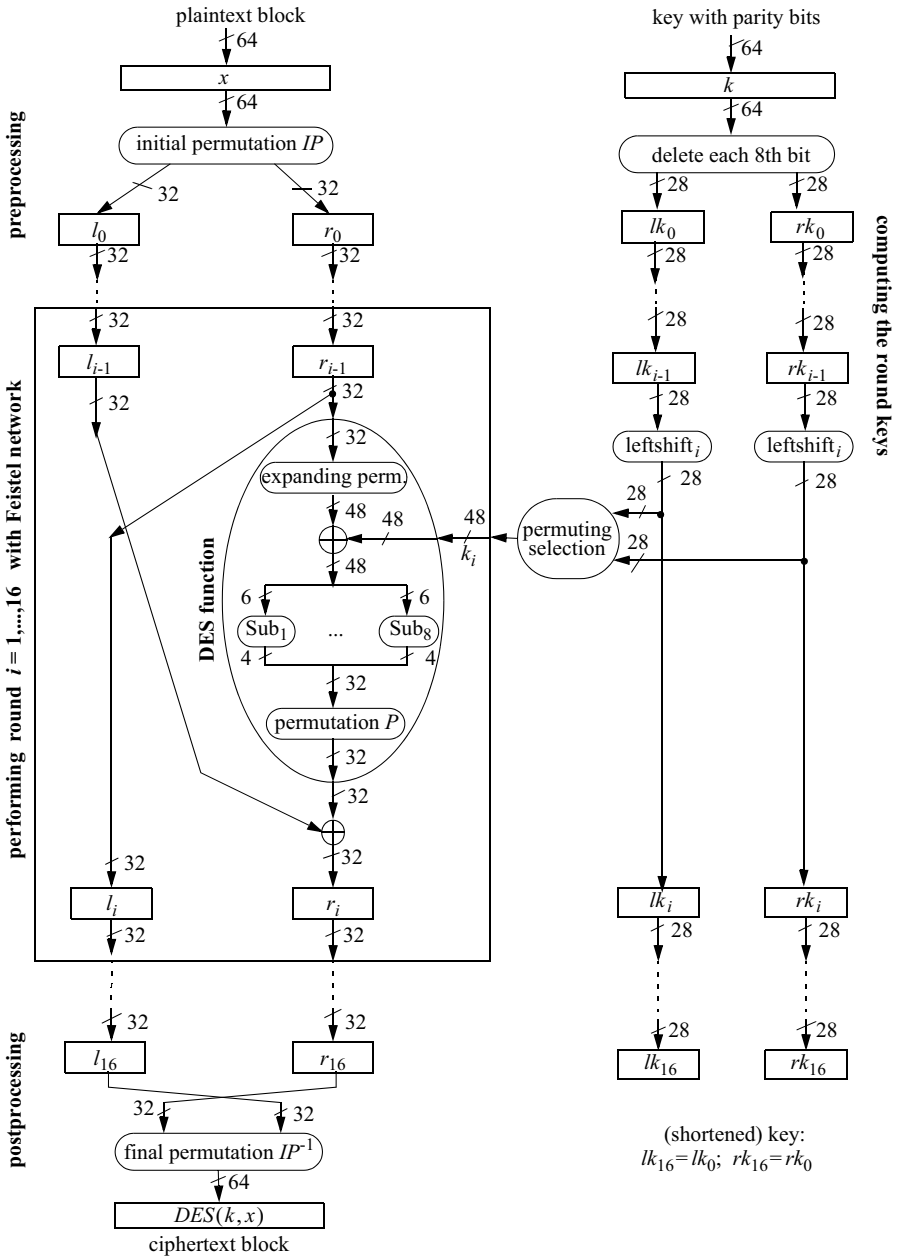
yielding the ciphertext y as

$$Enc(k_3, Dec(k_2, Enc(k_1, x)))$$

decryption algorithm: perform corresponding inverse algorithms to obtain

$$Dec(k_1, Enc(k_2, Dec(k_3, y)))$$

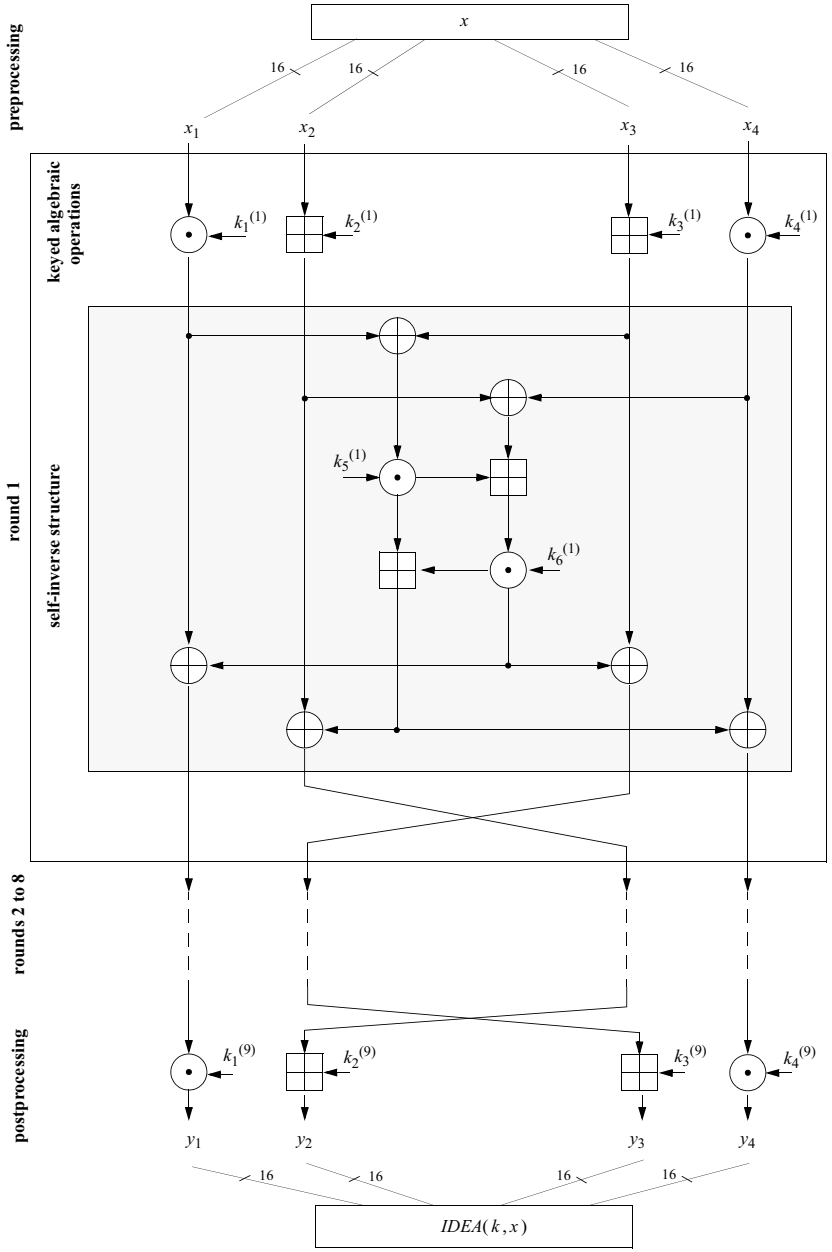
DES: overall structure



IDEA (International Data Encryption Algorithm)

- was developed as an alternative to DES
- is a further example of the *chaos* approach
- combines
  - a DES-like round structure operating on block parts and round keys
  - algebraic group operations
- was adopted for Pretty Good Privacy (PGP),
  but never reached common acceptance
- is *symmetric*, admitting *multiple* key usage
- operates *blockwise*, where the block length is 64 bits
- has a key length of 128 bits,
  still sufficient from today's perspective

IDEA: overall structure



AES–Rijndael (Advanced Encryption Standard)

- was designed by the Belgian researchers J. Daemen and V. Rijmen, winner of a public competition and evaluation, organized by the NIST
- follows the *chaos* approach, producing *confusion* and *diffusion*
- is *symmetric*, admits *multiple* key usage, operates *blockwise*
- permits block lengths varying from 128 bits to any larger multiple of 32 bits
- permits key length varying from 128 bits to any larger multiple of 32 bits
- is somehow restricted for standardization:
  - block length is fixed at 128 bits
  - key length is restricted to be 128, 192 or 256 bits, today regarded as sufficient to resist *exhaustive search* and *trial attacks*
- combines several long-approved techniques
  - operating *roundwise* on block parts and round keys
  - *superimposing* the randomness of the key on the blocks using *XOR*
  - *permuting* the positions of a block or a key
  - *employing* of advanced algebraic operations showing *one-way behavior*

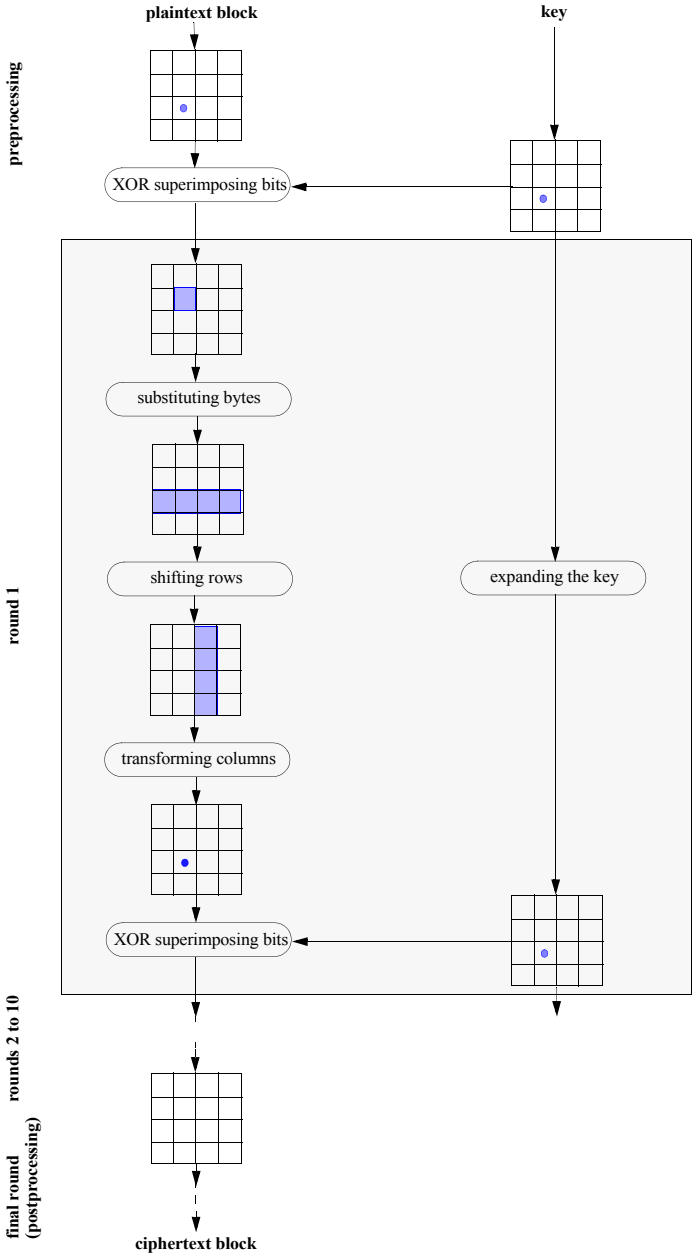
- operates on the following sets:
  - ***plaintexts*:**
    bit strings (blocks over $\{0,1\}$) of length 128 (or a larger multiple of 32),
    represented as a byte matrix of 4 rows and 4 columns,
    thus having 16 entries of 8 bits each
  - ***ciphertexts*:**
    bit strings (blocks) of the same length as the plaintext blocks
  - ***keys*:**
    bit strings of length 128 (or a larger multiple of 32),
    again represented as a byte matrix like the plaintexts

- employs three algorithms as follows
  - ***key generation*:** select a "truly random" bit string of length 128
  - ***encryption*:** perform byte matrix transformations, see next pages
  - ***decryption*:** invert the byte matrix transformations in reverse order,
    employing the round keys accordingly

Encryption algorithm $AES(k,x)$

- takes a key k and a plaintext x as input
- represents them as byte matrices
- operates on the current byte matrices
- uses some preprocessing and postprocessing
- performs 10 (or more for larger block or key lengths) uniform *rounds*
- executes four steps in one round:
  (1) bytewise substitutions
  (2) permutations that shift positions within a row
  (3) transformations on columns and
  (4) bitwise XOR operations with the round key

Structure of the AES–Rijndael symmetric block cipher



AES–step (1): bytewise substitutions

- step (1) is defined by a *non-linear*, *invertible* function S_{RD} on bytes, i.e., each byte of the current matrix is independently substituted by applying S_{RD}
- *invertibility* ensures that a *correct* decryption is possible just by applying the inverse function S_{RD}^{-1}
- *non-linearity* is aimed at achieving *confusion*, in terms of both
  - algebraic complexity
  - small statistical correlations between argument and value bytes
- the substitution function S_{RD} has two convenient representations:
  - tabular representation organized as a lookup table of size 16×16
  - algebraic representation

Tabular representation of the substitution function

argument byte *a*: seen as composed of two hexadecimal symbols *li* and *co*
value byte *v*: table entry for line *li* and column *co*

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Algebraic representation of the substitution function

- the representation treats a byte as an element of the finite field GF(2^8), where each *bit* of a *byte* is seen as
  a *coefficient* of a *polynomial* with degree at most 7

- the multiplicative structure is defined by
  the usual *multiplication of polynomials*, followed by a *reduction*
  modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$

- the function S_{RD} has a representation of the form

  $S_{RD}(a) = f(a^{-1}))$, where

  – the inversion operation refers to the multiplicative structure of GF(2^8)
  – f is an affine function in GF(2^8), basically described by
    – a suitable 8×8 bit matrix F
    – a suitable constant byte c

    such that

    $f(a) = (F \times a) \oplus c$

AES–step (2): permutations shifting positions within a row

- step (2) is defined by the offsets to be used for each of the rows:
  the offsets are 0, 1, 2 and 3 byte positions, meaning that
  - the first row remains invariant
  - the second, third and fourth rows are shifted
    by 8, 16 and 24 bit positions, respectively, to the left

- the shiftings are aimed at achieving good *diffusion*,
  and can be easily redone for a *correct* decryption

AES–step (3): transformations on columns

- step (3) is defined by a linear, invertible function MC_{RD} on "columns":
  each column of the current matrix is considered as an element of $\{0,1\}^{32}$ and independently substituted by applying MC_{RD}
- invertibility ensures that a *correct* decryption is possible
- the specific selection of MC_{RD} is aimed mainly at achieving *diffusion*, now regarding the rows of the byte matrices
- additionally, the selection was influenced by efficiency reasons
- MC_{RD} admits an algebraic definition in terms of polynomial multiplication:

$$MC_{RD}\left(\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}\right) = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

AES–step (4): bitwise XOR operations with the round key

- XOR *superimposes* the *randomness* of the sophisticatedly manipulated key on the intermediate state of the byte matrix

- effects of the superimposition can be *correctly* undone by applying these XOR operations with the same key arguments

- round keys are inductively computed by employing complex algebraic operations, while at the same time achieving an acceptable efficiency

- for the given block length and key length of 128 bits each (or suitably adapted for other possible lengths), the initial 4×4 byte matrix for the key k given as input is *expanded* into a $4\times(1+10)\cdot 4$ byte matrix, i.e., for each of the 10 rounds, four new columns are generated and taken as the *round key*

AES: key expansion

- the *key expansion* scheme distinguishes between
  the first column of a new round key and the remaining columns,
  but each column i is defined in terms of the
  - corresponding column $i-4$ of the preceding round key
  - the immediately preceding column $i-1$

- remaining columns:
  the column i is computed by directly applying the bitwise *XOR operation*

- first column:
  the preceding column is first transformed by a non-linear function
  that is a suitable composition of
  - the bytewise application of the substitution function S_{RD}
  - a permutation that shifts the positions in a column
  - the addition of a round constant

AES: decryption

- there is a *straightforward decryption algorithm*:
  basically, it performs the inverses of all byte matrix transformation
  in reverse order, employing the round keys accordingly

- the design also includes an equivalent decryption algorithm:
  it maintains the sequence of steps within a round,
  replacing the steps by their respective inverses

AES: efficiency

- NIST requirements:
  successor of DES should enable an efficiently implementation on *smartcards*, which could, for example, be used as *personal computing devices*

- the Rijndael proposal:
  the community was convinced regarding efficiency for implementations in both hardware and software

- the construction as a whole:
  high efficiency is enabled even though it operates on structures consisting of 128 bits (or even more)

- in combination with some block mode:
  transmission rates are suitable for large multimedia objects

- like any other symmetric block cipher:
  usage as part of a *hybrid encryption method* is possible

Stream ciphers using block modes

- *underlying block cipher*
  encrypts plaintext blocks and decrypts ciphertext blocks of a fixed length l_B

- *fragmentation*
  - divides a longer message into appropriate *fragments*
  - treats the resulting stream of fragments
    by using the block cipher in what is known as
    a *block mode* (*mode of operation*)

Two basic approaches to fragmentation

- (1) the original message is divided into fragments of
  length equal to exactly the block length l_B of the underlying block cipher

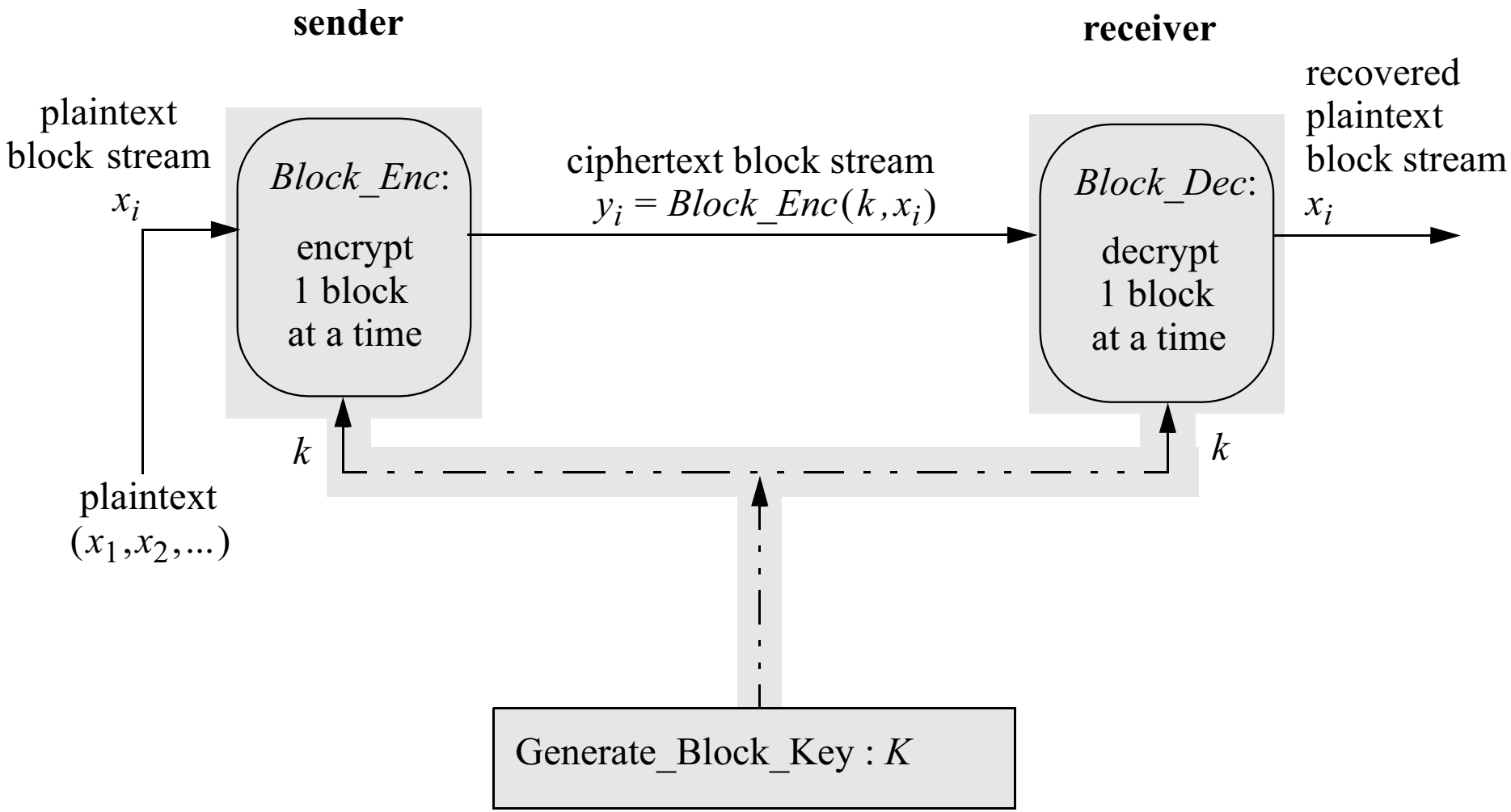
  (2) the block cipher treats the fragments
  - either separately (electronic codebook)
  - or in a suitably chained way (cipher block chaining)

- (1) the original message is divided into fragments
  of length $l \leq l_B$ (typically, $l = 1$ or $l = 8$)
  such that a plaintext stream of bits or bytes results

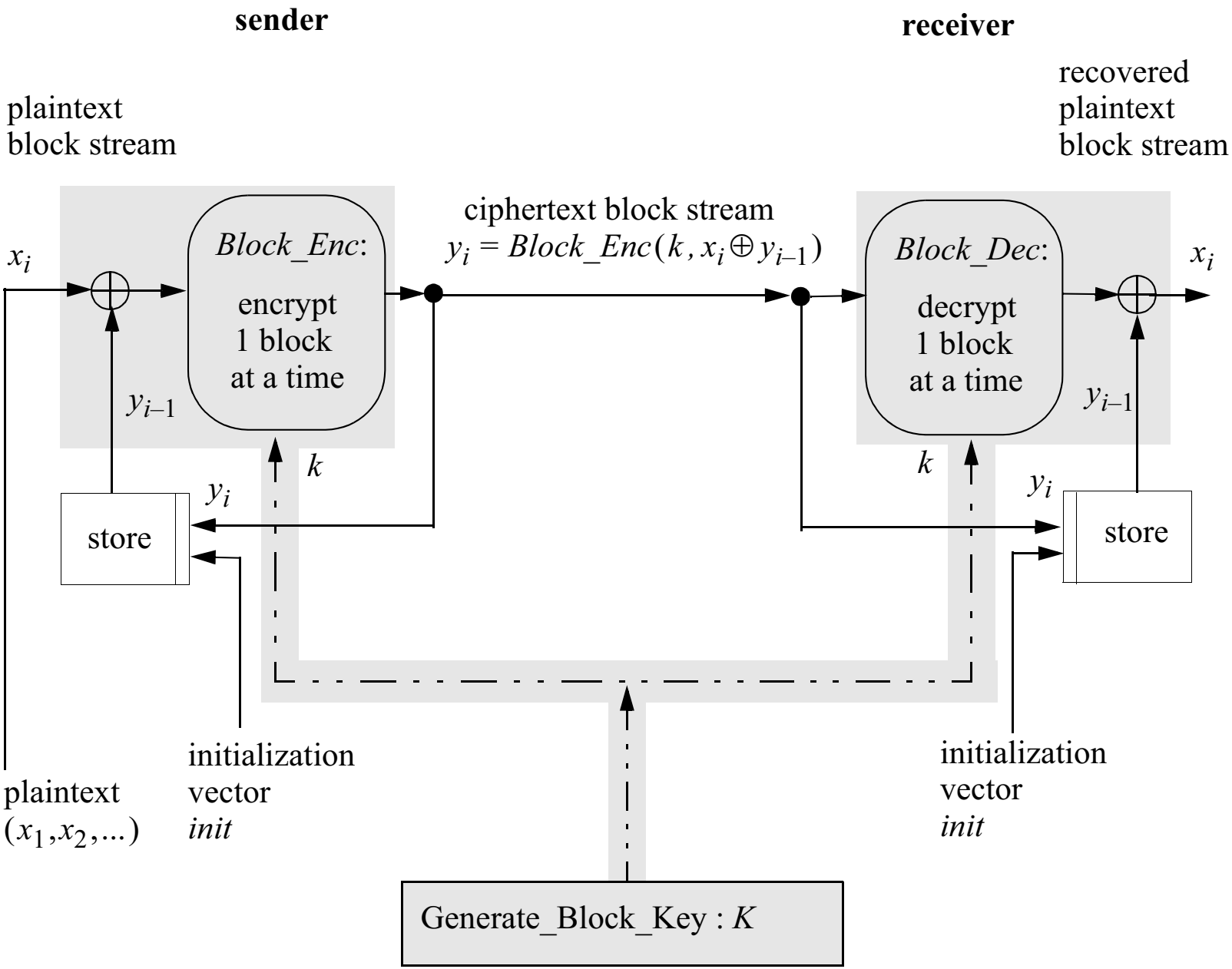
  (2) the underlying block cipher is used
  to generate a corresponding (apparently pseudorandom) cipher key stream
  that is superimposed on the plaintext stream
  by using the *XOR operation*
  (cipher feedback, output feedback, counter-with-cipher-block-chaining)

  can be seen as a variant of the *one-time key* encryption mechanism,
  where *perfectness* is abandoned for the sake of
  a reusable, short key as demanded by the underlying block cipher

Electronic Codebook (ECB) Mode



Cipher Block Chaining (CBC) Mode



CBC: correctness

- **encryption** algorithm *Enc*:
  - for the first block x_1,

    $Enc(k, x_1) := Block\_Enc(k, x_1 \oplus init)$
  - for all further blocks x_i with $i > 1$,

    $Enc(k, x_i) := Block\_Enc(k, x_i \oplus Enc(k, x_{i-1}))$

- **decryption** algorithm *Dec*:
  - for $i = 1$,

    $Dec(k, y_1) := Block\_Dec(k, y_1) \oplus init$

    $= Block\_Dec(k, Block\_Enc(k, x_1 \oplus init)) \oplus init$

    $= (x_1 \oplus init) \oplus init = x_1$
  - for $i > 1$,

    $Dec(k, y_i)$

    $:= Block\_Dec(k, y_i) \oplus y_{i-1}$

    $= Block\_Dec(k, Enc(k, x_i)) \oplus Enc(k, x_{i-1})$

    $= Block\_Dec(k, Block\_Enc(k, x_i \oplus Enc(k, x_{i-1}))) \oplus Enc(k, x_{i-1})$

    $= (x_i \oplus Enc(k, x_{i-1})) \oplus Enc(k, x_{i-1}) = x_i$

CBC: producing a message digest

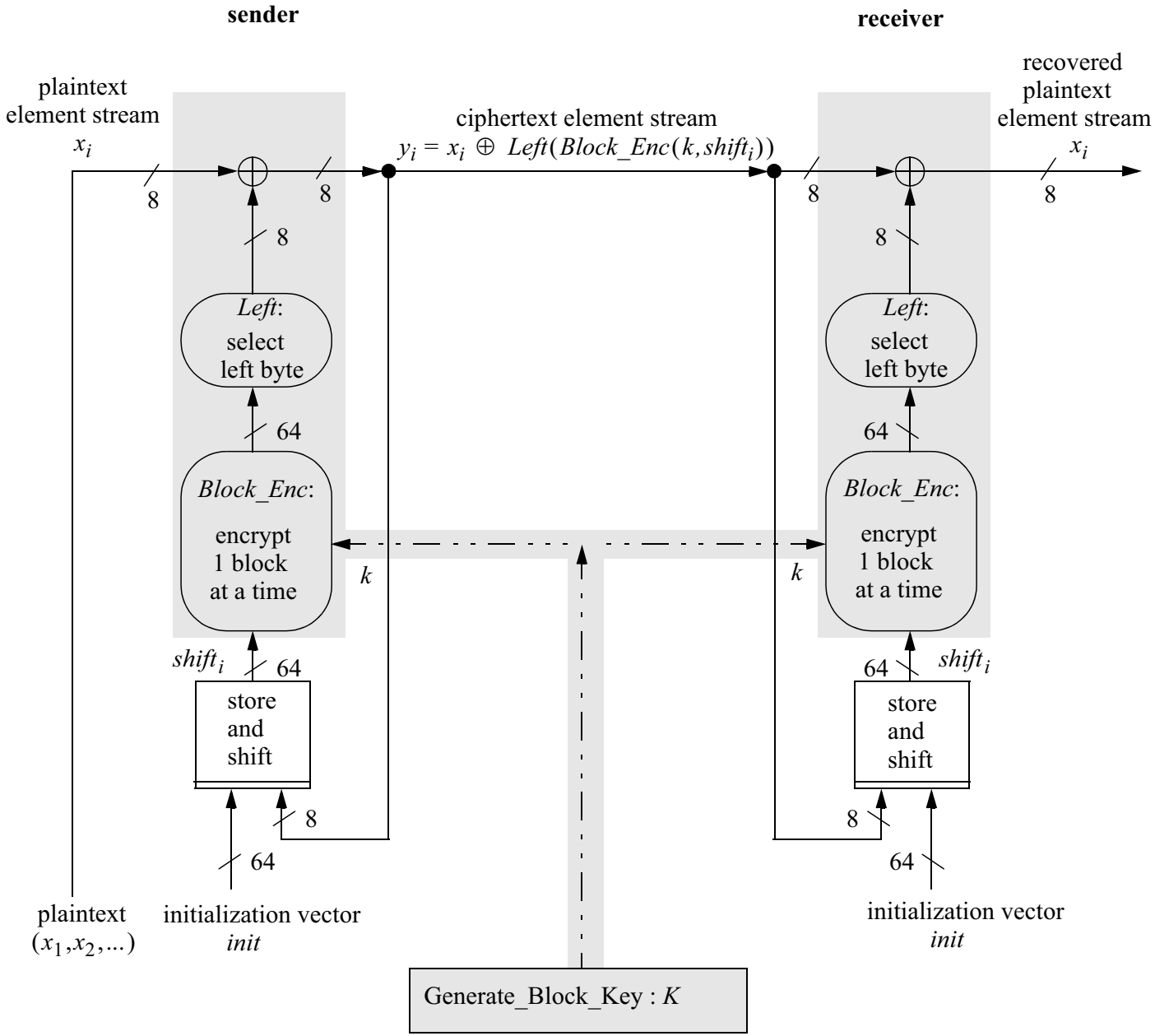
- characteristic feature of the cipher block chaining mode:
  all blocks are treated in a connected way
  requiring strict serialization

- the last resulting ciphertext block seen as a *message digest*:
  this block can be employed as a
  piece of *cryptographic evidence* (a *cryptographic exhibit*)
  for an *authenticity verification* algorithm

Cipher Feedback (CFB) Mode

- follows the second basic approach,
  achieving a variant of the one-time key encryption mechanism
- generates the required pseudorandom *cipher key* stream
  by means of the encryption algorithm *Block_Enc*(*ryption*)
  of the underlying block cipher
- does not employ the corresponding decryption algorithm,
  and thus *cannot* be used for an asymmetric block cipher
- extracts the cipher key stream from the outputs of the block cipher encryption,
  whose inputs are taken as a feedback from the ciphertext stream
- uses an *initialization vector init* as a *seed*,
  which must be used only once
  but can be communicated to the receiver without protection
- example: fragment length $l = 8$
  block size of the underlying block cipher $l_B = 64$

CFB: overall structure



CFB: correctness

- **encryption** algorithm *Enc*:
  for each plaintext byte x_i,
  $Enc(k, x_i) := x_i \oplus Left(Block\_Enc(k, shift\_sender_i))$.

- **decryption** algorithm *Dec*
  for each ciphertext byte y_i,

$$\begin{aligned} Dec(k, y_i) &:= y_i \oplus Left(Block\_Enc(k, shift\_receiver_i)) \\ &= (x_i \oplus Left(Block\_Enc(k, shift\_sender_i))) \\ &\quad \oplus Left(Block\_Enc(k, shift\_receiver_i)) \\ &= x_i, \end{aligned}$$

  provided $shift\_sender_i = shift\_receiver_i$

- required equality of the $shift_i$ inputs on both sides is achieved
  by using the same initialization vector *init* and then, inductively,
  by employing the same operations and inputs to generate them

CFB: producing a message digest

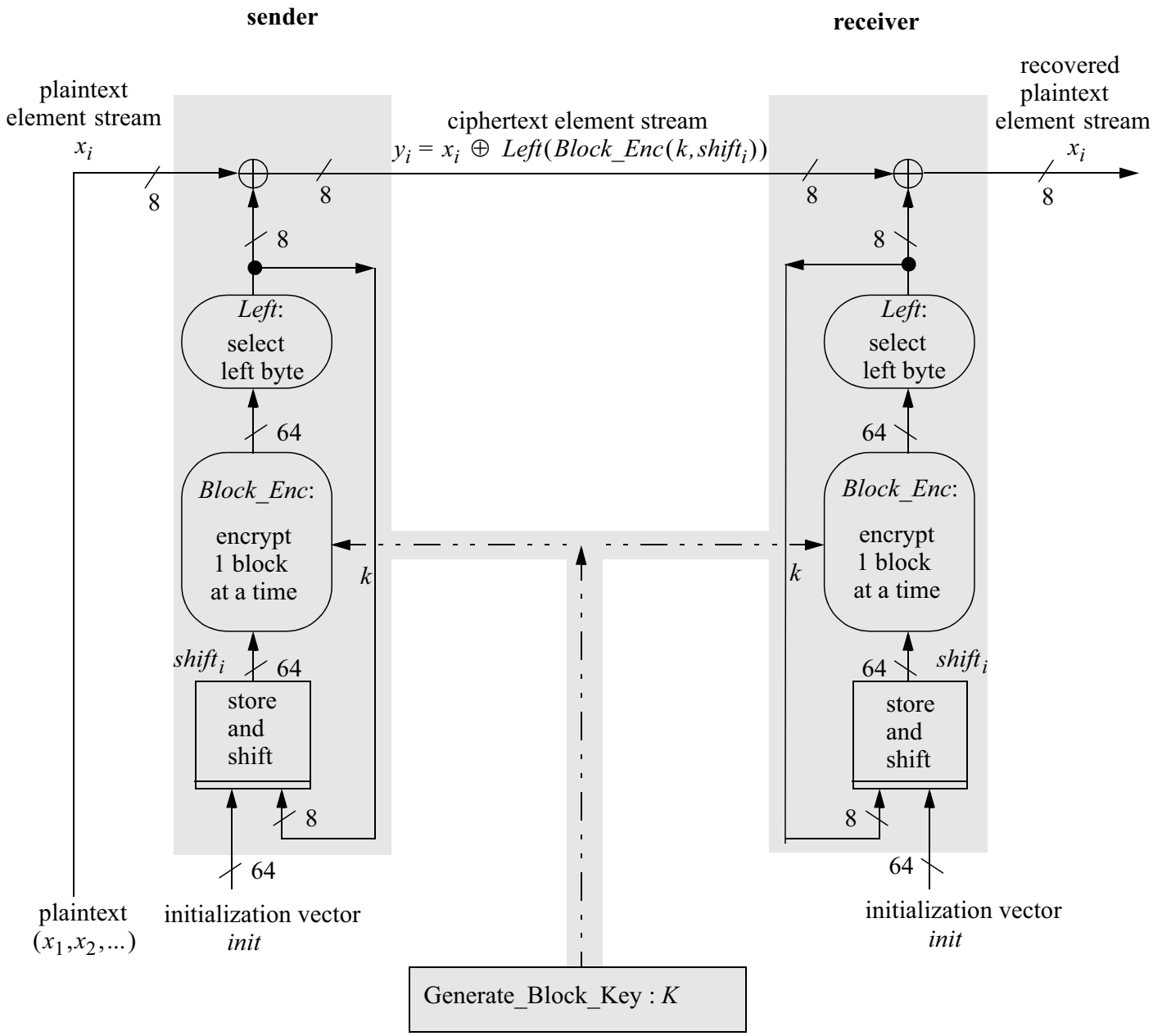
- characteristic feature of the cipher feedback mode:
  the last resulting ciphertext block
  depends potentially on the full plaintext stream

- the last resulting ciphertext block seen as a *message digest*:
  this block can be employed as a
  piece of *cryptographic evidence* (a *cryptographic exhibit*)
  for an *authenticity verification* algorithm

Output Feedback (OFB) Mode

- follows the second basic approach
- required pseudorandom *cipher key* stream is generated
  as for the cipher feedback mode, except of the following
- the block cipher encryption takes the feedback directly from its own outputs
- since only the encryption algorithm of the underlying block cipher is involved,
  this mode cannot be used for an asymmetric block cipher
- example:
  - fragment length: $l = 8$
  - block size of the underlying block cipher: $l_B = 64$

Output Feedback (OFB) Mode: overview



Counter-with-Cipher-Block-Chaining Mode (CCM)

- generates a pseudorandom *cipher key* stream
  by encrypting a sequence of *counters* $count_i$
  using the underlying block encryption
- computes the counters by

$$count_i := init + i \bmod 2^{l_B},$$

  assuming a block size l_B of the block cipher and
  taking an *initialization vector init* of that size
- cannot be used for an asymmetric block cipher
- exploits that for each $i = 1, 2, \ldots$:
  - the pair of the counter $count_i$ and the corresponding plaintext block x_i can be treated independently of all other pairs, as for ECB
  - the counter $count_i$ is independent of the ciphertext stream (and thus of the plaintext stream), as for OFB

- achieves *authenticated encryption*:
  - additionally performs CBC encryption
    without transmitting the resulting ciphertext blocks
  - superimposes the last resulting CBC ciphertext block y_{fin}
    on the counter $count_0 = init$
  - appends the resulting block $y_{fin} \oplus count_0$ as a *message digest*

Features of block modes

- *initialization vector*:
  - some computational overhead is necessary
  - a *parameterization* of the encryption is achieved:
    if the initialization vector is varied for identical messages and kept secret,
    then the encryption could even be seen as *probabilistic*

- *fault tolerance*, for the sake of *availability*:
  propagation of a *modification error* is considered:
  - in the *plaintext* stream
  - during transmission, in the *ciphertext* stream:
    - all modes recover shortly after a modification error
    - OFB and CCM even behave optimally
      (only the error position is affected)

- *modification error* in the *plaintext* stream:
  - ECB, OFB and the main part of CCM recover
    shortly after the error position or totally prevent propagation
  - for CBC, CFB and the digest production part of CCM,
    an error might “diffuse” through the full succeeding cipher stream:
    accordingly, the resulting final cipher block can be seen as a
    message digest and can thus be employed as
    a piece of *cryptographic evidence* (a *cryptographic exhibit*)

- *synchronization errors* owing to *lost fragments*:

  for all modes, additional measures must be employed, e.g.,
  by suitably inserting separators at agreed fragment borders

Rudimentary comparison of block modes

	ECB	CBC	CFB	OFB	CCM
Initialization vector/ parameterization	no	yes	yes	yes	yes
Propagation of error in plaintext fragment	limited to block	unlimited up to end of stream	unlimited up to end of stream	limited to error position	limited to error position, except for superimposed last CBC cipher block
Suitable for producing a message digest	no	by last cipher block	by last cipher block	no	by superim-posed last CBC cipher block
Propagation of error in ciphertext fragment	limited to block	limited to block and succeeding block	limited to block and succeeding block	limited to error position	limited to error position

Some rough advice to a security administrator

- **electronic codebook mode**
  is suitable for short, randomly selected messages
  such as nonces or cryptographic keys of another mechanism

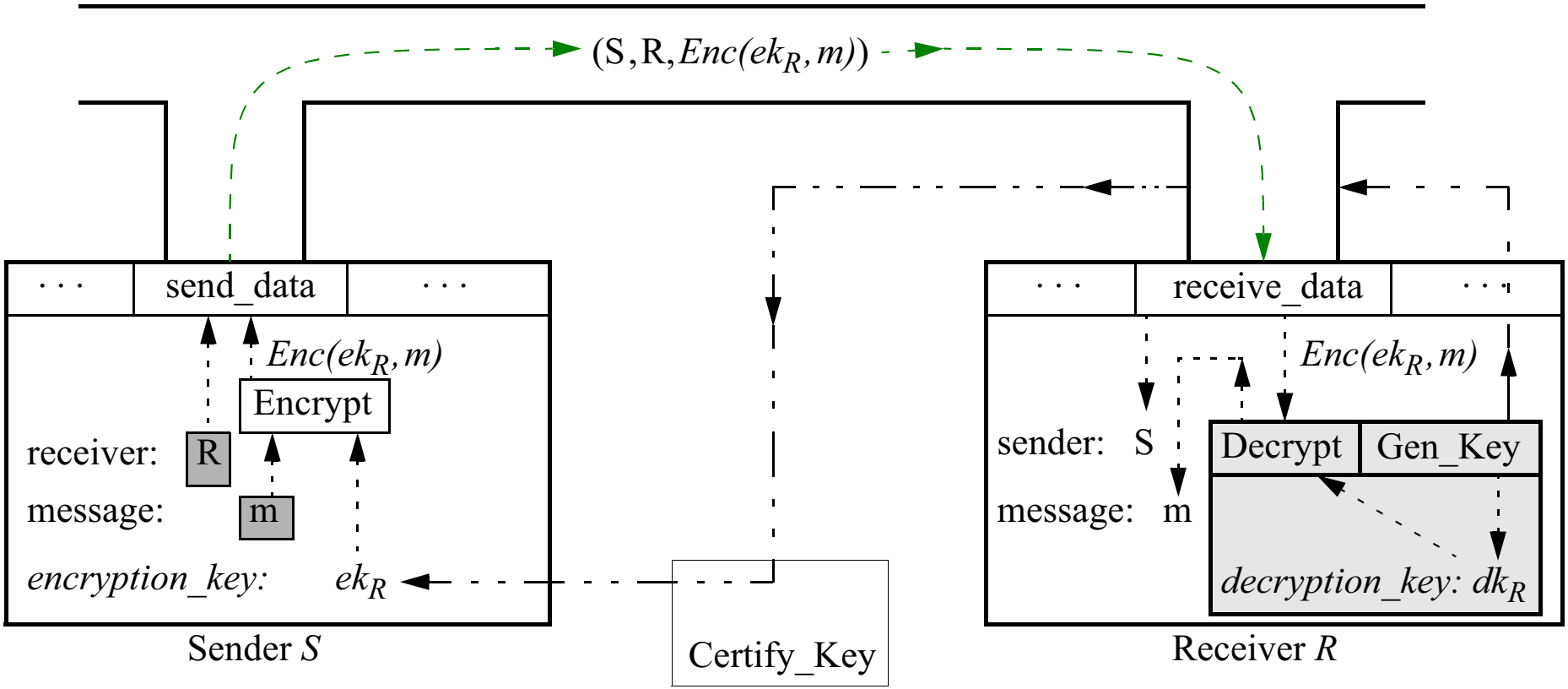
- **cipher block chaining mode**
  might be employed for long files with any non-predictable content

- **cipher feedback mode, output feedback mode and counter mode**
  support the transmission of a few bits or bytes,
  e.g., as needed for connections between a central processing unit
  and external devices such as a keyboard and monitor

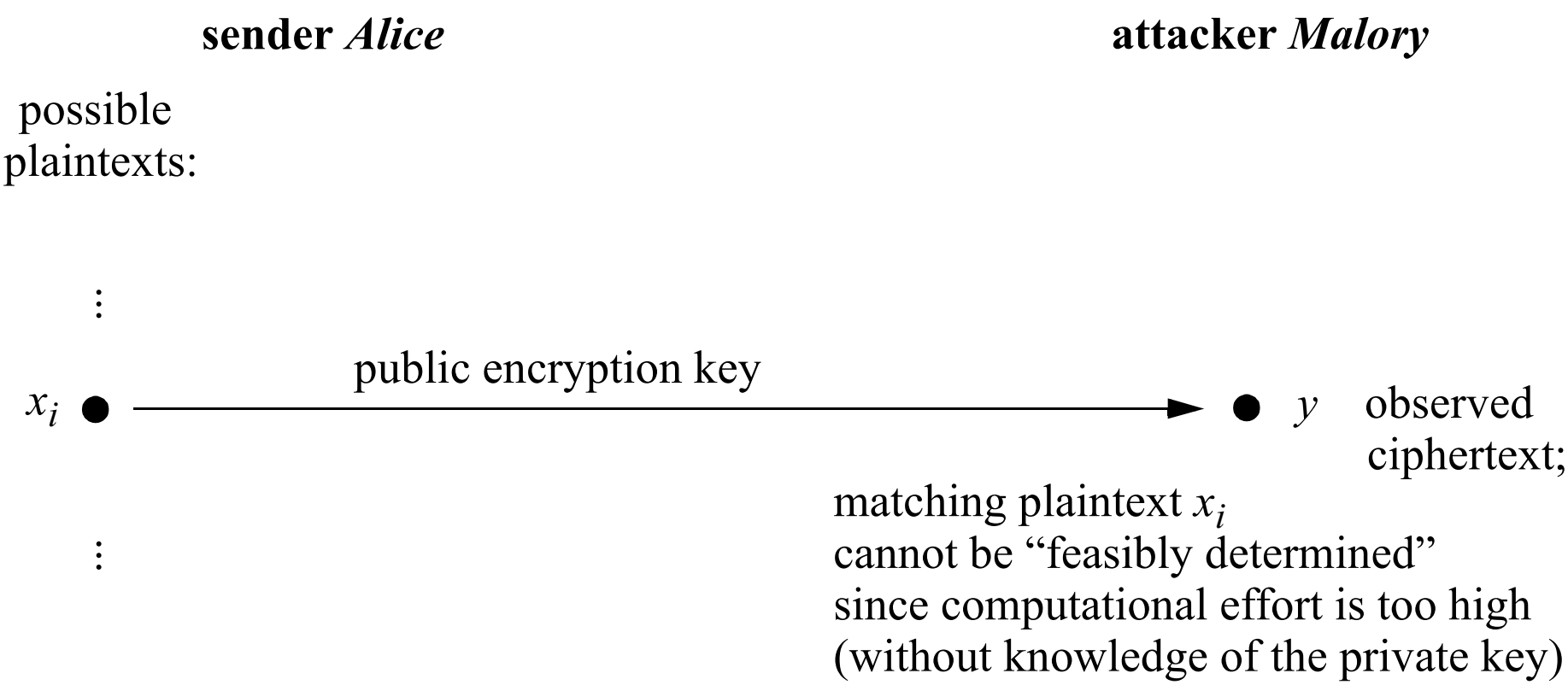
- **output feedback mode and counter mode**
  might be preferred for highly failure-sensitive applications,
  since modification errors are not propagated at all
  (except for the added message digest)

14 Asymmetric Encryption and Digital Signatures with RSA

Asymmetric encryption



Complexity-theoretic secrecy property (one-way function approach)



Family of one-way functions with trapdoors

parameterized family of functions f_k such that for each k:

- function $f_k : D_k \rightarrow R_k$
  is *injective* and computable in *polynomial time*

- inverse function $f_k^{-1} : R_k \rightarrow D_k$
  is *computationally infeasible* without a knowledge of k

- inverse function $f_k^{-1} : R_k \rightarrow D_k$
  is computable in polynomial time
  if k (the private key) is used as an additional input

it is an outstanding *open problem* of computer science,
closely related to the open problem of whether $P \neq NP$,
whether such families actually exist

RSA functions

- an *RSA function* $RSA^{n,\,e}_{p,\,q,\,d}$ is a number-theoretic function where
  - (p,q,d) is used as the *private key*
  - (n,e) as the *public key*

- the designated *secret holder* generates, *randomly* and *confidentially*, two different, sufficiently large *prime numbers* p and q

- $n := p \cdot q$
  is published as the modulus of the ring $(\mathbf{Z}_n,+,\cdot,0,1)$:
  - all computations are performed in this ring
  - the multiplicative group is formed by those elements that are relatively prime to the modulus n, i.e.,

    $\mathbf{Z}_n^* = \{\, x \mid 0 < x < n \text{ with } gcd(x,n) = 1 \,\}$
  - this group has a cardinality $\phi(n) = (p-1)\cdot(q-1)$
  - *Euler phi function* ϕ,
    is used for investigating properties of exponents for exponentiations

- the designated secret holder *randomly* selects
  the second component e of the *public key* such that
  $1 < e < \phi(n)$ and $gcd(e, \phi(n)) = 1$
- additionally, the designated secret holder *confidentially* computes
  the third component d of the *private key*
  as the multiplicative inverse of e modulo $\phi(n)$:
  $1 < d < \phi(n)$ and $e \cdot d \equiv 1 \bmod \phi(n)$
  - in principle, multiplicative inverses can be efficiently computed
  - in this specific situation a knowledge of $\phi(n)$ is needed,
    which requires one to know the secretly kept prime numbers p and q

- the *RSA function* for the selected parameters is defined by
  $RSA^{n,e}_{p,q,d} : \mathbf{Z}_n \to \mathbf{Z}_n$ with
  $RSA^{n,e}_{p,q,d}(x) = x^e \bmod n$
  - can be computed by whoever knows the public key (n, e)
  - the required properties of
    injective one-way functions with trapdoors (are conjectured to) hold

Injectivity and trapdoor: theorem

in the setting of the *RSA function* $RSA^{n,e}_{p,q,d}$,

for all $x \in \mathbf{Z}_n$,

$$(x^e)^d \equiv x \bmod n$$

Injectivity and trapdoor: sketch of proof

the following congruences modulo n are valid for all $x \in \mathbf{Z}_n$:

$$
\begin{aligned}
(x^e)^d &\equiv x^{e \cdot d} && \text{exponentiation rules} \\
&\equiv x^{k \cdot \phi(n) + 1} && e \cdot d = k \cdot \phi(n) + 1\text{, definition of } d \\
&\equiv x \cdot (x^{\phi(n)})^k && \text{exponentiation rules}
\end{aligned}
$$

Case 1, $x \in \mathbf{Z}_n^*$:

multiplicative group $\mathbf{Z}_n^*$ has order $\phi(n)$: $(x^{\phi(n)})^k \equiv 1^k \equiv 1 \bmod n$

thus: $(x^e)^d \equiv x \bmod n$

Case 2, $x \notin \mathbf{Z}_n^*$:

case assumption: $gcd(x,n) \neq 1$

n product of prime numbers p and q: $gcd(x,p) \neq 1$ or $gcd(x,q) \neq 1$

show for each subcase: $(x^e)^d \equiv x \bmod p$ and $(x^e)^d \equiv x \bmod q$

by the definitions of n, p and q

and *Chinese remainder theorem*: $(x^e)^d \equiv x \bmod n$

Subcase 2a

$gcd(x,p) \neq 1$:

p is prime: p divides x and
thus any multiple of x as well

hence: $(x^e)^d \equiv x \bmod p$

similarly:

$gcd(x,q) \neq 1$ implies

$(x^e)^d \equiv x \bmod q$

Subcase 2b

gcd(x,p) = 1:

then $x \in \mathbf{Z}_p^*$ and, accordingly,
the following congruences modulo p are valid:

$$
\begin{aligned}
x^{\phi(n)} &\equiv x^{(p-1)\cdot(q-1)} && \text{definition of } \phi(n) \\
&\equiv (x^{p-1})^{q-1} && \text{exponentiation rules} \\
&\equiv 1^{q-1} \equiv 1 && x \in \mathbf{Z}_p^* \text{ has order } \phi(p) = p-1
\end{aligned}
$$

as in Case 1, we then obtain the following congruences modulo p:

$$
\begin{aligned}
(x^e)^d &\equiv x^{e\cdot d} && \text{exponentiation rules} \\
&\equiv x^{k\cdot\phi(n)+1} && e\cdot d = k\cdot\phi(n)+1\text{, definition of } d \\
&\equiv x\cdot(x^{\phi(n)})^k && \text{exponentiation rules} \\
&\equiv x\cdot 1 \equiv x && \text{congruence shown above}
\end{aligned}
$$

similarly:

$gcd(x,q) = 1$ implies

$(x^e)^d \equiv x \bmod q$

Factorization conjecture of computational number theory

the factorization problem
restricted to products of two prime numbers, i.e.,

given a number n of known form $n = p \cdot q$
where p and q are prime numbers,

to determine the actual factors p and q,
is computationally infeasible

RSA conjecture

if the non-keyed inversion problem for RSA functions
was computationally feasible,

then the factorization problem
would be computationally feasible as well

specialized RSA conjecture

if the non-keyed inversion problem for RSA functions
by means of determining the private exponent d from an argument–value pair
was computationally feasible,

then the factorization problem
would be computationally feasible as well

RSA conjecture and further conjectures

- RSA conjectures roughly says:
  "factorization" is *feasibly reducible* to "RSA inversion"

- the converse claim, namely:
  "RSA inversion" is feasibly reducible to "factorization",
  provably holds:
  if an "attacker" was able to feasibly factor the public modulus n
  into the prime numbers actually employed,
  then he could feasibly determine the full private key
  by just repeating the computations of the designated secret holder

Some similar proven claims

"factorization" is feasibly reducible to any of the following problems, and vice versa:

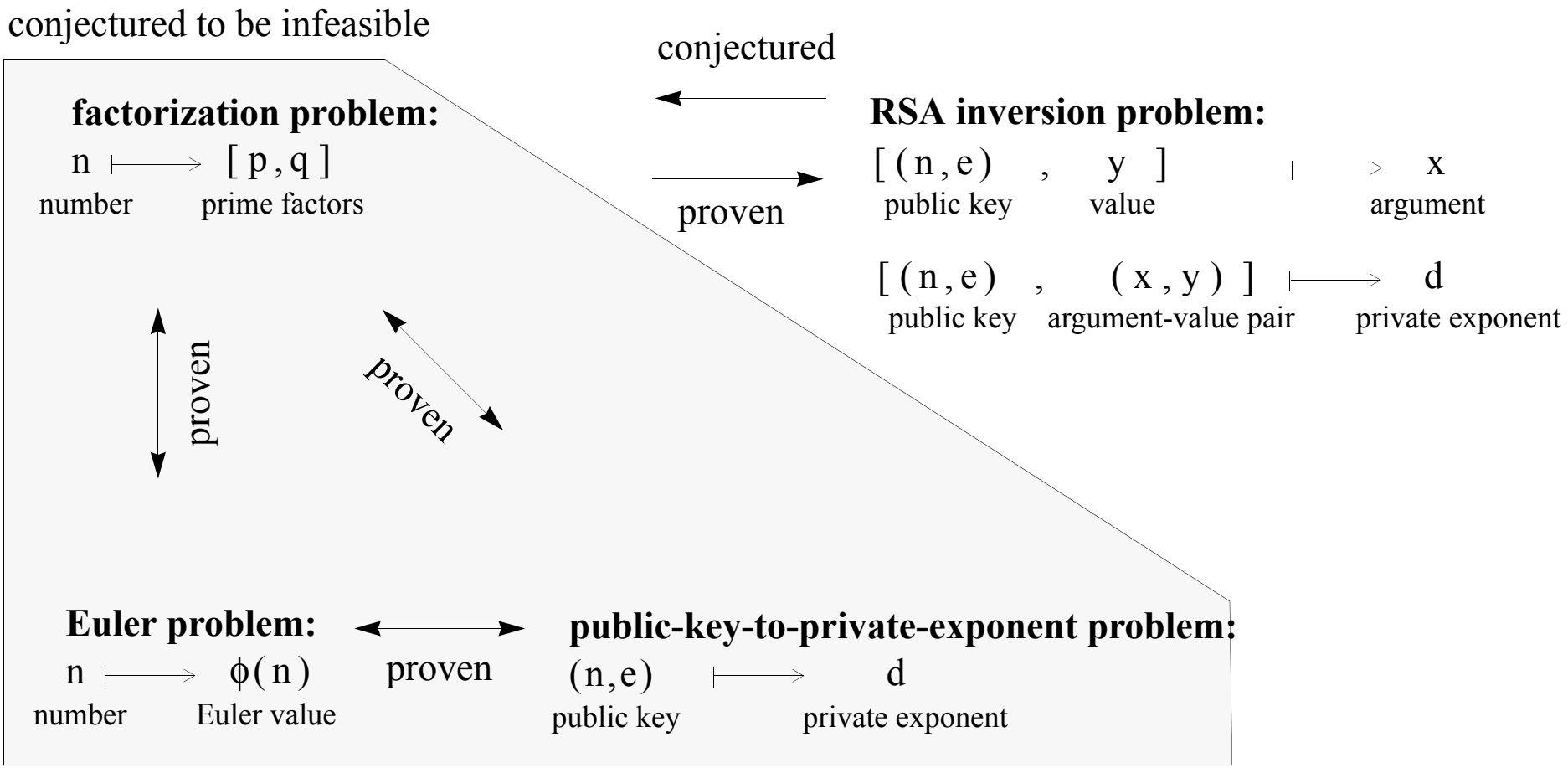
- ***Euler problem*:**
  given a number n of known form $n = p \cdot q$,
  where p and q are prime numbers,

  to determine the value $\phi(n)$

- ***public-key-to-private-exponent problem*:**
  given the public key (n, e),

  to determine the private exponent d

Conjectures and proven claims about feasible reducibility



RSA asymmetric block cipher

- is an example of the *one-way function* approach
- is based on RSA functions and their properties
- is *asymmetric*, admitting *multiple* key usage
- operates *blockwise*, where the block length is determined by the parameters of the underlying RSA function
- achieves *complexity-theoretic security*, provided:
  - the *factorization conjecture* and the *RSA conjecture* hold
  - the key is properly generated and sufficiently long
  - some additional care is taken

RSA encryption: protocol outline

- **key generation:**
  selecting a *private key* (p,q,d) and a *public key* (n,e) for $RSA^{n,\,e}_{p,\,q,\,d}$

- **preprocessing of** a message m, using an agreed *hash function*:
  - adding a nonce *non* (for *probabilistic encryption*)
  - adding the hash value $h(m,non)$ (for *authenticated encryption*)

- **encryption:** computing $y = x^e \bmod n$
  for $x = (m,non,h(m,non))$,
  if interpretable as a positive number less than n

- **decryption**: computing $y^d \bmod n$
  for received message y

- **postprocessing** of the decryption result:
  - extracting the three components
  - recomputing the hash value of the first two components
  - comparing this hash value with the third component (received hash value):
    if the received hash value is verified,
    the first component is returned as the (presumably) correct message

RSA encryption: underlying sets

for each fixed setting of an RSA function $RSA^{n,\,e}_{p,\,q,\,d}$:

- **plaintexts:**
  bit strings over the set $\{0,1\}$
  of some fixed length $l_{mes} \leq \text{ld}\ n$

- **ciphertexts:**
  bit strings over the set $\{0,1\}$,
  basically of length ld n
  (binary representation of a positive number less than n (residue modulo n))

- **keys:**
  given the public key (n,e),
  in principle there is a unique residue modulo n
  that can be used as the private decryption exponent d,
  whose binary representation is a bit string,
  basically of length ld n or less
  (from the point of view of the nondistinguished participants,
  this decryption exponent cannot be "determined")

RSA: key generation Gen

- selects a *security parameter l*
  that basically determines the length of the key
- generates randomly two large prime numbers p and q
  of the length required by the security parameter
- computes the modulus $n := p \cdot q$
- selects randomly an encryption exponent e
  that is relatively prime to $\phi(n) = (p-1) \cdot (q-1)$
- computes the decryption exponent d as the
  solution of $e \cdot d \equiv 1 \bmod \phi(n)$

RSA: encryption algorithm Enc

- takes a possibly padded message m of length l_{mes} as a plaintext
- generates a random bit string non as a nonce of length l_{non}
- computes a hash value $h(m, non)$ of length l_{hash}
- concatenates these values with appropriate separators:
  the resulting bit string x must, basically, have length ld n
  ($l_{mes} + l_{non} + l_{hash} \leq$ ld n,
  binary representation of a positive number less than n (residue modulo n))
- taking the public key (n, e),
  computes and returns the ciphertext
  $y = x^e \bmod n$

RSA: decryption algorithm Dec

- taking the first component n of the public key (n, e) and the third component d of the private key (p, q, d), inverts the given ciphertext y by computing

  $x' = y^d \bmod n$

- decomposes the result x' into
  - message part m'
  - nonce part non'
  - hash value part $hash'$

  according to the separators employed

- inspects the received hash value:
  - if $h(m', non') = hash'$, then m' is returned as the (supposedly) correct message
  - otherwise, an error is reported

RSA: fundamental properties

- to be considered: *correctness*, *secrecy* and *efficiency*
- the *modulus n* should have a length of at least 1024;
  even a larger length might be worthwhile to resist dedicated attacks
- there is a trade-off between secrecy and
  efficiency, roughly estimated:
  - *key generation* consumes time $O((\mathrm{ld}\ n)^4)$
  - operations of *modular arithmetic*, needed for *encryption* and *decryption*,
    consume time at most $O((\mathrm{ld}\ n)^3)$
- high performance can be achieved in practice
  by employing specialized algorithms for both software and hardware
- there are some known weaknesses of specific choices of the parameters
- preprocessing and postprocessing are necessary:
  - *probabilistic encryption* demanded for sophisticated secrecy property
  - *added nonce* needed for several purposes

RSA: added nonce

- enlarges the search space for the straightforward *inversion algorithm* that an attacker could use
  given a ciphertext and the public key

- prevents a known *ciphertext/plaintext* vulnerability,
  by ensuring that a given plaintext *m*
  will produce different ciphertexts when being sent multiple times

RSA: authenticated encryption

- needed to prevent active attacks enabled by the *multiplicativity property* (*homomorphism property*) of exponentiation:

  for all x, y and w: $(x \cdot y)^w = x^w \cdot y^w$,

  which is inherited by any RSA function

- example of an *attack to decrypt* an observed ciphertext y:
  - select a multiplicatively invertible element $u \in \mathbf{Z}_n^*$
  - compute $t := y \cdot u^e \mod n$, by employing the public key (n, e)
  - somehow succeed in presenting t as a (harmless-looking) ciphertext to the holder of the private key and obtain

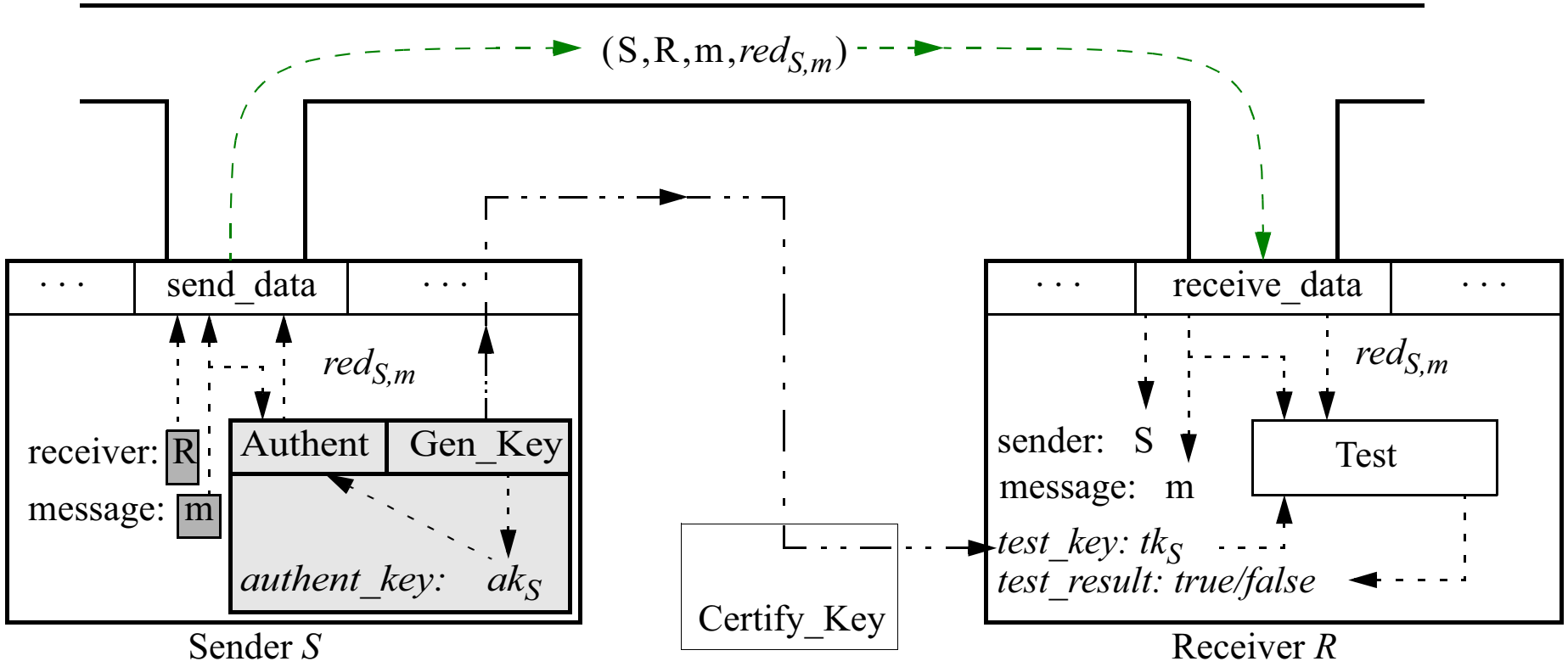
    the corresponding plaintext t^d with property

    $t^d \equiv (y \cdot u^e)^d \equiv y^d \cdot u^{e \cdot d} \equiv y^d \cdot u \mod n$
  - solve the congruence for the wanted value y^d by computing

    $y^d = t^d \cdot u^{-1} \mod n$

- this attack will not succeed with the employment of a hash function, provided this hash function does not suffer from the same multiplicativity property

Asymmetric authentication (digital signing)



RSA asymmetric digital signatures

- is an example of the *one-way function* approach
- is based on RSA functions and their properties
- is *asymmetric*, admitting *multiple* key usage
- achieves *complexity-theoretic security*, provided:
  - the *factorization conjecture* and the *RSA conjecture* hold
  - the key is properly generated and sufficiently long
  - some additional care is taken
- is obtained by exchanging the roles of encryption and decryption, given a suitable *RSA function* $RSA^{n,e}_{p,q,d}$ with
  - *private key* (p,q,d)
  - *public key* (n,e),

RSA digital signatures: protocol outline

- **preprocessing** of a message m using an agreed *one-way hash function*:
  computing a hash value $h(m)$

- **authentication:**
  computing the "RSA decryption" of the hash value

  $$red = h(m)^d \bmod n,$$

- **verification:**
  - computing the "RSA-encryption" of the cryptographic exhibit

    $red^e \bmod n$

    to recover the presumable hash value
  - comparing the result
    with the freshly recomputed hash value of the received message m

RSA digital signatures: underlying sets

- **messages:**
  bit strings over the set $\{0,1\}$
  that can be mapped by the agreed one-way hash function h
  to bit strings basically of length ld n
  (positive numbers less than n (residues modulo n))
- **cryptographic exhibits:**
  bit strings over the set $\{0,1\}$,
  basically of length ld n
  (positive numbers less than n (residues modulo n))
- **keys:**
  given the public key (n,e),
  in principle there is a unique residue modulo n
  that can be used as the private decryption exponent d,
  whose binary representation is a bit string, basically of length ld n or less;
  (from the point of view of the nondistinguished participants,
  this decryption exponent cannot be "determined")

RSA digital signatures: three algorithms

- ***key generation* algorithm *Gen*:**
  same as for RSA encryption

- ***authentication* (*signature*) algorithm *Aut*:**
  - takes a message m of an appropriate length
  - computes $h(m)$, where h is an agreed *one-way hash function*
  - returns $red = h(m)^d \bmod n$

- ***verification* algorithm *Test*:**
  - takes the received cryptographic exhibit red
  - computes $hash := red^e \bmod n$
  - takes the received message m
  - determines its hash value $h(m)$
  - checks whether this (correct) hash value equals the (received) value $hash$:

    $Test((n,e),m,red)$ returns *true* iff $h(m) = red^e \bmod n$

RSA digital signatures: fundamental properties

- to be considered: *correctness*, *unforgeability* and *efficiency*
- basic aspects of these properties can be derived like for RSA encryption
- regarding *correctness*:
  the commutativity of multiplication and exponentiation, i.e.,
  for all b, e_1, e_2:

$$(b^{e_1})^{e_2} = b^{e_1 \cdot e_2} = b^{e_2 \cdot e_1} = (b^{e_2})^{e_1},$$

  is inherited by
  - encryption function $x^e \bmod n$
  - decryption function $y^d \bmod n$
- these functions are mutually inverse,
  independent of the application order

RSA encryption and digital signatures

- any *commutative* (asymmetric) *encryption* mechanism
  with encryption algorithms *Enc* and *Dec* that satisfy,

  for all plaintexts or ciphertexts x and for all keys (ek, dk)

  $$Dec(dk, Enc(ek, x)) = Enc(ek, Dec(dk, x))$$

  can be converted into an *authentication* (*signature*) *mechanism*

- **authentication:** $Aut(dk, x) = Dec(dk, x)$,
  using the private decryption key *dk* as the authentication key

- **verification:** $Test(ek, x, red) = true$ iff $x = Enc(ek, red)$,
  using the public encryption key *ek* as the test key

- *correctness* of the authentication
  is implied by the encryption correctness:
  $Enc(ek, Aut(dk, x)) = Enc(ek, Dec(dk, x)) = Dec(dk, Enc(ek, x)) = x$

- *unforgeability* is implied by the secrecy of the encryption

ElGamal asymmetric block cipher

- is another well-known example of the *one-way function* approach
- is based on ElGamal functions and their properties
- is *asymmetric*, admitting *multiple* key usage
- operates *blockwise*, where the block length is determined by the parameters of the underlying ElGamal function
- achieves *complexity-theoretic security*, provided:
  - the *discrete logarithm conjecture* and the *ElGamal conjecture* hold
  - the key is properly generated and sufficiently long
  - some additional care is taken

Asymmetric block ciphers based on elliptic curves

- are increasingly important examples of the *one-way function* approach
- are based on generalized ElGamal functions that
  are defined over appropriately constructed finite cyclic groups
  derived from elliptic curves based on a finite field
- are *asymmetric*, admitting *multiple* key usage
- operate *blockwise*, where the
  block length is determined by the parameters of the underlying elliptic curve
- achieve *complexity-theoretic security*, provided:
  - the pertinent *discrete logarithm conjecture* and related conjectures hold
  - the key is properly generated and sufficiently long
  - some additional care is taken
- offer a large variety of alternatives to the still predominant RSA approach,
  and thus diminish the dependence on the special unproven conjectures
- promise to achieve the wanted degree of secrecy
  with improved efficiency in comparison with the RSA approach

Asymmetric authentication by ElGamal and elliptic curves

- similar to encryption

15 Some Further Cryptographic Protocol

Covert commitments

- ***committing*:**
  the *committer* discretionarily selects some value v_{com}
  and commits to this value, in a covert form regarding the *receiver*
- ***revealing*:**
  the *committer* reveals a value v_{show} to the *receiver*,
  who in turn either accepts or rejects it as the committed value

- ***binding property*** (combined *correctness* and *unforgeability* property)**:**

  for all values v_{com}:
  if the committer enters the revealing phase at all,
  then the receiver accepts the revealed value v_{show}
  if and only if it is the committed value v_{com}

- ***secrecy property*** (after committing and before revealing)**:**

  for all values v_{com},
  the receiver cannot "determine"
  the committed value v_{com} from the covert form

Secret sharing

- ***distributing*:**
  the *owner* of the secret v computes *shares* $s_1, \ldots, s_n$ and
  distributes them to appropriate *receivers*
- ***combining*:**
  for some threshold $t \leq n$, t (or more) *receivers* collect their shares $s_{i_1}, \ldots, s_{i_t}$
  and use them to recover the secret

- ***correctness property*:**

  for all values v:
  the receivers succeed in determining the secret value v
  from any set of t distinct shares $s_{i_1}, \ldots, s_{i_t}$

- ***secrecy property*:**

  for all values v:
  the receivers cannot "determine" the secret value v
  from any set of $t - 1$ shares

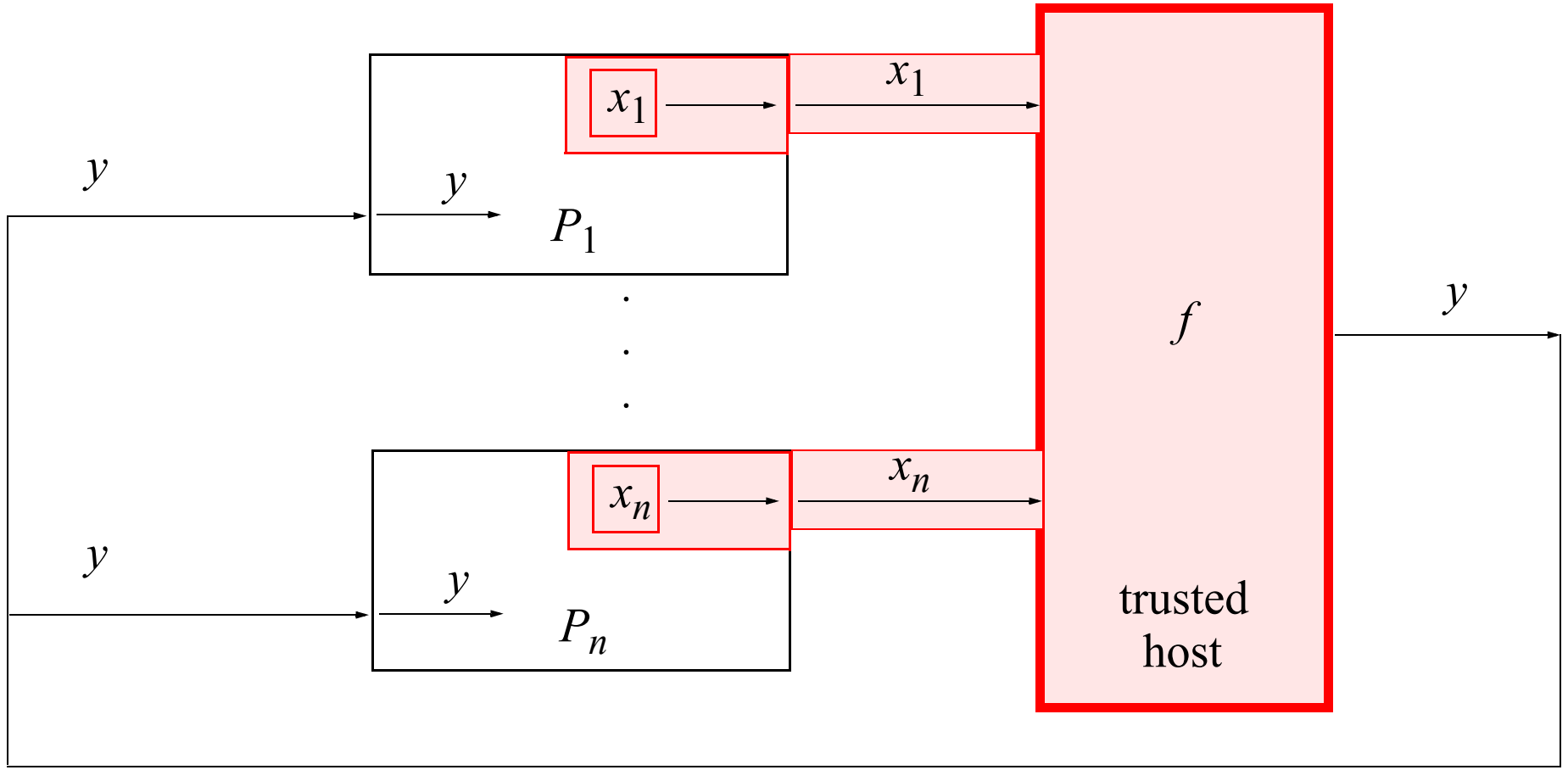
Multiparty computations

- multiparty computations address a very general situation of *cooperation in the presence of threats* between n parties P_i
- parties aim at jointly computing the value y of some agreed n-ary function f:
  - each P_i secretly provides an argument x_i
  - at the end, each P_i knows the computed value $y = f(x_1, \ldots, x_n)$
  - no P_i learns anything new about the other parties' arguments

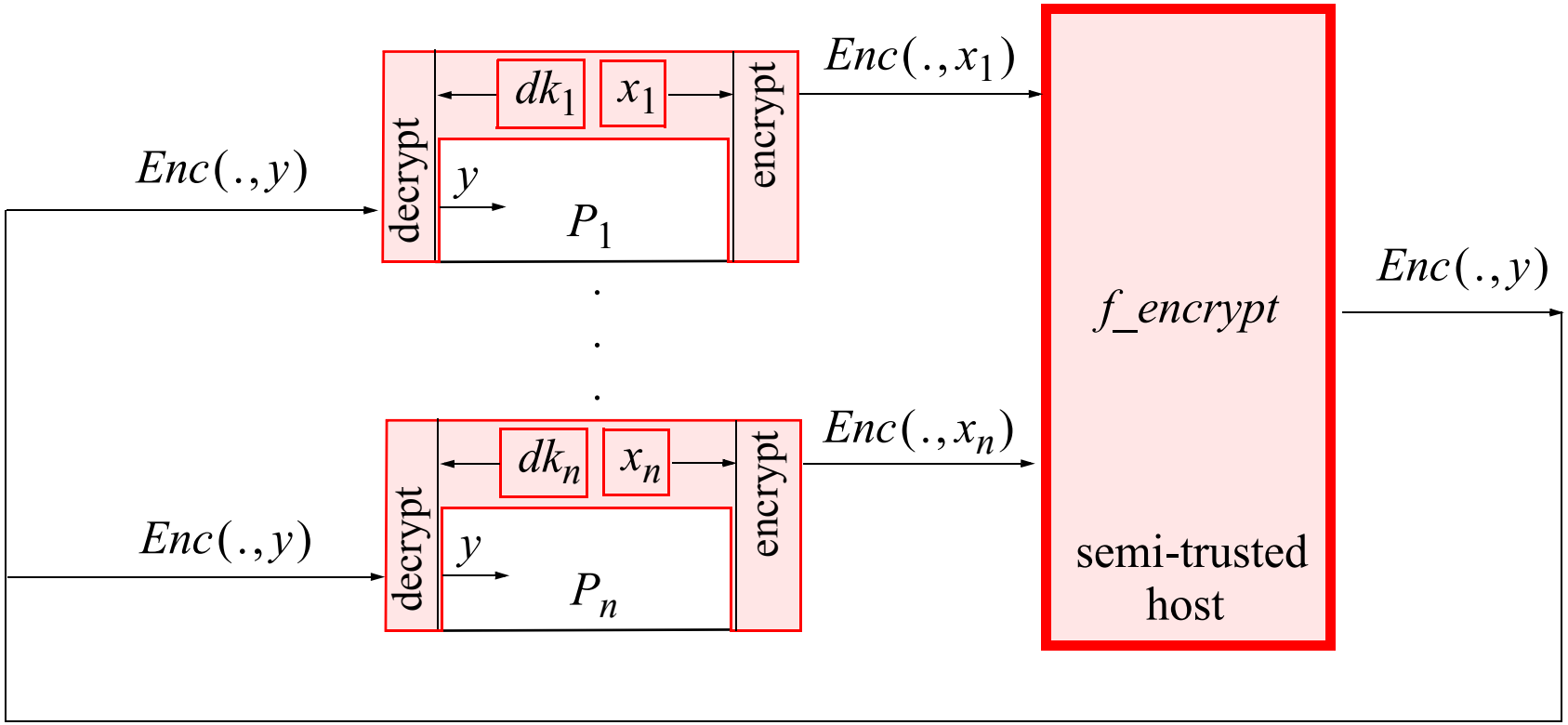
- ***correctness property*** (with threshold t)**:**
  for all inputs $x_1, \ldots, x_n$ of the parties $P_1, \ldots, P_n$, respectively, with $n > 2$,
  if the adversary is formed by at most t attacking parties (a strict minority),
  then each of the honest parties obtains $f(x_1, \ldots, x_n)$ as the final result

- ***secrecy property*** (with threshold t)**:**
  for all inputs $x_1, \ldots, x_n$ of the parties $P_1, \ldots, P_n$, respectively, with $n > 2$,
  an adversary formed by at most t attacking parties (a strict minority)
  cannot "determine" any of the secret inputs of the honest parties

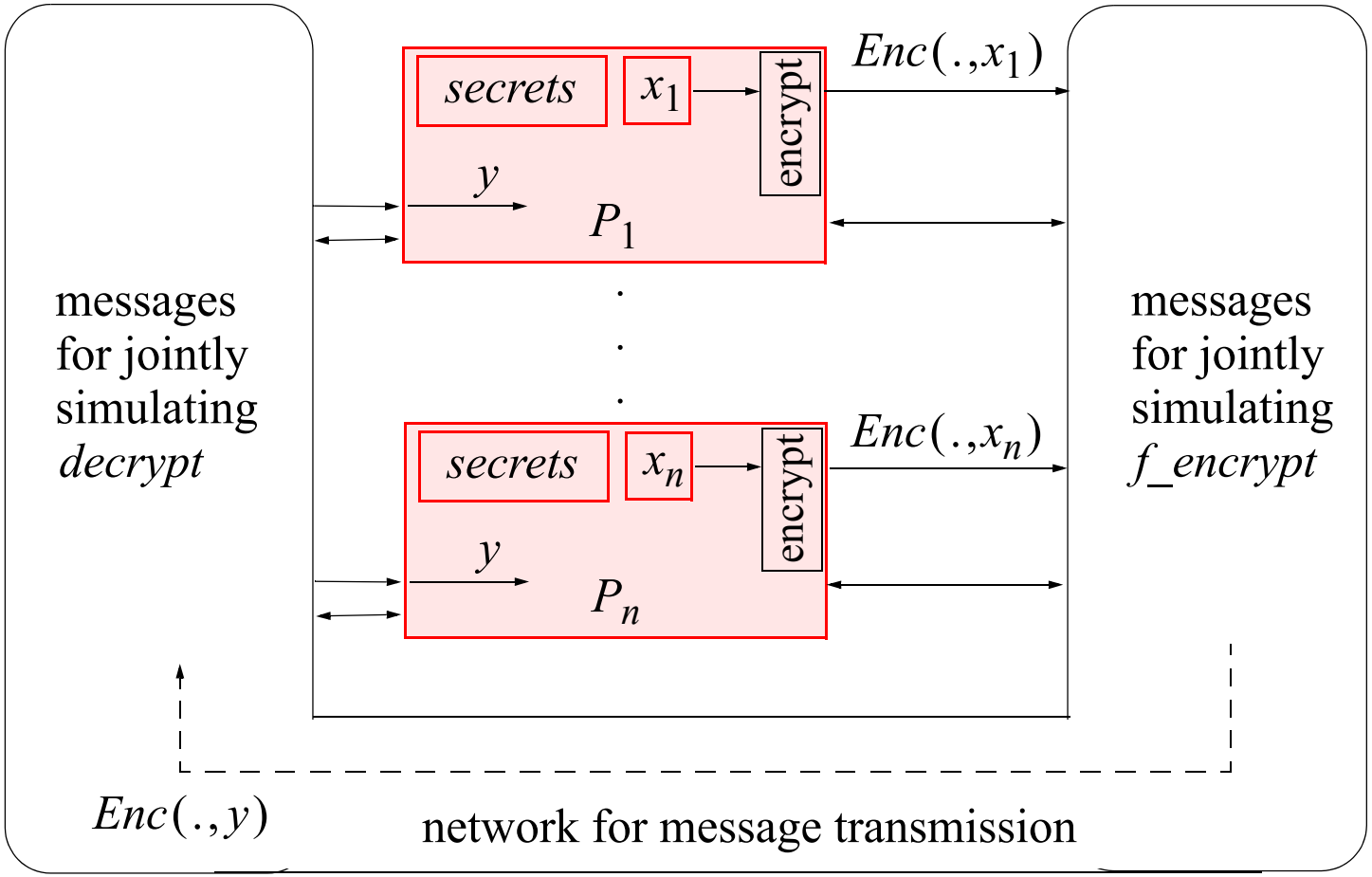
A trusted host with private input channels



A semi-trusted host operating on ciphertexts



Parties with protected local operations and message transmissions



A combined correctness and secrecy property (with threshold t)

whatever violations of correctness and secrecy
can be achieved in the model of
parties cooperating by protected local operations and message transmissions

can also (inevitably) happen in the trusted-host model,
and thus, in particular,
without observing messages of the honest parties at all

Index (erstellt von Katharina Diekmann)

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W