This discussion is based on the paper:

Basic Concepts and Taxonomy of Dependable and Secure Computing,

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr,

IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 1, NO. 1, JANUARY-MARCH 2004
Basics

- We have discussed the basic issues in the second part of Sequence 8

- Now we will focus on more survivability related issues of the aforementioned paper

- Most of the material is directly taken from the paper and (to avoid visual clutter) will not be explicitly cited!
Threats to Dependability and Security

System Life Cycle: Phases and Environment

- Development phase

- Development Environment of system consists of
  - physical world with its natural phenomena
  - human developers (+lacking competence, malicious objective)
  - development tools: software and hardware
  - production and test facilities
3) Threats to Dependability and Security

3.1: System Life Cycle: Phases and Environment

Development phase

Development Environment of system consists of

- physical world with its natural phenomena
- human developers (+lacking competence, malicious objective)
- development tools: software and hardware
- production and test facilities
3) Threats to Dependability and Security

Use phase

- System is accepted for use and starts delivering services
  1. Service delivery
  2. Service outage
  3. Service shutdown

- Maintenance may take place during all three periods of use phase
Use Environment elements:

- **Physical world**: with its natural phenomena

- **Administrators** (includes maintainers): have authority to manage, modify, repair and use system. Some authorized humans may lack competence or have malicious objectives
Use Environment elements:

- **Users**: humans or other system that receive services

- **Providers**: humans or other systems that deliver services

- **Infrastructure**: entities that provide services to the system, e.g., information sources (time, GPS) communications equipment/links, power, cooling etc.
Use Environment elements:

- **Intruders**: malicious entities (human or other systems)
  - attempt to exceed authority they have
  - alter services
  - halt them
  - alter system’s functionality or performance
  - access confidential information
  - examples: hackers, vandals, corrupt insiders, governments, malicious software
Maintenance

Fig. 3. The various forms of maintenance.
Faults: Overview

Fig: 4

elementary fault classes

Faults

<table>
<thead>
<tr>
<th>Phase of creation or occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development faults</td>
</tr>
<tr>
<td>[occur during (a) system development, (b) maintenance during the use phase, and (c) generation of procedures to operate or to maintain the system]</td>
</tr>
</tbody>
</table>

| Operational faults |
| [occur during service delivery of the use phase] |

<table>
<thead>
<tr>
<th>System boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal faults</td>
</tr>
<tr>
<td>[originate inside the system boundary]</td>
</tr>
</tbody>
</table>

| External faults |
| [originate outside the system boundary and propagate errors into the system by interaction or interference] |

<table>
<thead>
<tr>
<th>Phenomenological cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural faults</td>
</tr>
<tr>
<td>[caused by natural phenomena without human participation]</td>
</tr>
</tbody>
</table>

| Human-Made faults |
| [result from human actions] |

<table>
<thead>
<tr>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware faults</td>
</tr>
<tr>
<td>[originate in, or affect, hardware]</td>
</tr>
</tbody>
</table>

| Software faults |
| [affect software, i.e., programs or data] |

<table>
<thead>
<tr>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious faults</td>
</tr>
<tr>
<td>[introduced by a human with the malicious objective of causing harm to the system]</td>
</tr>
</tbody>
</table>

| Non-Malicious faults |
| [introduced without a malicious objective] |

<table>
<thead>
<tr>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliberate faults</td>
</tr>
<tr>
<td>[result of a harmful decision]</td>
</tr>
</tbody>
</table>

| Non-Deliberate faults |
| [introduced without awareness] |

<table>
<thead>
<tr>
<th>Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accidental faults</td>
</tr>
<tr>
<td>[introduced inadvertently]</td>
</tr>
</tbody>
</table>

| Incompetence faults |
| [result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization] |

| Permanent faults |
| [presence is assumed to be continuous in time] |

| Transient faults |
| [presence is bounded in time] |
3.2.3 On Human-Made Faults

- Non-malicious faults
  - introduced without malicious objectives
  - non-deliberate fault: due to mistakes, i.e., unintended action, developer/operator/maintainer is not aware
  - deliberate fault: due to bad decisions, i.e., unintended action that are wrong and cause faults
3.2.3 On Human-Made Faults

- Non-malicious faults
  - further partitioning into:
  - accidental faults
  - incompetence faults
Fig. 6. Classification of human-made faults.
Non-malicious faults

- Incompetence faults
  - individual, group, organization
  - e.g., Advance Automation System to replace aging USA air traffic control system
Non-malicious faults

- **Deployment faults**
  - **hardware**
    - e.g., HW “errata” are listed in specification updates
    - may continue during lifetime of the product
  - **software**
    - software aging: progressively accrued error conditions cause performance degradation of failure
    - e.g., memory bloating/leaking, unterminated threads, storage space fragmentation, accumulation of round-off errors, ...
3.2.4 On Malicious Faults

- Malicious human-made faults
  - typical goals:
    - disrupt or halt service => denial of service
    - access confidential information
    - improperly modify the systems
3.2.4 On Malicious Faults

- **Malicious logic faults**
  - development faults: e.g., Trojan horses, logic or timing bombs, trapdoors
  - operational faults: e.g. viruses, works, zombies

- **Intrusion attempts**
  - operational external faults. May be performed by system operators/admins
  - may use physical means to cause faults, e.g., power fluctuation, radiation, wire-tapping, heating/cooling
logic bomb: *malicious logic* that remains dormant in the host system till a certain time or an event occurs, or certain conditions are met, and then deletes files, slows down or crashes the host system, etc.

**Trojan horse**: *malicious logic* performing, or able to perform, an illegitimate action while giving the impression of being legitimate; the illegitimate action can be the disclosure or modification of information (attack against confidentiality or integrity) or a *logic bomb*;

**trapdoor**: *malicious logic* that provides a means of circumventing access control mechanisms;

**virus**: *malicious logic* that replicates itself and joins another program when it is executed, thereby turning into a **Trojan horse**; a virus can carry a *logic bomb*;

**worm**: *malicious logic* that replicates itself and propagates without the users being aware of it; a worm can also carry a *logic bomb*;

**zombie**: *malicious logic* that can be triggered by an attacker in order to mount a coordinated attack.
3.2.5 On Interaction Faults

- Occur in use phase
  - elements of the use environment interaction with the system
  - all *external*
  - human-made

- Examples
  - configuration faults, reconfiguration faults
3.3 Failures

- **Service failure**
  - def.: event that occurs when the delivered service deviates from correct service

- *service failure modes*: different ways in which deviation is manifested

- *content failure*: content of info delivered deviates from implementing the system function

- *timing failure*: time of arrival (early or late) or duration of info delivered at service interface deviates from implementing the system function.
3.3 Failures

- Service failure cont.

- both information and timing are incorrect:
  - *halt failure*: external state becomes constant
  - *silent failure*: no service is delivered at interface
  - *erratic failure*: service is delivered (not halted) but is erratic, e.g. babbling
3.3 Failures

Fig. 8. Service failure modes with respect to the failure domain viewpoint.
3.3 Failures

Consistency

- consistent failures: incorrect service is perceived identically by all system users

- inconsistent failures: some of all users perceive differently incorrect service. Byzantine failures
Service failure modes

Fig. 9. Service failure modes.
3.3.2 Development Failures

- **Budget failure**
  - “broke” before system passes acceptance testing

- **Schedule failure**
  - schedule slips to a point in the future where the system would be technologically obsolete or functionally inadequate for user’s needs
3.5 Faults, Errors and Failures

Fault dormancy may vary considerably, depending upon the fault, the given system's utilization, etc. The ability to identify the activation pattern of a fault that had caused one or more errors is the fault activation reproducibility. Faults can be categorized according to their activation reproducibility: Faults whose activation is reproducible are called solid, or hard, faults, whereas faults whose activation is not systematically reproducible are elusive, or soft, faults. Most residual development faults in large and complex software are elusive faults: They are intricate enough that their activation conditions depend on complex combinations of internal state and external factors.
examples

traditional hardware fault tolerance view

- physical fault (may be dormant), e.g., stuck-at
- produces error
- may result in failure
examples

- programming “bug”
  - error by programmer leads to failure to write the correct instruction or data
  - this results in a (dormant) fault in code or data
  - upon activation the fault becomes active and produces an error
  - this error may result in failure
examples

- Specification related
  - *error* by a specifier leads to *failure* to describe a function
  - this results in a *fault* in a written specification, e.g., incomplete description of a function.
  - this incomplete function may deliver service different from expected service
  - user perceives this as *error* resulting in *failure*
examples

- Inappropriate human-system interaction
  - inappropriate human-system interaction performed by operator during operation of system
  - results in external fault (from system’s viewpoint)
  - resulting altered processed data is an error...
examples

Reasoning

- *error* in reasoning leads to a maintenance or operating manual writer’s *failure* to write correct directives

- results in a *fault* in the manual (faulty directives) that will remain *dormant* as long as the directives are not acted upon...
examples

- Combined action of several faults
  - consider trap-door (by-pass access control)
  - this is a development fault
  - remains dormant until exploited
  - intruder login is deliberate interaction fault
  - intruder may create an error -> service affected
    -> failure
Hard and Soft Faults

- Hard (or solid) faults
  - fault activation is reproducible

- Soft (or elusive) faults
  - not systematically reproducible

Fig. 13. Solid versus intermittent faults.
4. Dependability and Security

From definition point of view

Fig. 14. Relationship between dependability and security.
4. Dependence and Trust

**Dependence**

- The dependence of system A on system B represents the extent to which System A’s dependability is (or would be) affected by that of System B.

- A component $a$ depends upon a component $b$ if the correctness of $b$’s service delivery is necessary for the correctness of $a$’s service delivery.

**Trust**

- Trust is accepted dependence.
4. Dependence and Trust

- Levels of dependence
  - from total dependence to complete independence

- Accepted dependence
  - judgement that level of dependence is acceptable
  - judgement possibly explicit, e.g., contract between “parties”
  - judgement may be unwilling, e.g., there is no other option!
  - the extent to which A fails to provide means of tolerating B’s failures is a measure of A’s (perhaps unthinking or unwilling) trust in B.
4.3 Attributes of Dep. & Sec.

- Availability, integrity, maintainability, reliability, safety, confidentiality...
- Don’t think binary, absolute, or deterministic
- Do think relative and probabilistic
Fig. 15. Dependability, high confidence, survivability, and trustworthiness.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Dependability</th>
<th>High Confidence</th>
<th>Survivability</th>
<th>Trustworthiness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>1) ability to deliver service that can justifiably be trusted</td>
<td>consequences of the system behavior are well understood and predictable</td>
<td>capability of a system to fulfill its mission in a timely manner</td>
<td>assurance that a system will perform as expected</td>
</tr>
<tr>
<td>Threats present</td>
<td>1) development faults (e.g., software flaws, hardware errata, malicious logic)</td>
<td>• internal and external threats</td>
<td>1) attacks (e.g., intrusions, probes, denials of service)</td>
<td>1) hostile attacks (from hackers or insiders)</td>
</tr>
<tr>
<td></td>
<td>2) physical faults (e.g., production defects, physical deterioration)</td>
<td>• naturally occurring hazards and malicious attacks from a sophisticated and well-funded adversary</td>
<td>2) failures (internally generated events due to, e.g., software design errors, hardware degradation, human errors, corrupted data)</td>
<td>2) environmental disruptions (accidental disruptions, either man-made or natural)</td>
</tr>
<tr>
<td></td>
<td>3) interaction faults (e.g., physical interference, input mistakes, attacks, input mistakes, attacks, including viruses, worms, intrusions)</td>
<td></td>
<td>3) accidents (externally generated events such as natural disasters)</td>
<td>3) human and operator errors (e.g., software flaws, mistakes by human operators)</td>
</tr>
</tbody>
</table>
5.1 Fault Prevention

- General engineering
  - e.g., prevention of development faults
- development methodologies
  - SW: e.g., information hiding, modularization strongly-typed programming languages
  - HW: e.g., design rules
5.1 Fault Tolerance

- Concepts
  - Diagnosis
  - Rollback recovery
  - Forward recovery
  - Fault masking
- How are these concepts related?
one of the important benefits of the self-checking component approach is the ability to give a clear definition of error confinement areas [63].

It is evident that not all fault tolerance techniques are equally effective. The measure of effectiveness of any given fault tolerance technique is called its coverage. The imperfections of fault tolerance, i.e., the lack of fault tolerance coverage, constitute a severe limitation to the increase in dependability that can be obtained. Such imperfections of fault tolerance (Fig. 18) are due either to development faults affecting the fault tolerance mechanisms with respect to the fault assumptions stated during the development, the consequence of which is a lack of error and fault handling coverage (defined with respect to a class of errors or faults, e.g., single errors, stuck-at faults, etc., as the conditional probability that the technique is effective, given that the errors or faults have occurred), or to fault assumptions that differ from the faults really occurring in operation, resulting in a lack of fault assumption coverage, that can be in turn due to either 1) failed component(s) not behaving as assumed, that is a lack of failure mode coverage, or 2) the occurrence of common-mode failures when independent ones are assumed, that is a lack of failure independence coverage.

The lack of error and fault handling coverage has been shown to be a drastic limit to dependability improvement [8], [1]. Similar effects can result from the lack of failure mode coverage: conservative fault assumptions (e.g., Byzantine faults) will result in a higher failure mode coverage, at the expense of necessitating an increase in the redundancy and more complex fault tolerance mechanisms, which can lead to an overall decrease in system dependability and security [54].

An important issue in coordination of the activities of multiple components is prevention of error propagation from affecting the operation of nonfailed components. This issue becomes particularly important when a given component needs to communicate some information to other components. Typical examples of such single-source information are local sensor data, the value of a local clock, the local view of the status of other components, etc. The consequence of this need to communicate single-source information from one component to other components is that nonfailed components must reach an agreement as to how the information they obtain should be employed in a mutually consistent way. This is known as the consensus problem [43].

Fault tolerance is (also) a recursive concept: it is essential that the mechanisms that implement fault tolerance should be protected against the faults that might affect them. Examples of such protection are voter replication, self-checking checkers, “stable” memory for recovery programs and data.

Systematic introduction of fault tolerance is often facilitated by the addition of support systems specialized for fault tolerance (e.g., software monitors, service processors, dedicated communication links).

Reflection, a technique for transparently and appropriately augmenting all relevant actions of an object or software component, e.g., in order to ensure that these actions can be undone if necessary, can be used in object-oriented software and through the provision of middleware [17].

Fault tolerance applies to all classes of faults. Protection against intrusions traditionally involves cryptography and...
Fault Coverage

Fig 18 fault tolerance coverage

Fault Tolerance Coverage

- Error and Fault Handling Coverage
- Fault Assumption Coverage
  - Failure Mode Coverage
  - Failure Independence Coverage
5.3 Fault Removal

During Development

Verification

the process of checking whether the system adheres to given properties, termed the verification conditions

Diagnosis

diagnosing the fault(s) that prevented the verification conditions from being fulfilled

Correction

after correction repeat verification: nonregression verification
5.3 Fault Removal

Static Verification

- Verification without actual execution
- On System:
  - use static analysis
  - theorem proving
- On Model of system behavior
  - model checking: state transition model
  - e.g., Petri net, state automata
Side Note

What is the relationship between Specification and what has been implemented?

(discussion about mapping in two directions)
Verification approaches

Fig. 19. Verification approaches.
5.4 Fault Forecasting

- Predictive approach

  - **qualitative evaluation**, aims to identify, classify, and rank the failure modes, or the event combinations (component failures or environmental conditions) that would lead to system failures;

  - **quantitative (or probabilistic) evaluation**, aims to evaluate in terms of probabilities the extent to which some of the attributes are satisfied; those attributes are then viewed as measures.
The authors are pleased to acknowledge many fruitful interactions with numerous colleagues, in particular Jean Arlat, Alain Costes, Yves Deswarte, Cliff Jones, and especially with fellow members of IFIP WG 10.4 on Dependable Computing and Fault Tolerance. Early part of this work received support from the CNRS-NSF grant "Tolerance to intentional faults."

ACKNOWLEDGMENTS

REFERENCES


AVI/ZIENIS ET AL.: BASIC CONCEPTS AND TAXONOMY OF DEPENDABLE AND SECURE COMPUTING