Fraunhofer Institut
Experimentelles
Software Engineering

# Systematic Elicitation of Security Requirements in the Context of AmI Systems


**Author:**

Reinhard Schwarz

BelAmI Report No. 003.05/E
IESE-Report No. 098.05/E

Date: November 2005
Version: 1.0
Distribution: Public

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach (Executive Director)
Prof. Dr. Peter Liggesmeyer (Director)
Fraunhofer-Platz 1
67663 Kaiserslautern

# Abstract

Security is an emergent property of a system that cannot be added to the system design independently of functional features. Security requirements have to be addressed from the beginning of the system development. Unfortunately, software security engineering is still a relatively immature discipline even for conventional software systems. With the advent of systems designed according to the new paradigm of ambient intelligence (AmI), new security problems arise,

In this report, we present useful approaches for a systematic elicitation of security requirements as a first step towards the engineering paradigm «secure by design». We survey existing methods and discuss their applicability in the context of the engineering of ambient intelligence systems. We address shortcomings of traditional approaches and suggest extensions to better cover the specific security needs of AmI systems.

**Keywords** security, system security engineering, requirements analysis, quality assurance, Common Criteria, ISO/IEC 15408, ambient intelligence, pervasive computing, ubiquitous computing, BelAmI, survey

# Preface

In software engineering the aspect of security can be approached in two different ways:

- It can be tackled directly — that is at the *product level* — with guidelines addressing architectural and functional properties of the system under construction, and by providing tools and techniques that implement concrete security features.

- Alternatively, it can be tackled indirectly — that is at the *process level* — with guidelines addressing properties of the engineering process, and by defining suitable process areas, best practices and capabilities that form a controlled, repeatable high-assurance process.

The latter approach tells the software engineer «what to do» and in what order, while the former details «how to do it». Both has its merits. On the one hand we can hardly expect to achieve sufficient security assurance if we lack control over the production process. On the other, a mature process is only a necessary but not a sufficient condition for a secure design and implementation of the system; what finally counts are the detailed properties of the engineering product.

In this report, we concentrate on direct methods to achieve system security. More specifically, we survey available guidance for the early life-cycle phases of the system: requirements analysis and early design stages. These early phases are decisive because security cannot be added as an afterthought, but has to be designed into the system from the beginning as an intrinsic property. Although not explicitly mentioned we assume, however, that the designers will follow a mature engineering process according to secure software engineering best practices (see, for example [1, 2, 3] for more information about process guidance).

# Table of Contents

# 1 Security, Safety, and Functional Security Requirements

To design a secure system, we must first anticipate the security needs that arise from its deployment and use. Understanding the application context and intended functionality of the system is the key to provide adequate security.

While there is a strong interrelation between functional requirements and security requirements, there are also significant differences. Therefore, the standard methods to capture functional requirements are less effective when it comes to eliciting security requirements. In this chapter, we will discuss how security requirements analysis relates to functional requirements analysis.

## 1.1 Security vs. Safety

The notions of security and safety are often mixed up.[1] To further contribute to this confusion, a number of related terms, such as *reliability* or *dependability*, are used in similar context — sometimes essentially as synonyms, sometimes to express subtle differences. Without diving into philosophical details, we can broadly define safety and security for the scope of our exposition as:

- **Safety** — The ability of a system to protect its users and environment from negative effects caused by system malfunction or failure.

- **Security** — The ability of a system to protect itself and its assets against abuse or damage by hostile users or environments.

Thus, safety considerations take an inside-out perspective («How could the system harm its environment?»), while security looks at the system from an outside-in perspective («How could the environment harm the system?»). As we will explain below, this is a crucial difference, although both the safety and the security perspective have several common concerns, for example:

- Integrity:
  Destroying the structure of the system or the content of its information

1 Differentiating *safety* from *security* is particularly difficult for German speakers, as both terms translate to «Sicherheit».

assets is likely to both damage the system's functionality (security), and to cause inappropriate system interaction with the environment (safety).

- Availability:
  Preventing the system from providing service means damaging the system functionality (security) on the one hand, and preventing adequate, potentially crucial support for the environment (safety) on the other.

As security has its roots in information technology, there are some unique security concerns that have no counterpart in the safety domain, which originated in the physical world of engineering, chemistry, physics, and pharmacy. A typical example is «confidentiality», a security aspect that is only relevant for information assets (i.e., software), but has no counterpart in the realm of pure hardware.

However, the most crucial difference between safety and security is that safety is concerned with threats arising from *incidental* malfunction or failure of the system. In contrast to that, security is concerned with threats arising from incidental mishaps as well as *intentional*, maliciously planned attacks on the system. As a consequence, safety analysis may resort to statistical arguments to show that the residual probability of a threat is negligible, while in security analysis, there is (typically) no meaningful way to assign a probability to the occurrence of a deliberate security incident: If it is feasible at all, a malicious adversary will create that incident (if it is worth the effort)!

The non-statistical nature of security implies that safety and security engineering have to be approached differently. In particular, safety engineering has an obvious quantitative foundation, based on statistical measures such as failure probability distribution, or meantime between failures.

For security engineering, we still lack comparable quantitative measures, and there is little reason to believe that this situation will change within the nearer future.[2] Therefore, security analysis is largely based on qualitative reasoning and indirect quality indicators such as process maturity that lack a strict causal relation to the actual security level of the system being engineered.

For the remainder of this exposition, we restrict our discussion to the security perspective of requirements elicitation, only cursory touching the safety aspects where it is deemed appropriate.

2 In [4], Smith and Spafford list «quantitative information systems risk management» among the four Grand Challenges in information security.

## 1.2 Functional security

Eliciting the functional requirements of a system design, that is the set of features the system should provide to its users, is a relatively mature art. Unfortunately, security — like safety, portability, or maintainability — is not a feature or a set of features: It is an emergent property of the system as a whole! For these types of properties, the term «nonfunctional properties» has been coined.

In fact, security requirements are not as «nonfunctional» as they appear at first. Security requirements typically challenge the design of a new or modified system component to mitigate the threat of misuse: Either, we may modify existing functionality and change all related use cases accordingly to remove a vulnerability of an existing feature. Or, we may add additional safeguards — such as encryption, logging, or access mediation and control — to provide a new layer of defense. In both cases, the so-called «nonfunctional requirement» eventually leads to a functional property of the system design. Thus, with respect to security the distinction between «functional» and «nonfunctional» is a bit unfortunate: In the final system, most security requirements are actually satisfied by adding suitable security functionality.

### Functional requirements — Use cases

Functional requirements define a concrete function, a tangible service that the system must provide to its users. Conceptually, it should be possible to assign a dedicated logical component to each such function or service in the early design stages. The nice thing about functional properties is that a system is supposed to provide only a finite, reasonably small set of functions. Thus, we can expect that after requirements analysis, we end up with a comparatively short list of functional requirements, and that these can be mapped to a correspondingly small number of system components in our design.[3]

There are well-established means for the elicitation and documentation of functional requirements. A very popular approach is to collect *use cases* [5] that illustrate required user/system interaction, see Table 1 for an example of a use-case template. From a purely functional viewpoint, an almost «mechanical» translation of use cases to their corresponding system components and functions — actors, objects being acted upon, and operations performed by actors on objects — seems feasible, provided, of course, that we have collected all significant use cases of a system.

3 Note, however, that there is no one-to-one mapping of required functions to use cases, because we have to take into account subtle interactions between functions. These feature interactions require clarification by use cases that describe the intended behavior in such specific cases. Unfortunately, it is difficult to identify each significant feature interaction, and this is an important source of system insecurity.

**Table 1** Template for the specification of use cases according to Kulak and Guiney [6]

| **Template for Use Cases** | |
|---|---|
| **Name** | A simple, intuitive name that uniquely identifies the use case |
| **Iteration** | One of «facade», «filled», «focused», «finished» — denoting how refined the description is |
| **Summary** | One or two sentences describing the interaction |
| **Basic course of events** | The steps that the actors and the system go through to accomplish the goal of this use case; this is «the most common path taken», where no errors occur and the goal of the use case is reached |
| **Alternative paths** | Less common paths than the basic course |
| **Exception paths** | Paths taken when errors occur |
| **Extension points** | Steps in the use case from where extending use cases diverge |
| **Triggers** | The entry criteria for the use case, i.e., what initiates it |
| **Assumptions** | Conditions assumed true for normal execution of the use case. But contrary to preconditions, assumptions cannot be guaranteed by the system itself |
| **Preconditions** | Conditions that must be true before the use case can be performed |
| **Postcondi-tions** | What will be true when the use case is completed; this should cover any alternative path, but not necessarily every exception path |
| **Related business rules** | Use cases describe the system boundary in an operational manner; this can be coupled to more declarative business rules |
| **Author and Date** | |

Characterizing the functional requirements in the format of use cases provides valuable input to the analysis of security requirements. Important information drawn from use cases that will help to identify security requirements are, for example:

- Who are the *actors*, that is the active entities of the system in operation? There may be internal actors (e.g., system processes, daemons, processors) as well as actors external to the system (e.g., users, cooperating other systems).

- What are the *assets*, that is the passive entities being managed by the system? In a classical IT system, we only consider information and hardware

components as assets, whereas in AmI environments, additional asset types besides information assets (e.g., actuators) need to be considered.

- What are the *operations*, that is the functions that assets may perform to control and query the system's assets?

If our collection of significant use cases is sufficiently complete and we have not overlooked important system interactions, we can expect that the derived lists of relevant actors, assets and (externally visible) operations achieves a good coverage of the entities relevant for security analysis. Furthermore, we can derive additional security information from the assumptions, pre- and post-conditions that are attributes of most use-case templates proposed in the literature:

- What kinds of *restrictions* apply, that is what assumptions and preconditions must be met for an actor to perform a certain operation on certain assets?

However, when deriving security restrictions from use cases we must not expect a very high coverage of required security constraints because use cases are formulated with functionality in mind, not with specific security needs (i.e., things that must *not* happen!). To improve security coverage, we have to complement use cases with additional analysis techniques.

### Security requirements — Misuse cases

Nonfunctional requirements in general, and security requirements in particular, do not so much refer to tangible services that the system must provide to its environment. Rather, they mostly define how the system is *not* supposed to behave.

Use cases, by their nature, concentrate on what the system should do. They have less to offer when describing the opposite. In particular, their focus is on valid uses of the system by authorized users, but not on adversaries and hostile operations that these may trigger to undermine the basic design assumptions of the system. Therefore, we can expect that the list of actors derived from use cases lacks a complete enumeration of the possible illegitimate actors and their available operations.

Mirroring the use-case approach that has proven successful for functional requirements, it has been suggested to elicit nonfunctional requirements in a similar fashion, by describing so-called misuse cases [7, 9]: After all, undesirable behavior is still behavior, so why not describing it similarly? A misuse case is simply a use case from the point of view of an actor hostile to the system under design. Table 2 shows an exemplary template for the specification of misuse cases.

**Table 2** Template for the specification of misuse cases based on Sindre and Opdahl [8]

| **Template for Misuse Cases** | |
|---|---|
| **Name** | A simple, intuitive name that uniquely identifies the use case |
| **Iteration** | One of «facade», «filled», «focused», «finished» — denoting how refined the description is |
| **Summary** | One or two sentences describing the interaction |
| **Stakeholders and risks** | Risks could simply be described textually; with more ambition one might try to quantify misuse likelihoods and costs; here, conventional risk and hazard analysis techniques would come into play |
| **Misuse profile** | The field can be useful for stating whatever there is to state about the mis-actor; for instance, some kinds of misuse are most likely to be performed by intent whereas other may happen accidentally. Some require insiders or people with high technical skill, others do not. |
| **Basic course of events** | The steps that the actors and the system go through to accomplish the goal of this misuse case; this is «the most common path taken» |
| **Alternative paths** | Various ways the misuse can be done. |
| **Capture points** | Options for how to prevent / detect the misuse at particular steps |
| **Extension points** | Optional paths may be included in the misuse case in some situations; these will be options taken by the mis-user, for instance to hack around hindrances, whereas capture points work against the mis-user. |
| **Trigger** | The condition that initiates the misuse case, or just the predicate «True», indicating that some danger is permanently present |
| **Assumptions** | Environmental conditions that make the misuse case possible |
| **Preconditions** | Those states of the system that make the misuse case possible |
| **Worst case threat** | Describes the outcome if the misuse succeeds; if the misuse case has alternative paths, this condition will often contain a disjunction to describe slight variations in the outcome |
| **Prevention guarantee** | Describes the guaranteed outcome whatever prevention path is followed; if no prevention path is followed, one might alternatively formulate what one would want the system to achieve with respect to the attempted misuse, but without stating how |
| **Detection guarantee** | Describes the guaranteed outcome whatever detection path is followed; as above, one might also make this a wanted detection guarantee |
| **Related business rules** | This is interesting for misuse cases, just as for use cases; it is useful to see exactly what business rules are broken by each misuse case, and possibly discover situations where the business rules themselves are too weakly formulated, thus opening for misuse |
| **Author and Date** | |

In fact, eliciting security requirements with misuse cases is considered current best practice [10]. Every time a new feature or use case is created, the designer should spend some time thinking about how that feature might be unintentionally misused or intentionally abused. To identify potential misuse or abuse requires expert knowledge in the application domain as well as experience with the implementation technology that is being used. Experts challenging every use case for potential misuse scenarios can achieve very good coverage of security requirements.

However, in contrast to the limited set of functional features there is an infinite set of possible misbehavior. Thus, we cannot expect to capture all security properties by naively listing all types of undesirable system interactions. The challenge here is to identify the relevant classes of adversaries and misuse cases without getting lost in the vast numbers of specific hostile interactions.

More specifically, what we are really aiming at are the relevant root causes for all kinds of inappropriate system interaction. Therefore, the most practical approach for creating misuse cases is through a process of informed brainstorming, to cover a lot of ground quickly. As Table 2 indicates, elaborating a misuse case in all detail is quite time intensive. Therefore, the attributes shown in the misuse-case template should serve merely to suggest potentially useful characteristics of misuse: To avoid overuse of the template in the early analysis, the specification of brainstormed misuse cases should be restricted to the bare minimum of essential attributes — maybe as few an intuitive name and a single text line of description (i.e., the «facade» first iteration). The template is just a suggestion for fields that a misuse case description may contain in its finished state. Along the way, most of the fields would not be mandatory. In particular, to avoid premature design decisions the designer should not try to elaborate the «prevents» or «detects» relations too early.

After we have quickly generated a list of «facade» style misuse cases, we may lean back, asking for common root cases and general vulnerabilities underlying all list items before we continue to further elaborate each misuse case:

- Are there specific assets, operations, or actors, that reoccur over and over again in several misuse cases? These may be the origin of a common vulnerability.

- Do multiple misuse cases exploit a common use-case assumption or precondition that can be invalidated by an adversary? Then use cases should rather not rely on this assumption or precondition, or a system component should be added to enforce proper preconditions for all use cases.

- Is there a simple common method to prevent or detect different misuse cases? If so, we may simply add the method to the system's features in order to get rid of all these vulnerabilities.

By grouping misuse cases into classes with common root causes we can avoid to generate redundant cases that will not further contribute to the mitigation strategy to be chosen later in the design phase.

After a coarse brainstorming of misuse cases and their grouping into «equivalence classes» it may suffice to finish a single representative of each class to adequately reflect the underlying security requirement in the final requirements documentation, and to provide sufficient justification for security design restrictions of the system. Restricting the generation of misuse cases to the most significant representatives is a prerequisite to strike the right balance between cost and value of the analysis.

As with use cases, some misuse cases are so obvious or an appropriate mitigation strategy is so straightforward that the security requirement is best captured in a simple requirements statement such as: «All communication between client and server shall be encrypted». In this situation, the generation of (mis)use cases should be abandoned. A straight requirement is sometimes more natural than forcing everything into use-case format.

### Safety requirements as misuse cases

As noted by Alexander [9], misuse cases are a valuable tool for the elicitation of various «-ility» requirements, such as reliability, maintainability, portability, usability, and also safety. Of course, safety scenarios do not necessarily involve a human agent, and therefore — strictly speaking — it is not meaningful to describe a malicious intent of an actor in case of a safety failure of a system component. Nevertheless, it can be advantageous to look at the system from a anthropomorphic point of few. For example, Alexander suggests to analyze weather as an hostile agent «intending» to make the car skid in order to identify potential weaknesses of an electronic stability control system. This is an easily understood metaphor to emphasize the safety requirements in different weather conditions. Because human language (and probably human thought) is metaphoric, according to Alexander it is wise to express requirements in this way.

Consequently, misuse cases may serve as a common tool for the elicitation and study of both security and safety requirements. There should be no fundamental difference.

### Other sources for functional security requirements

In some domains, there are legal obligations to provide a given set of security properties, or minimum standards imposed by certain branches of industry. In some cases, these norms suggests (or even prescribe as mandatory) a specific functionality for the implementation of a security feature (e.g.: «A FIPS-141 approved crypto algorithm shall be used.»).

Typical mandatory security standards regulate, in particular, minimum cryptographic strength of algorithms, minimum auditing and accounting obligations (e.g., the ability of authorities to tap a communication link, or the ability to keep the transaction history for some interval of time), and minimum privacy protection (e.g., economy of identity-related information storage, unlinkability, secure destruction after use).

Typical domains where such regulations apply are telecommunication services, financial information processing, or e-government. In these domains, many security requirements are implied directly by applicable laws or standards.

# 2 Structural Security

In Chapter 1 we pointed out that security properties of a system have much in common with ordinary functional properties; in fact, much of the security requirements will be reflected in the final system design by corresponding security functions such as encryption, authentication, or logging.

However, security goes beyond simple security functions. It has also structural aspects that have no clear manifestation in a specific functionality, but occur as a general design paradigm: These are truly *nonfunctional* properties.

## 2.1 The need for structural security

Ideally, a system should support all legitimate use cases, and it should reject all misuse cases. If all functions of the system were implemented correctly and all safeguards were in place to discriminate legitimate uses from illegitimate uses, then there should be no need for more than functional security — in principle.

Unfortunately, it is far from trivial to foresee all possible misuse cases. Sometimes, for example, all use cases work as expected, but there is a subtle feature interaction between legitimate functionality that causes a security vulnerability if (and only if) certain operations are executed in a certain order. The more complex a system design, the more likely is it that some misuse possibilities have been overlooked.

Structural security aims at a design that is as simple as possible, lends itself to easy analysis, and has some additional security margins in reserve. That it, structural security compensates for omissions in our functional security design. In a perfect system, security could be achieved without structural security aids, but as we are unable to fully master the complexity of software systems, structural security is a *must* in real-world designs.

Structural security guidelines have the advantage that they do not depend an a specific application domain, or any particular implementation technology. They are universally applicable. The downside of generality is that structural security is harder to implement than functional security, as it provides little concrete assistance for the programmer, and refers to global properties of the whole system design.

## 2.2 Principles of structural security

Back in 1975, Saltzer and Schroeder proposed eight design principles for the protection of information in computer systems [11]. Inspired by these ideas, Viega and McGraw pointed out 10 guiding principles to achieve better security [12], see Table 3. Among these principles, the first and the last («secure the weakest link», «use your community resources») do not refer to architecture, but address the question of best choices. The other eight principles are genuine guidelines for structural security.

**Table 3** Security principles according to Viega and McGraw[12]

| | Principle | Explanation |
|---|---|---|
| 1 | Secure the weakest link | Intruders will typically choose the attack with the least resistance. It is most cost-effective to secure this attack path. Be sure to secure the weakest, not the most obvious link! |
| 2 | Practice defense in depth | The system should not rely on a single line of defense, but it should provide several layers of protection. If one layer is defeated, the next layer is in place to stop the attack. |
| 3 | Fail securely | Even in the presence of failures, the system must fall back to a secure system state that does not open new attack paths due to an inconsistent system status. |
| 4 | Follow the principle of least privilege | An entity should be granted only the minimum set of privileges that is required to perform its designated task. After the task has been finished, these privileges should be withdrawn immediately. |
| 5 | Compartmentalize | Divide the system into isolated components so that subverting one component's security will not help an adversary to subvert other components. Prevent damage from spreading. |
| 6 | Keep it simple | There is virtue in simplicity, both with respect to economy of mechanism (Saltzer's and Schroeder's first principle) and with respect to the user interface and its psychological acceptability (Saltzer's and Schroeder's eights principle). The probability that a security vulnerability has been missed increases with the complexity of the design. |
| 7 | Promote privacy | Although we should prefer an open design where security does not depend on hidden implementation details (Saltzer's and Schroeder's fourth principle of open design), it is wise to reveal as little detail about the system, its execution status, its application data, and users to the outside, if not strictly necessary. |
| 8 | Remember that hiding secrets is hard | Do not put encryption keys, passwords, PINs and other confidential information in the code or hidden fields of the output: Someone will always be able to extract it from there! |
| 9 | Be reluctant to trust | Design each component of the system so that it mistrusts the other components. |
| 10 | Use your community resources | It is better to rely on designs that have been scrutinized widely by several experts than to use a homegrown, private solution. |

Several variations of such structural security guidelines have been proposed in the literature, most of them closely related to the principles in Table 3 (see, e.g., [13, 14] for an elaborated summary of existing security principles). Among these recommendations are, for example:

- «Minimize your attack surface!»
  This guideline relates to Viega's and McGraw's «Keep it simple» or to Saltzer's and Schroeder's «Economy of mechanism». Another variant of this theme is «Don't be more general than necessary» [13] or «Use the least common mechanism».

- «Follow the principle of necessary privileges!»
  This recommendation requires that there should be no system access for anonymous users, guest accounts, or debugging «back doors» that provide system functionality without requiring any access privileges.

- «Use secure defaults!»
  This guidelines requires that the system should operate in secure mode by default, without forcing the user to activate specific security options. It should require extra effort (and a conscious choice of the user) to run the system in insecure operation mode.

In the early stages of requirements analysis, the designer may have difficulties to take structural security guidelines into account because it is not at all clear what the final system design will look like. As there are no isolated «function boxes» that could represent global structural security, it is hard to express it before we can at least outline the system's functionality and overall structure. But as functional requirements start shaping the design of the system under development, structural security guidelines should give us increasing leverage in cross-checking our security requirements for completeness and adequacy with respect to the conceived system design. Therefore, structural security is particularly useful for feedback loops from early design stages back to requirements analysis.

# 3 Stimulating Creativity in Systematic Security Requirements Capture

Apart from taking use/misuse cases and structural principles as guideline for a methodological enumeration of potentially security-critical issues, little technical support exists to enforce a proper coverage of all security-related problems in the requirements phase. The general recommendation is simply to form a team consisting of subject matter experts and security analysts, and to jointly brainstorm a list of misuse scenarios, which — after initial assessment by the team — can be elaborated further together with requirements engineering experts.

Although we cannot enforce the completeness of our security analysis, there are a number of «stimulants» that help the analyst to challenge assumptions, gain different perspectives, and identify hidden weaknesses in the early design stages. Below, we will present some of these «creativity aids» as current best practice.

## 3.1 Negative stimulants: What we should avoid

If security is «the science of how the system is not supposed to behave», then we must ask ourselves: What vulnerabilities need to be prevented? What assets are at stake? There are several techniques that put the analyst in the proper disposition for such a negative thinking.

### Vulnerability taxonomies

There have been several attempts to classify known security vulnerabilities systematically, see, for example, [15, 16, 17]. Although most of these classification schemes have their weaknesses — for example, most are ambiguous in that they fail to assign each vulnerability to a unique vulnerability class — a suitable taxonomy may provide new ideas for potential misuse or abuse of the system under design.

Most taxonomies are somewhat narrow in scope. They either consider only a specific technology such as Unix operating systems or IP networks, or they are restricted to a certain application domain, for example aerospace controls. Moreover, there are taxonomies for design vulnerabilities, coding vulnerabilities, deployment vulnerabilities, and also for security management vulnerabilities. They all fit different needs and purposes.

In order to profit from a taxonomy for the requirements analysis of a system, it is crucial to select a classification that is related to the domain and/or technology under study, that addresses vulnerabilities at the proper level of abstraction, and that targets the early life-cycle phases of system development. A WWW search should provide suitable starting points for the identification of useful vulnerability taxonomies in a given project context.

### Characteristic failure modes

A related approach is to systematically ask for typical deviations from expected behavior. This is a standard method in safety analysis, for example, when assessing the dangers of energy flows. For example, in the domain of IT security we may ask the following questions to determine potential vulnerabilities of information flows:

- What happens if the information arrives **too early**? What if it arrives **too late**?
- What if the message contains **too few** bytes? What if it contains **too many**?
- What if messages arrive in **wrong order**?
- What if we receive the message multiple times, that is **too often**? What if we do **not** receive it **at all**?
- …

The point here is that *too early / too late, too few / too many, too often / not at all, out of order, …* are typical failure modes for communication channels. Different component types have different failure modes, which are often characteristic for a whole class of components. A simple list of phrases indicating characteristic types of failure will often stimulate our imagination and lead us to the discovery of new misuse scenarios.

### Characteristic threats and attack trees

For certain types of systems, there are characteristic types of threats that cause security breaches. Challenging each system component's security in the light of these threats may stimulate new insights.

At Microsoft, for example, Howard and LeBlanc [18] proposed the acronym STRIDE as a memory aid for the most typical threats to the security of information systems: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privileges (Table 1).

**Table 1** The STRIDE classification of security threats [18]

| Threat category | Guiding question | Derived requirements to counter threat (examples) |
|---|---|---|
| **S**poofing | Is this attack target susceptible to spoofing? | authentication, secured credential storage |
| **T**ampering | Can this attack target be manipulated by an attacker? | authorization, cryptographic checksum, signing |
| **R**epudiation | Can the attacker repudiate this action? | authentication, signing, logging, trusted third party |
| **I**nformation Disclosure | Can the attacker view this item or observe this activity? | authorization, encryption |
| **D**enial of Service | Can the attacker interrupt or obstruct this process or data flow? | filtering, authentication |
| **E**levation of Privileges | Can the attacker elevate her privileges by attacking this process? | Least privilege, compartmentalization |

For each potential target of attack, the analyst should ask whether this target is vulnerable by any of the STRIDE threats. If so, an attack tree [19] can be constructed with the overall attack target as its root, and a separate branch for each applicable STRIDE category as potential goals for an attack.[4] Each goal may now be further subdivided into more elementary subgoals. These subgoals may be linked conjunctively («AND»: all subgoals have to be met to meet the goal) or disjunctively («OR»: at least one subgoal has to be met to meet the goal).

The subgoals of an attack tree should be subdivided until simple, fully understood elementary goals are reached at the lowest level. There is no need to split trivial attacks to ever more detail: This effort should be spent on more fruitful analysis.

Note that complex attack trees are typically not expanded in graphical format, but as indented lists, with each indentation indicating the next sublevel of the tree (see Schneier [19] for an elaborated example).

Unfortunately, STRIDE — as well as most other simple threat classification schemes — refers to standard information system security only: Specific threats of embedded or ambient systems (e.g., location privacy, loss of a device, malicious energy exhaustion) are not specifically addresses. The lack of adequate AmI threat classifications is probably due to the relative immaturity of this new field of computer science. Thus, before applying a classification scheme the

4 Attack trees are roughly the security equivalent of *Fault Trees,* a well-established modeling technique in safety engineering.

standard categories should be enriched by new types of threats that are characteristic of ubiquitous computing scenarios.

### Malicious stakeholders and their driving motivations

In some cases it is trivial to identify a (technical) threat and suitable countermeasures once we have recognized that an item is at risk at all. The problem is to identify the hidden attack target.

For example, there was a programmer who accumulated fractional values remaining from rounding operations in financial transactions on his private bank account. For a long time, this manipulation slipped unnoticed because nobody missed the accounted money that was lost due to rounding errors. And because nobody had ever thought that rounding errors may be targeted for a security breach, nobody took the time to inspect the software for this type of attack.

To prevent this type of unexpected security traps, it is worthwhile to do some brainstorming about malicious stakeholders and their potential attack targets and motivations:

- Why would an adversary want to attack the system?
- Is there anything worthwhile attacking (e.g., to gain access to it, to destroy it or render it useless for others)?
- What could be gained by the adversary, or lost by legitimate users?

### Attack patterns

In [20], Hoglund and McGraw describe more than 40 common software vulnerabilities that enable reoccurring attack variants on IT systems. They call these reoccurring variants attack patterns.

Strictly speaking, attack patterns in the sense of [20] address poor coding practices. But the root cause of many patterns lies in earlier design phases where fundamental security requirements have been neglected, such as: «Validate all input!», «Instead of rejecting known bad input, permit only known good input!», … .

Using attack patterns as starting points, working back to their abstract root causes and applying these to our system context, we often end up with a basic security requirement. In this sense, attack patterns may serve as a good source for inspiration.

Like most vulnerability taxonomies, most existing attack patterns originate from the realm of information systems. Therefore, they cover little material specific for the context of ambient intelligence systems.

## 3.2 Positive stimulants: What could be helpful

Over the past, a body of knowledge has been collected about security best practices for software systems. As we saw in Chapter 2, this knowledge has been condensed into Design Principles [11, 12] or Golden Rules [13] that cover a whole spectrum of engineering phases: Recommendations address requirements, design, coding, and even managerial aspects of security.

An interesting approach is to narrow the scope, and to provide more specific guidance to the analyst and the designer by specifying fixed security patterns or functional components, respectively, to address reoccurring security problems.

### Security Patterns

Design patterns have been very influential in object-oriented software engineering since the appearance of the seminal publication [21] by the so-called Gang of Four in 1995. In this book, the metaphor of a proven, successful architectural pattern for constructing buildings is transferred to the realm of software, and it is shown how fixed object-oriented software patterns may promote reuse — and also software quality.

There have been attempts to extend the metaphor one step further, and to provide comparable «security patterns» as prototypes for security solutions [22, 23]. One motivation is to provide better security with less effort — and more reliably and predictably; the other is to leverage expert knowledge to a broader audience. And by specifying security patterns in a formal notation, there is even a possibility to validate such patterns, for example with model checking [24, 25], even though this research is still in its infancy.

Table 4 shows a template for the specification of security patterns. Several variants have been proposed by different authors, but the template below is a representative example. Several collections of security patterns are publicly available for download by various organizations.

Patterns provide useful insight into reoccurring security problems and their solutions. There are several problems and open research issues with existing pattern collections, though:

- Security patterns are provided at different levels of abstractions and for different needs: There are structural patterns (comparable to design

**Table 4** A template for security patterns according to [26]

| Template for Security Patterns | |
|---|---|
| **Pattern Name** | Descriptive name that captures the essence of the pattern |
| **Abstract** | Brief summary of purpose and intent of the pattern, including some indications of any limitations applying to the pattern |
| **Problem** | Outline of the conditions that motivate the usage of the pattern, and the context where it is applicable |
| **Solution** | High-level description of how the pattern solves the problem |
| **Issues** | More detailed explanations, hints, and caveats concerning the application of the pattern; potential residual vulnerabilities despite proper pattern application |
| **Examples** | Citation of known uses of the pattern, potentially at different implementation levels |
| **Trade-offs** | Potential impact of pattern on other functional or non-functional properties, in particular with respect to<br>• Accountability<br>• Availability<br>• Confidentiality<br>• Integrity<br>• Maintainability<br>• Usability<br>• Performance<br>• Cost<br>with focus on primary effects |
| **Related Patterns** | Links to other related patterns with a brief indication of the type of relationship |
| **References** | Citations in the literature related to the pattern |

principles as discussed in Chapter 2), procedural patterns, and code patterns. Different people count different artefacts at different levels of formality as a pattern — informal textual descriptions versus UML-like specifications with well-defined formal semantics.

- Remember that security is not a feature: Some security properties do not easily fit into a fixed pattern that could be depicted as, for example, a UML component, but emerge from the system design as a whole.

- Remember that security is about correct behavior and state in every detail, not about «roughly correct at large»: Doesn't that contradict the template approach of generalized patterns? Satisfying the pattern only superficially, at macroscopic level may give us a deceptive feeling of security, while a vulnerability is still present due to a remaining weakness in the detailed system implementation.

- Many patterns are too abstract, providing little tangible guidance in project context; others are almost too detailed, limiting their portability. Consensus has not been reached yet what the suitable level of abstraction should be.

So far, research in security patterns is still rather software-centric, dealing mostly with the classical problems of information security. We still lack patterns that address the integrated systems view of software, hardware, and environment, that is typical for security in the realm of embedded systems design.[5]

### Security Functional Components

In Chapter 1.2, we pointed out that most security-related requirements are eventually met by providing some safeguarding functionality. Different software systems face similar kinds of security threats, therefore we can expect that they have most of their safeguards in common: The same type of security functionality reoccurs in different system designs. A canonical list of security functionality should be a valuable aid to check a security design for completeness.

The Common Criteria for Information Technology Security Evaluation [27] provide such a canonical collection of so-called *security functional components*. This list was originally conceived as an profile against which the security of IT products could be evaluated objectively. But besides *evaluators*, [27] explicitly counts *developers* among the principle users of this standard.

According to their authors, the Common Criteria's functional components represent the current state of the art in security requirement specification and evaluation. Of course, the Common Criteria do not presume to include all possible security functional components, but their authors tried to be as complete as possible. Note that the internationally harmonized standard [27] has been based on (and essentially supersedes) a number of renown national predecessors, among them the U.S. *Trusted Computer Security Evaluation Criteria* (TCSEC, also known as «Orange Book»), the *Canadian Trusted Computer Product Evaluation Criteria* (CTCPEC), and the European *Information Technology Security Evaluation Criteria* (ITSEC). Contributions came also from governmental organizations in Australia, New Zealand, and Japan. This warrants a closer look at the security functional components as defined by the Common Criteria.

In the next chapter, we give an overview of the Common Criteria, the underlying system model and its security functional components. We then discuss the applicability of the Common Criteria in the context of AmI systems.

5 Note that we have better models for system *safety*: In safety analysis there are statistical models to describe the probability of (hardware) component failures, and these models can be used to predict the failure probability of the overall system. But even in the safety domain, we lack adequate predictors for software failure probabilities and distributions.

# 4 Requirements Analysis Based on the Common Criteria

The Common Criteria Version 2.1 have been standardized as international standard ISO/IEC 15408:1999. Based on user feedback, the new version 3.0 has been created and is currently under revision, and we expect that ISO/IEC 15408 will eventually adapt to this new release of the Common Criteria. The discussion below is based on [27], the latest revision of the Common Criteria, published in June 2005.

## 4.1 Intent and structure of the Common Criteria

Prior to the Common Criteria (CC) several countries had their own national standards for the security evaluation of IT systems. As a consequence, evaluations done in country *A* were not accepted by the authorities in country *B*; thus, they had to be repeated for every country. To save this waste of effort, and to facilitate comparability and mutual acceptance of the results of independent security evaluations, the United States, Canada, and (parts of) the European Community harmonized their national evaluation standards and developed the new CC as their common successor. In 1999, CCv2.1 was formally accepted as ISO/IEC 15408, and new versions CCv2.2 and CCv3.0 followed in 2004 and 2005, respectively.

According to [27], the CC permit comparability between security evaluations by «*… providing a common set of requirements for the security functionality of (collections of) IT products and for assurance measures applied to these IT products during a security evaluation. The evaluation process establishes a level of confidence that the security functionality of these products and the assurance measures applied to these IT products meets these requirements. The evaluation result may help consumers to determine whether these IT products fulfill their security needs. […] The CC is applicable to IT security functionality implemented in hardware, firmware, or software.*»

CCv3 consists of the following parts:

- Part 1: Introduction and general model
- Part 2: Security functional components

- Part 3: Security assurance components

Part 2 provides guidance and reference when formulating statements of security requirements. This is the most relevant part of the CC for system analysis. In particular, it provides the canonical list of security functionality, reminding the system designer of security issues that might be relevant. Part 2 mainly ensures good *coverage* of security needs: As far as the list of functional components is complete, no security threat should slip unnoticed.

Part 3 provides guidance when determining the required levels of *assurance* for each security issue. That is, it proposes a hierarchy of assurance measures with increasing strength of evidence, and the analyst may select from that scale the appropriate assurance level with respect to each security functional component. For our discussion, the assurance dimension of security evaluations is less important, as our focus is on requirements analysis rather than system validation.

Part 1 provides background information and reference. Below, we restrict the discussion to Part 2, which is most relevant for our discussion.

## 4.2 System reference model

### Building blocks

The functional components of the CC (and our extensions presented later in the discussion) refer to a reference model according to Figure 1. The main building blocks of the model are:

- The *target of evaluation* (TOE), a system under study with well-defined structure and boundaries
- The *TOE security functionality* (TSF), those parts of the system that are relevant for enforcing the security policy of the system
- *Subjects*, the active entities of the system (e.g., processes)
- *Objects*, the passive entities of the system (e.g., files, database entries)
- *External active entities* interacting with the TOE (e.g., users or other systems— «machine users» in CC parlance)
- *Operations,* the possible interactions between subjects and objects (and associated modifications of subject or object attributes)

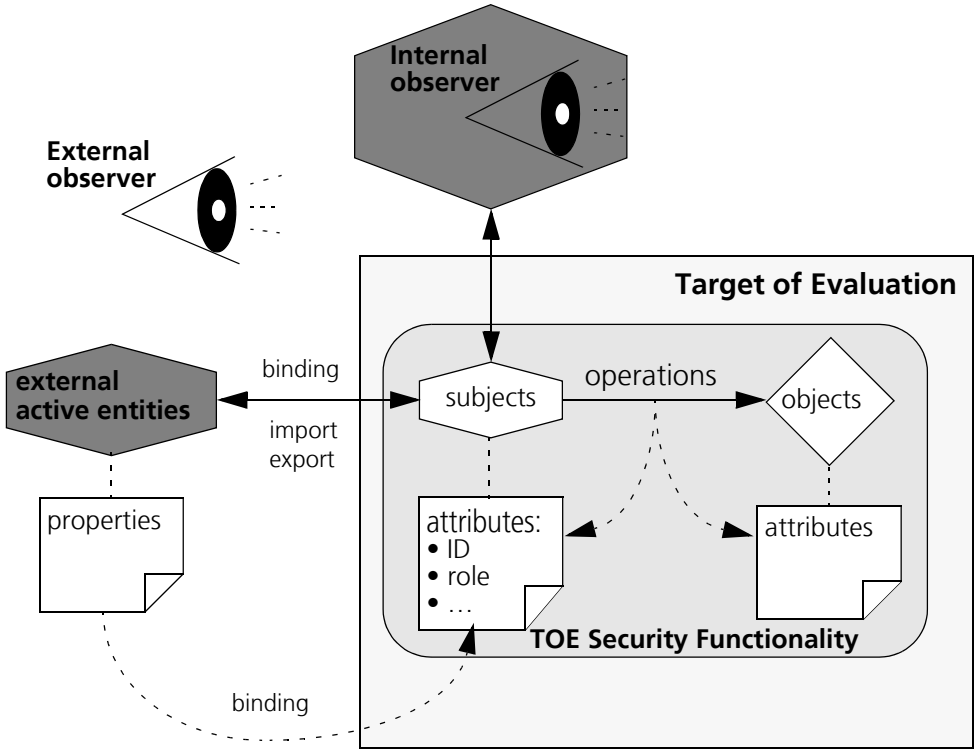


**Figure 1** Main components of an evaluation scenario according to [27]

- *Bindings*, a formal process of associating an external active entity to a TSF subject acting on its behalf, where entity properties are translated to suitable subject attributes (e.g., logging in with account name and password to a system).

Note that according to this model, objects are only accessible via subjects and their available operations. For example, a user can only create, modify, read, or delete objects if he binds to a subject that acts on behalf of the user and provides the required operations. The implementation of the system has to ensure that direct object access by external entities is prevented.

Whether a subject is allowed to operate on an object depends on access rules that are based on the security attributes of subjects and the objects. The security policy determines what kind of access rules are required.

With respect to observability, non-repudiation, and privacy, two types of observers can be distinguished:

1. *Internal observers* who are bound to a subject (e.g., as a regular user of the system) and may query the status of the system (i.e., internal state, data values) with their accessible operations to acquire information about other

objects, subjects, or the users that they represent.

2. *External observers* — not formally bound to a TOE subject — who are able to trace

   – data explicitly exported or imported by the system

   – signals emitted by the system containing implicit information (e.g., electromagnetic emanation that reveals presence, location, and maybe even identity of devices or their users).

Relative to this reference model Part 2 of the CC defines six functional classes, and within each class several functional families as detailed below. Each functional family contains one or more components, any one of which may be selected to specify the security requirements with respect to this functional family. For a given TOE, some functional components may be useful or necessary, while others may turn out to be irrelevant and may be omitted in the specification. The functional classes and their scope are:

- *Data Protection and privacy* (FDP) —
  Proper initialization of security attributes of subjects and objects, access permissions for subjects on objects with respect to the available operations, and rollback of operations

- *Identification, authentication, and binding* (FIA) —
  Binding of external entities to subjects with proper authentication, lock-out and termination of such bindings

- *Communication* (FCO) —
  Import and export of data between subjects and external entities, observability and accountability of these exports and imports

- *Audit* (FAU) —
  Level of detail of log data being recorded, automated means of log information preprocessing, and automated response to critical log entries

- *Protection of the TSF* (FPT) —
  Self-protection and resilience to failure of security functionality and its system resources, so that other functional components can rely on the availability and integrity of the security mechanisms.

- *Miscellaneous* (FMI) —
  Remaining issues that did not fit in any of the previous classes

### Requirements specifications with Common Criteria

According to the reference model of CC Part 2, requirements are formulated in terms of subjects, objects, operations, security attributes, and bindings. To this end, the CC provide template requirements statements of varying stringency for each functional component. For example, functional component «Audit data generation with time» (FAU_GEN.2) may be specified according to these template specifications:

**FAU_GEN2.1** The TSF shall store an audit record in *<object>* of the following events: [ *start-up of the audit functions* | *shut-down of the audit functions* | *<other events that will be audited>* ].

**FAU_GEN2.2** The TSF shall record within each audit record the following information:

- Date and time of the event, type of event, values of *<security attributes of the subject>*, the [ *success* | *failure* | *<other outcome>* ] of the event; and
- *<other information>*.

For a concrete system, suitable text has to be selected for the attributes printed in italics. Of course, if no CC certification is required for the system under study, then the analyst is free to use any specification format that fits the purpose best; in this case, the CC templates merely provide suggestions for requirements that may be useful, but they are not binding. For example, the analyst may reformulate CC requirements templates as (mis)use cases for the particular system under analysis.

### Applicability of the reference model to AmI systems

The reference model as depicted in Fig. 1 on page 25 was originally inspired by standard IT systems where we can safely assume a fixed system configuration, well-defined system boundaries with a clear distinction between «inside the system» and «outside the system», and an explicit binding under user control. All these assumptions are met only partially in an AmI context.

First, mobile devices may dynamically enter and exit an AmI system. Thus, the system configuration is subject to constant change, and it is hard to predict what kinds of security-critical constellations may occur during the lifetime of the system. Second, if the actual ensemble of system components during operation is unpredictable, then the system boundaries are so, too. But without proper system boundaries, the concept of internal and external entities is blurred, and so is — to some degree — the concept of «bindings». To further exacerbate the

situation, mobile devices may join or leave an AmI environment autonomously, without any conscious user interaction — almost «behind the back» without forcing the user to issue an explicit binding request. Furthermore, wireless communication adds to the problem of clearly defining (and enforcing!) the boundaries of the TSF.

As a consequence, some of the AmI-specific security aspects are only marginally covered by the predefined functional components, and the reference model has to be applied with some liberality to capture the spirit of the original standard.

For example, in the absence of a fixed configuration of system components we may not be able to determine the exact borderline between internal subjects and external active entities. Nevertheless, from the perspective of each individual component we can argue that the protection of this component must be self-contained, and therefore (a selected subset of) the CC apply relative to the component's boundaries.[6] We may also be able to define a proper superset of the maximum configuration that our system will support. In this case, the superset as a whole defines a «maximum TSF» to which the CC can be applied.

We also have to consider additional functional components that are specific to AmI scenarios and are therefore missing in the CC component catalogue. For example, the theft of system components is not specifically addressed by the CC; it is only loosely covered by the CC component «TSF physical protection» (FPT_PHP). In an AmI environment, however, losing as personalized device that contains valuable private information — and may serve as an access token to other critical parts of the system — has quite different connotations than protecting a stationary mainframe against unauthorized physical access. As another example, consider an adversary trying to maliciously exhaust a mobile component's energy resources by constantly involving it in unsolicited communication: In contrast to processing power and storage capacity, energy is not mentioned in the CC as a relevant resource that requires protection against monopolization or misuse.

We will come back to the issue of necessary extensions to the CC and their proper interpretation after we have discussed the available functional components. Independently of the shortcomings mentioned above, the CC provide a reasonable framework for the analysis of a complex IT system. At present, it is one of the most carefully elaborated collections of general security requirements that is available. Therefore, we recommend it as a starting point for the exploration and classification of security threats and countermeasures.

6 This approach is in line with the general principle that a chain can be only as strong as its weakest link.

## 4.3 Security functional components of the Common Criteria

Below, we present the existing security functional components of the current CCv3. For further reference, and for the requirements specification templates associated with each component, we refer the reader to [27].

For each component, we provide the official component name and label according to the CC. Furthermore, we briefly characterize the scope of each component, that is, the type of requirements that the component tries to regulate. In addition to that, we added to each component a list of guiding questions to illustrate the typical kinds of security problems that should be considered in the context of the component. These questions form a checklist for the analysis of security requirements.

### Class FDP — Data protection and privacy

This class defines components for the interaction between subjects and objects (operations). It also defines criteria for security attributes, how they obtain an initial value, and how they are subsequently modified. These components are used to define access permissions, access right propagation, and observability.

**Table 5** Functional class FDP according to [27]

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FDP_ACC | Access control | Rules for the performing of operations on objects by subjects, based on security attributes; automatic modification of security attributes based on this access | What are the rules that restrict the flow of information between user and system, or between different users? Which attributes of subjects or objects determine access permission? Which subjects, objects and operations should be manageable in what way? Who should be allowed to enable, disable, or modify what types of functions?<br>What security related roles are defined for the system? Are there any dependencies between roles such as exclusion relationships? Shall a user implicitly assume all roles allowed for him/her, or only one default role and assume any other roles only explicitly?<br>What audit information is available for the various authorized audit roles? Is there any supporting functionality for audit analysis (e.g., searching, sorting, or defining logical views)? Who may use these additional functions? |
| FDP_ISA | Initialization of security attributes | Rules and defaults for attribute initialization for newly created objects or subjects | Does the system enforce any minimal conditions (e.g., safe defaults, consistency restrictions) on security attributes? What are the default attributes for newly created subjects or objects? What are the security attributes maintained by the system for each individual user, if any? How are they initialized? |

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FDP_ROL | Rollback | Rules and capabilities for undoing operations | Should it be possible to recover (undelete) data or undo operations? Which ones? If so, how do we prevent unauthorized users from recovering and reading deleted information? |
| FDP_MSA | Management of security attributes | Rules for access and modification of attributes of existing subjects or objects; automatic changes in security attributes. | Who is allowed to create/read/modify security attributes (such as access rights) in what way? Which attributes? If privileges are granted or revoked from a user, when does that take effect?<br>Are there any automatic changes in security attributes? What are the triggers for automatic changes, and what are the rules that determine the type of change? In particular, should security attributes expire and what happens to them after expiring? |
| FDP_UNL | Unlinkability | Restrictions that determine whether subjects are unable to link different subjects, objects, or operations together in some way | Is it possible for internal observers to identify (some? all?) actions performed by the same user or pseudonym as being related?<br>For all data that might be associated with a user, should it be possible to trace the data back to a specific person or pseudonym? Who should be allowed to resolve pseudonyms to true identities?<br>Is the identity (or user account) only concealed relative to other users, or even to the system itself? |
| FDP_UNO | Unobservability | Whether subjects are unable to observe other subjects | Should it be possible for an internal observer to deduce that a system resource is in use, that an operation is being performed, or that a certain system entity has a certain value? How much information can one subject obtain about all unrelated subjects, objects, and operations in the system? Is there a mechanism for observability for a limited subset of authorized observers? Who may still observe resource access and system activity?<br>Can users hide their locations? Can they hide them even from (parts of) the system? Or from what other user groups? Is there a need for strict location privacy, or is it sufficient to reduce the achievable accuracy of location information?<br>Is there any data that the system must not store at all, or that must be deleted within a given period of time? Are there any other data protection («Datenschutz») requirements? What legal obligations apply to information stored in the system? |

### Class FIA — Identification, authentication and binding

This class defines components that regulate how and under which conditions users may bind to subjects, what the consequences of this binding are for the subject, and under which conditions the binding is dissolved or temporary disabled.

**Table 6** Functional class FIA according to [27]

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FIA_URE | User registration | Receiving user properties and authentication data, and storing them; returning derived properties and authentication data to the user | Does the system require formal user registration before a user can access the system? If so, what properties are required for registration, and what properties does the system assign to each registered user? Can users register themselves (online registration), or do we need offline registration via a separate administration action? |
| FIA_QAD | Quality of authentication data | Ensuring sufficient quality of authentication data | Should there be any prescriptions for the format and quality of authentication data (e.g., password length and character set, number of bits of a key)? Is a minimum quality enforced by the system (verification)? Should the system actively provide security secrets that meet the required quality standards (generation)? How are users identified (e.g., by the hard-wired origin of their binding request, by a hardware token)? |
| FIA_UID | User identification | Allowing or denying anonymous bindings | Does the system provide anonymous access without prior user identification? To which subjects within the system? Which bindings require prior user identification?<br>Should the system (or a subgroup of its users) always know about the true identity of a certain user, or should the system accept pseudonyms in place or user names (i.e. names which may be resolved to the true identity of the user, but only with the help of an external entity and by authorized personnel)? Is it possible for a user to use more than one pseudonym? Which operations offer pseudonymity? |
| FIA_UAU | User authentication | Requiring authentication on binding and re-authentication before critical operations; protecting the authentication data and process | Are there any actions that can be performed on behalf of the user prior to reliable authentication? Are there any actions requiring a re-authentication of a user? How are users authenticated and what are the constraints in the authentication process? Is there a need to formally validate the authenticity of information? For what types of objects or data types? Who should be able to generate or verify evidence that can be used to validate the authenticity of this information? How should data be verified to ensure that it is valid and authentic? Should the system limit the authentication feedback to prevent others from observing the user's authentication secrets (e.g., by echoing back only '*' characters to avoid «shoulder surfing»)? |

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FIA_AFL | Authentication failures | Reacting to failed authentications | How should failed or manipulated authentication attempts be handled? Should the system restrict user feedback to avoid assisting impostors in guessing valid credentials? |
| FIA_TBR | TSF binding rules | Restricting bindings even for authenticated users, based on other security attributes | Based on what attributes is permission or scope of the session restricted (e.g.: origin, time, number of existing bindings, security of communication link, strength of authentication method)? Is there a limit to the number of parallel user/subject bindings that is allowed? Should differently strong types user/subject bindings be possible? How should the type of binding be chosen? |
| FIA_USB | User/subject binding | Rules for transforming user properties into subject attributes so that the subject correctly represents the user | Does the subject change its attributes on user binding? Should the actions a user performs be associated with that user's properties, or do they uniformly run under equal conditions, independent of a specific user? How are the user's properties mapped to corresponding subject attributes? What are these attributes? Should the system (or a subgroup of its users) always know about the true identity of a certain user? |
| FIA_SUA | Subject/TSF authentication | Authenticating the TSF to a user to prevent impersonation of the system | Does the user require server authentication? Is their any danger of system impersonating? |
| FIA_TIN | TSF information | Rules for displaying advisory warnings or access histories to users | How many feedback does the system provide prior to and during successful authentication? Should the recent access history of the user be displayed upon successful user/subject binding? |
| FIA_LOB | Lock-out of bindings | Rules for temporary disabling of bindings | How should session locking be handled? Should there be an automatic time-out? In what state should session locking leave the system and its security attributes? |
| FIA_TOB | Termination of bindings | Rules for enforced termination of bindings | How should session termination be handled? Should there be an automatic session time-out? Should the system restrict sessions to official «working hours» or working days, shutting down active sessions at the end of the «office hours period»? In what state should session termination leave the system and its security attributes? |

### Class FCO — Communication

The components of this class address the protection of the communication between subjects and users bound to those subjects.

**Table 7** Functional class FCO according to [27]

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FCO_ETC | Export to outside TSF control | Rules for limiting the transfer of accessible data to users | How is secure data to be exported out of the system? Who may receive such data, under what circumstances? Are security attributes firmly associated with the data (attached), or are they supplied separately from the data (detached)? What happens to these attributes on export? |
| FCO_TED | Translation of exported data | Rules for the transformation of exported data and attributes relevant to security | Do we need to translate data or security attributes before export to (machine) users to allow consistent interpretation? Which data or attributes require translation? What are the rules for translation? |
| FCO_AED | Availability of exported data | Availability requirements for exportable data | Is there any data that must be highly available to users of the system? What is the metric to express the availability constraints, and (with respect to this metric) what degree of availability is required for each user group? |
| FCO_CED | Confidentiality of exported data | Confidentiality requirements for exportable data | Consider what kind of system data (e.g. passwords) is allowed to be exported to a remote trusted system, and how it may be transferred. Is there a need to protect the data transfer against unauthorized disclosure? |
| FCO_IED | Integrity of exported data | Integrity requirements for exportable data | Consider what kind of system data (e.g. passwords) is allowed to be exported to a remote trusted system, and how it may be transferred. Is there a need to protect the data transfer against modification? |
| FCO_NRE | Non-repudiation of exported data | Providing evidence for the origin to of data to the user; requesting evidence from the user for the receipt of data | Should it be provable that a user has received a certain message such as an access code for a prepaid service? Should it be provable that the system has sent certain information? What attributes should be stored with the respective evidence (e.g. the time of receipt)? |
| FCO_UNE | Unobservability of export | Hiding the destination of export, or hiding that data is being exported at all | Is there a need to hide the fact that data is being exported, or to hide origin (system identity) or destination (user identity) of the exported data? |
| FCO_ITC | Import from outside TSF control | Rules for limiting a subject from importing data from users | Under what circumstances is insecure data from outside the secure system allowed to be imported into the system? Are security attributes firmly associated with the data (attached), or are they supplied separately from the data (detached)? What happens to these attributes on import? |

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FCO_TID | Translation of imported data | Rules for the transformation of imported data and attributes relevant to security | Do we need to translate data or security attributes on import from (machine) users to allow consistent interpretation? Which data or attributes require translation? What are the rules for translation? |
| FCO_CID | Confidentiality of imported data | Confidentiality requirements for imported data | Consider what kind of system data (e.g. passwords) is allowed to be imported into the system, and how it may be transferred. Is there a need to protect the data transfer against unauthorized disclosure? |
| FCO_IID | Integrity of imported data | Integrity requirements for imported data | Consider what kind of system data (e.g. passwords) is allowed to be imported into the system, and how it may be transferred. Is there a need to protect the data transfer against unauthorized modification? |
| FCO_NRI | Non-repudiation of imported data | Proving evidence to the user for the receipt of data; requesting evidence from the user for the origin of the data | Should it be provable that the system has received a certain message such as an electronic buying order? Should it be provable that the user has sent certain information? What attributes should be stored with the record of action and the respective evidence (e.g. the time of receipt)? |

## Class FAU — Security audit

This class defines components that describe logging of events, automatically analyzing them and acting on the results of this analysis. We use such components to specify, for example, requirements for accountability and traceability.

**Table 8** Functional class FAU according to [27]

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FAU_GEN | Security audit data generation | Types of events that are recorded, attributes that are recorded per event | What is the minimum amount of information to be included in the audit? For each auditable event, what data should its audit record contain? Is there a trusted time source so that each event can be reliably timestamped? How are events selected for auditing (e.g., based on user, subject, object, operation, origin, time of day, type of communication link)? |
| FAU_SAA | Security audit analysis | Types of automated monitoring applied to audit log, thresholds, profiles, heuristics and patterns used as reference for analysis | Is there a mechanism to automatically analyze the audit logs (e.g., based on basic thresholds, anomaly detection, attack signatures)? What are the underlying threshold values, profiles, or signatures? Is this monitoring performed in real-time, or periodically? |

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FAU_ARP | Security audit automatic response | Rules for triggering actions in case of a potential security violation | Does the system compute a continuous alert level (e.g., «traffic light» status: green, yellow, red), or does it generate imminent alert notifications. Does it lock or terminate sessions in response to an audit alert? What other reactions are triggered, and under what circumstances? |

## Class FPT — Protection of the target of evaluation security functionality

This class describes components that define the self-protection of the TSF. Note that all other security function classes assume that the TSF itself is physically protected, that TSF integrity is maintained, and that the TSF has an infinite amount of idealized resources at its disposal.

**Table 9** Functional class FPT according to [27]

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FPT_TOU | Testing of users | Testing environmental conditions and connected machine users | Should the system check the integrity of its execution environment? Which of the surrounding systems («machine users») need to be tested, when, and how often? What types of tests should be applied, what kind of attributes need to be tested? |
| FPT_TST | TSF self test | Rules for verifying the integrity of TSF hardware and software, including data | Should the system run self-tests to demonstrate its proper functioning and the integrity of code and data? Which components need to be tested, in what way, when and how often should the tests be performed?<br>How can the system detect data integrity violations, and how will it react to an integrity error? |
| FPT_FLT | Fault tolerance | Requirement to continue to meet security requirements in the presence of faults | Should the system be able to endure some partial failure without moving to failure mode (i.e., compensate failure and still continue to meet all security requirements as in the non-failure case, avoiding FPT_FLS conditions)? |
| FPT_FLS | Fail secure | Requirement to enter secure failure mode in the presence of unrecoverable failure; specification of security requirements that are no longer met in failure mode | Are there any situations where the system might partly or completely fail? How should the system react to such failures, and is there is a need that the system is finally able to recover from failure. What types of failures must be securely handled? |
| FPT_PHP | Physical protection | Deterrence and detection of, resistance and response to physical attack | Should there be any physical protection (hardware keys, locked rooms) of the system? Is it sufficient to detect physical attack and to notify the users, or should the system resist to certain types of tampering attempts? How should the system respond to an attack? |

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FPT_RCV | Trusted recovery | Requirement to recover from failure mode to non-failure mode for certain failure classes | Should the system or its users be able to recover from failure mode (FPT_FLS) back to non-failure mode? Should the recovery occur automatically or under manual control? For which types of failures? Should the system have special, restricted mode of operation for recovering from crashes ("maintenance mode")? Do we need complete recovery, or is partial recovery of core functionality sufficient? How much loss of data is tolerable? Can we determine the amount of data loss after recovery? |
| FPT_PRI | Priority | Rules for assigning priorities to resource allocation requests and operations issued by subjects | Should all subjects be treated alike regarding resource consumption, or should some subjects have higher priority than others? For example, should service requests from anonymous users be limited to less resource consumption than those from identified users? Should low-priority tasks be preempted in favour of high-priority tasks if there is a lack of resources? |
| FPT_RSA | Resource allocation | Minimum and maximum quotas for subjects and objects | Should there be a minimum quantity of resources is kept in reserve for certain users or tasks? Should there be limits on the maximum amount of resources allocated to a user, a subject, or an operation ("quota")? For which entities and resources? Are there any size limits on security system data (e.g., the audit log files), and what happens if they are (almost) exceeded? |
| FPT_RIP | Residual information protection | Enforcing permanent deletion of data, preventing unauthorized recovery. | How can we ensure that deleted data is not recovered and read by an unauthorized user? Does the system prevent deleted data from reappearing in core dumps, audit logs or error messages? |

## Class FMI — Miscellaneous

This class defines miscellaneous components that did not really fit into another class of the CC but warranted no class of their own.[7]

**Table 10** Functional class FMI according to [27]

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FMI_RND | Random number generation | Requirements for the quality and secure storage of random numbers | Are there any prescriptions for the statistical and cryptographic quality of random data (e.g. session keys, nonces), for example, with respect to irregularity, proper distribution, and unpredictability? Is a minimum quality enforced by the system (verification)? How should random data be generated, stored, and exchanged to prevent eavesdropping or manipulation? |

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FMI_TIM | Timestamps | Requirements for the accuracy time stamps | Do we need accurate timestamps? Do we need synchronized clocks between users, subjects, or objects? Would it harm the system if time-stamps could be tampered with? |

## 4.4 Additional functional components for AmI systems

As mentioned before, the Common Criteria were originally not conceived for application in an AmI context.[8] Therefore, they fail to cover some AmI-specific security properties. Below, we discuss some of the missing properties.

### Energy resources

For a system with small, mobile components energy supply is an important aspect for system availability. By maliciously exhausting the system's energy resources, a denial-of-service attack can be launched easily.

Two components of the CC address the need to protect system resources against monopolization and misuse:

- «Priority» (FPT_PRI):
  When resources become overtaxed, the TSF shall ensure that certain subjects or operations are served by the system with priority, without undue interference by low priority subjects or operations.

- «Resource allocation» (FPT_RSA):
  The TSF controls the use of resources by subjects or objects such that unauthorized monopolization of system resources is prevented.

The CC explicitly list processing, storage and communication resources, but leaves room for *<other resources>* in the requirements templates. In principle, energy can be treated simply as a special resource type to which components FPT_PRI and FPT_RSA apply. However, we recommend to add a separate component to address the energy problem, for several reasons:

7 In deviation from the original standard [27] we omitted the component «Choice» (FMI_CHO), which is actually not a security component but merely a technical «meta-component» for the composition of other security functional components.

8 An obvious reason why the CC assume conventional system configurations is because traditionally system designs for safety/security considered a static, predictable architecture as a necessary precondition for strong availability and integrity guarantees: It is still an open question whether AmI characteristics can be reconciled with security needs in a satisfactory way.

- From a traditional viewpoint, energy is easily overlooked as scarce resource that should be actively prioritized and managed under security constraints.

- Compared to processing power, storage capacity, or communication bandwidth, it is more difficult to reason about energy consumption and energy reserves at software design level; therefore, it may not be possible to solve the problem simply by specifying minimum or maximum quotas for energy resources, as it is suggested by the CC for the conventional system resources. Requirements for energy consumption most likely require more subtle approaches.

Thus, we suggest the following extension to the CC:

**Table 11** Extension to the functional class FPT w.r.t. the protection of energy resources

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FPT_ENG | Protection of energy resources | Requirements to protect battery power and other scarce energy resources form malicious exhaustion | Is there a danger that the energy resources of a system component may be exhausted prematurely by a malicious adversary (e.g., by flooding a device with unsolicited communication or operation requests)? Is there a way to detect such energy exhaustion attempts, and how should the system react in response to these attacks? |

### Privacy, anonymity and observability

Privacy is a major concern in all kinds of Ami scenarios because in an «intelligent» environment where the user is surrounded by different types of sensors and information processing devices, the system will collect a large amount of private information about the user's identity and habits. Due to the autonomy of the components part of this information is exchanged, stored, and processed without explicit user consent. By linking information from different sources some subtle conclusions may be drawn about the user.

Existing IT security standards are mostly geared to the problem of data privacy, that is they emphasize the need to protect information that is explicitly encoded as binary data. In an AmI system, privacy also has to cover implicit information, that is data that is not part of the application's data model, but is deducible indirectly from available sensor readings or other observable system properties.

Several functional components of the CC are related to privacy issues:

- «Access control» (FDP_ACC):
  The TSF shall permit or deny certain operations for certain subjects on certain objects.

- «Unlinkability of operations, subjects, and objects» (FDP_UNL):
  The TSF shall ensure that certain subjects are unable to determine whether certain operations were performed by the same subject, whether certain subjects are related to the same object, and whether certain objects are related to the same object, subject, or operation.

- «Unobservability of operations» (FDP_UNO):
  The TSF shall ensure that certain subjects are unable to obtain more than a predefined amount of information from observing certain operations on certain objects by certain subjects — in the extreme case: no information at all, that is, total unobservability.

- «Confidentiality of imported/exported data» (FCO_CID, FCO_CED):
  The TSF shall protect the confidentiality of certain data that is provided to users bound to certain subjects, or that is provided to the system by users bound to certain subjects.

- «Unobservability of export» (FCO_UNE):
  The TSF shall export certain data to certain users in such a way that it cannot be observed to which user export is taking place, or even that it cannot be observed whether that data are being exported or not.

Note that the first three components (FDP_ACC, FDP_UNL, FDP_UNO) only address subjects, objects, and operations inside the system, and only data and functionality that is explicitly modeled in software; the other components (FCO_CID, FCO_CED, FCO_UNE) relate only to users who are bound to the system, and to data explicitly encoded in software. All these components fail, for example, to address location privacy because the location of a user may be deducible simply by observing (e.g., direction and signal strength of) any data transmission, even though location information is never explicitly communicated or stored by the system.

Location privacy, one of the most difficult privacy problems in mobile or ambient applications, illustrates a more fundamental problem of complex, distributed systems: The prevention of covert channels. In traditional information processing, the problem of covert channels is well-known, too, but ignored for all but the the most sensitive systems — typically belonging to the field of military intelligence. In most applications, the dangers of information disclosure by covert channels can be ignored because

- either the effort to establish such a channel is too high,

- or the value of the information that leaks out is negligible.

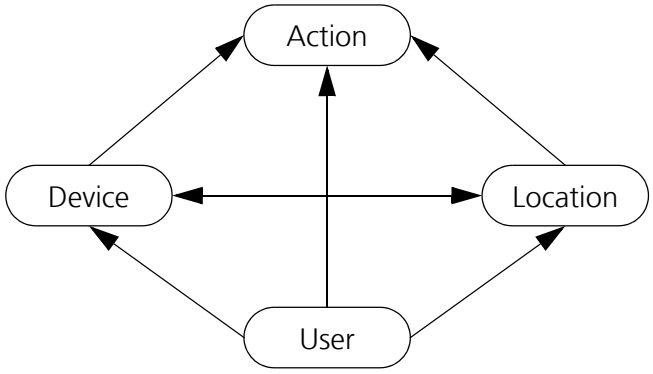


**Figure 2** The Freiburg privacy diamond

There are, however, some application domains where covert channels are a major concern. For example, a smartcard design has to make sure that the bit values of a stored private key are not leaked by varying energy consumption during an encryption operation. Although costly to obtain, information about encryption or signing keys is so valuable that it is worth the effort to obtain it via covert channel analysis.

In an AmI scenario, covert channels gain importance mainly because it is so easy to observe data exchanged at wireless interfaces. Even if each piece of observable information may be of limited value, the costs of observing it are low, and by combining multiple observations the observer may be able to deduce quite sensitive private data.

In [28] Zugenmaier et al. propose an interesting model for the possible interference between identity, action, device, and location of a system user, the so-called privacy diamond (Figure 2): To link a user with an action, we may either observe a direct relation between the user ID and the action, or we may deduce the relation indirectly, for example:

- We may determine the location from where the action request originated, and we may determine the device that is currently at this location. If we are then able to link the device to its user, anonymity of the action is broken.

That is, to preserve anonymity of system usage we must prevent the observer from constructing a directed path between «User» and «Action» in the diagram of Figure 2: We may, for example, prevent the observer from linking users with devices and locations, or we may prevent actions being attributed to devices and locations — whatever is required to interrupt the path between «User» and «Action». Note that the privacy diamond can be applied probabalistically. If we assign to each edge the probability that an observer may establish the corresponding relation, we can compute the overall probability that anonymity of a user transaction is disclosed.

To better reflect privacy, anonymity, and observability requirements under AmI conditions, we suggest the following extension to the CC:

**Table 12** Extension to the functional class FCO w.r.t. privacy.

| Label | Name | Scope | Guiding questions |
|---|---|---|---|
| FCO_EME | Electromagnetic emanation | Requirements for emitted signals not to convey private or confidential information, neither directly nor — most important — indirectly. | Are there any conclusions that can be drawn from the fact that the system is exporting data or emitting any signals — independently of the content being emitted? Could the derived information compromise privacy or confidentiality (e.g., a user's or device's location or presence)? |

Of course, it is far from trivial to avoid information leakage by electromagnetic emanation in an AmI environment, but there are a few standard techniques:

- The system may hide true information in an encrypted stream of «white noise», that is messages that do not contain any data but only padding bytes. Thus, the observer is unable to determine actual information exchange.
- A component may even hide itself in a group of dummy components: Each component sends and receives signals, but dummy components perform only dummy communication that is ignored by the other members, which keep the secret of the identity of dummies.
- We may reduce the probability that electromagnetic emanation is observable by reducing the signal strength to the minimum or by using directional radio, for example.
- A component may completely avoid to emit signals by only *listening* to information broadcast by the system, and computing required information locally. In this way, the device may conceal its location until the user is willing to trust the environment and reveal her presence by actively participating in wireless communication.

Whether these proposed techniques are viable solutions depends on the specific circumstances. In many cases, they are simply to costly relative to the privacy that can be gained.

# 5 Summary and Outlook

The analysis of security requirements is still dominated by human-centric, creative processes. At present, there is no automated substitute for experienced subject matter experts and security analysts, no «mechanical» approach to systematically eliciting all relevant security issues of a system design. This is true for classical IT systems, and even more for the new class of ambient systems, which pose new types of security threats that are typical for distributed, embedded technology.

Although a thorough understanding of the application domain and of the employed technology is indispensable, a body of best practices, guidelines, modeling techniques and documentation templates are available to assist the analyst.

## 5.1 Available guidance for AmI security

In general, security analysis in an AmI context is not fundamentally different from the analysis of a standard information processing system. At the process level, the same generic process areas and capabilities are required, and the same practices apply [1, 2, 3]. The main difference between conventional IT systems and AmI systems are:

- AmI systems are exposed to different types of threats.
- AmI systems are more restricted in their abilities to counter these threats due to resource limitations and their inherent openness.

Thus, we mainly need extended vulnerability taxonomies and more adequate description techniques for the specification of AmI-specific security requirements, and more comprehensive catalogues of corresponding design solutions.

Three threats, in particular, plague AmI designs. The first is denial-of-service attacks, which target the scarce resources of small, mobile components that are the ingredients of AmI systems. Because there are so many resource limitations, it is so difficult to protect the system from DoS attacks systematically and effectively.

The second severe threat is insufficient friend/foe detection. By their very nature, AmI systems require open designs where new components can be integrated seamlessly. This conflicts with a paradigm of general distrust of strangers that is prevalent in today's security (and also safety) designs.[9] Furthermore, the standard methods for proper authentication and authorization are often too ponderous for a flexible, mobile AmI context that enables ad-hoc cooperation of diverse components.

The third important threat is privacy violations. Due to extensive wireless communication and limited capabilities of small devices to apply proper encryption and other privacy protection techniques, disclosing private data is comparatively easy.

From the viewpoint of requirements analysis we can simply extend our collections of vulnerabilities, failure modes, attack patterns or security functional components to put more emphasis on these AmI peculiarities. However, at implementation level there seems to be no schematic solution to these problems: Countermeasures against DoS, authenticity or privacy infringements tend to be very context-dependent. Apart from some general structural design principles, little advice can be given to the designer how to handle such vulnerabilities in a concrete hardware/software environment. Security patterns are an initial step in the direction of canonical security solutions, but so far, available patterns do not specifically address embedded system's needs. Expressing the security needs of diverse embedded AmI components occurring in a wide spectrum of application contexts in the format of generic security patterns is in fact a challenging task.

In safety engineering model-checking has proven its value for the analysis of embedded systems, and for the validation of requirements. It even received more attention in the context of embedded designs than in the field of information processing, because small components better lend themselves to the resource-intensive computations required for model-checking, whereas standard IT is often too complex to be manageable, and has generally less severe safety needs. With respect to security analysis, however, model-checking had less impact so far. Although it is suitable to model some selected aspects of security (e.g., correctness, liveliness, and fairness of communication protocols, (un-)reachability of certain failure modes — all these are properties already covered by safety), it is not yet considered a standard tool for security analysis. Existing approaches [24, 25] are still too clumsy, too restricted in their expressiveness, and they require too much mathematical expertise to be widely acceptable among requirements engineers. And part of the analytical power of model-checkers that has contributed to their success in the safety domain is lost when it comes to security modeling because there are no quantitative measures for security comparable to, for example, failure probabilities as we know them for safety.

9 Current high-assurance architectures inevitably postulate a static system design, where all components and their properties are known at deployment time.

## 5.2 From security requirements to the implementation of secure AmI systems

In this survey our focus is on requirements analysis. We presented techniques that help the designer to determine and to specify the required security properties of the system under construction. Provided that this has been done properly and that all security issues have been identified, the next step is to refine requirements and design, and to derive a fine-grained specification that can be implemented.

In AmI systems step two seems to be the harder task. Although we may very well know the security property that we need to provide, we often have problems to find a viable path for its implementation. There seems to be an urgent need for generic security building blocks that can be reused with little effort to solve reoccurring design problems. It is an open question what abstraction level would be best suited for such security design patterns, how domain and context-specific such patterns need to be, and for what problem classes suitable patterns exist at all. As long as stringent resource limitations force the designer to optimize for maximum resource efficiency, it is unlikely that generic patterns are widely applicable, but custom-build, context-aware solutions will be required.

## 5.3 Combining safety engineering with security engineering

Superficially, safety and security appear to be very similar design problems, or as Jürjens once put it [29]: «Safety is security against stupid adversaries — security is safety for the paranoid». Based on this characterization, we may conclude that we could simply take a modeling technique for safety and replace the failure model with a suitable adversary model to obtain the corresponding security modeling technique.

In fact, this approach is suggested in [29] and has been applied to several design problems, for example to analyze a specification for an electronic purse. However it is doubtful that the relation between security and safety is such a trivial one, for several reasons:

- Some security requirements have no counterpart in safety, and require completely different solutions than the typical safety approaches. For example, confidentiality and privacy are unique security requirements; and component redundancy, the standard approach to improve the safety of a system, even adversely affects confidentiality because the more often information is duplicated, the more likely it will be disclosed [30].

- Security incidents are more than simply «very unlikely» failures that have been consciously provoked by a smart adversary. Modeling security threats is fundamentally different from modeling safety failures because there is no meaningful way to assign probabilities to security breaches.

- Although it may be tempting to model security vulnerabilities as simple system failures with arbitrarily low probability (hence the «paranoid» aspect of security modeling mentioned by Jürjens), this simplification misses an important aspect:

  – The safety «adversary» (mishap, accident) may «choose» a very unlikely failure scenario, but he is still subject to the rules of the failing system.

  – In contrast to that, the security adversary can — and typically will — violate all rules to attack the system in an «unfair» manner. He may even invalidate the laws of nature (at least at the logical level of inputs and stimuli provided to the system) to circumvent existing safeguards.

For example, the adversary may attack a smartcard by increasing and decreasing the clock speed of the circuitry at judiciously chosen times to force the processor to skip a specific test instruction, for example the one that discriminates correct from incorrect PIN inputs. This problem exists and has been exploited [31], but it is not visible at the logical level of the system design: Synchronized clock manipulation is an «illegal move» that is not part of the rules of the game! Even if the safety design of the smartcard considered «corruption of clock signals» as a potential failure mode, it would probably not spend a thought on this particular type of corruption, simply because it is impossible that a random failure could ever fit so perfectly to the timing of the instruction counter and the ALU that it would cause this specific misbehavior.

Unfortunately, playing against the rules is the origin of the severest threats that a system may face — something completely unexpected that leaves the system without any protection.[10] And this is exactly where the safety–security analogy breaks down.

Nevertheless, safety and security analysis may inspire each other as far as the above-mentioned analogy holds. For example, (security) attack trees have much in common with (safety) fault trees. We may exploit these similarities to apply safety tools — maybe after some adaptation — to security problems, or vice versa. Whether it is possible in the long run to provide a unified framework for safety and security analysis of AmI components and systems remains an interesting field for further study.

10 When hackers first exploited buffer overflows for so-called «stack smashing» attacks and managed to insert arbitrary code in the instruction flow, the software community was startled because the threat of stack-smashing is not present at the source-code level, but depends on clever exploitation of compiler artefacts. And even today, decades after the first stack-smashing incidents, buffer overflow attacks are the most frequent and most often exploited software vulnerabilities — programmers and testers have still not become fully aware to this type of threat.

## 5.4 Conclusion

Today most AmI systems are laboratory prototypes rather than established IT products. Designers and engineers of such systems still struggle to define the appropriate functionality that would be most useful for the user, and to find means to implement their visions. As it is not clear how AmI systems should look like and how they will eventually operate, their security needs and required security mechanisms are equally unclear. We simply lack experience with the deployment of AmI, thus we have problems to specify a canonical collection of security requirements!

With respect to functionality and security, AmI systems clearly face conflicting design goals. On the one hand they should offer a maximum of unintrusiveness and ease of use, and we do not want to be bothered with low-level configuration issues and operational details; on the other hand we want to fully control to what extent we are willing to reveal private information, and to whom. On the one hand AmI should enable seamless integration of new system components and the system should be open for anyone; on the other hand we are concerned about system integrity and require that only «well-behaved» devices should participate. On the one hand we want to exchange useful information with arbitrary strangers without lengthy establishment of trust relationships, on the other hand we are not willing to rely on data from dubious sources. On the one hand, AmI should be usable by everyone including children and elderly people without any IT knowledge; on the other hand our AmI scenarios raise critical issues about applicable security policies that even experts have difficulties to grasp … It is not at all clear how to reconcile these conflicting goals, and how to find the best balance between security and usability — an old problem in traditional IT systems that is exacerbated by AmI characteristics.

However, we should not repeat the mistake to restrict our attention to functionality, hoping that «security issues» could be settled independently at later design stages. Although it is difficult, we have little choice but to include security considerations from the beginning as an essential part of our design.

# List of Abbreviations and Acronyms

| | |
|---|---|
| ALU | Arithmetic/Logic Unit |
| AmI | Ambient Intelligence |
| CC | Common Criteria |
| CTCPEC | Canadian Trusted Computer Product Evaluation Criteria |
| IEC | International Electrotechnical Commission |
| ISO | International Organization for Standardization |
| IT | Information technology |
| ITSEC | (European) Information Technology Security Evaluation Criteria |
| PIN | Personal identification number |
| STRIDE | Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privileges |
| TCSEC | (U.S.) Trusted Computer Security Evaluation Criteria («Orange Book») |
| TOE | Target of evaluation |
| TSF | TOE security functionality |

# References


[1] The International Systems Security Engineering Association (ISSEA): Software Security Engineering — Capability Maturity Model.
http://www.sse-cmm.org/index.html

[2] Information Security Form (ISF): The ISF's Standard of Good Practice.
http://www.isfsecuritystandard.com/index_ns.htm

[3] International organization for Standardization: ISO/IEC 17799:2005 Information Technology - Security Techniques - Code of Practice for Information Security Management.
http://www.iso.org/iso/en/prods-services/popstds/informationsecurity.html

[4] S.W. Smith and E.H. Spafford: Grand Challenges in Information Security: Process and Output. IEEE Security and Privacy, Vol. 2, No. 1, pp. 69–71, January/February 2004
http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=8013&year=2004

[5] Alistair Cockburn: Writing Effective Use Cases. Addison-Wesley, Boston 2001

[6] D. Kulak and E. Guiney: Use Cases: Requirements in Context. ACM Press, 2000

[7] Sindre and A.L. Opdahl: Eliciting Security Requirements by Misuse Cases. Proc. 37th Intl. Conf. on Technology for Object-Oriented Languages and Systems (TOOLS-37), pp. 120–131, IEEE Press, November 2000
http://ieeexplore.ieee.org/iel5/7163/19277/00891363.pdf?arnumber=891363

[8] G. Sindre and A.L. Opdahl: Templates for Misuse Case Description. Proc. 7th Intl. Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ'2001), Interlaken, Switzerland, June 2001
http://www.ifi.uib.no/conf/refsq2001/papers/p25.pdf

[9] I. Alexander: Misuse Cases: Use Cases with Hostile Intent. IEEE Software, Vol. 20, No. 1, pp. 58–66, January 2003
http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1159030

[10] P. Hope, G. McGraw, and A.I Antón: Misuse and Abuse Cases: Getting Past the Positive. IEEE Security and Privacy, Vol. 2, No. 3, pp. 90–92, May/June 2004
http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=8013&year=2004

[11] J.H. Saltzer and M.D. Schroeder: The Protection of Information in

Computer Systems. Proceedings of the IEEE, Vol. 63, No. 9, pp. 1278–1308, September 1995
http://web.mit.edu/Saltzer/www/publications/protection/

[12] J Viega and G. McGraw: Building Secure Software — How to Avoid security Problems the Right Way. Addison-Wesley, September 2002

[13] H. Peine: Rules of Thumb for Developing Secure Software. IESE Report No. 038.04/E, Fraunhofer IESE, Kaiserslautern, 2004
http://bib.iese.fhg.de/reports/public/2004/iese-038%5F04.pdf

[14] H. Peine: 20 Regeln zur Entwicklung sicherer Software. Virtuelles Software Engineering Kompetenzzentrum (VSEk), 2004
http://www.softwarekompetenz.de/servlet/is/22055

[15] C. E. Landwehr, A. R. Bull, J. P. McDermott, W. S. Choi: A Taxonomy of Computer Program Security Flaws, with Examples. ACM Computing Surveys, Vol. 26, No. 3, pp.211-254, September 1994
http://chacs.nrl.navy.mil/publications/CHACS/1994/1994landwehr-acmcs.pdf

[16] T. Aslam: A Taxonomy of Security Faults in the Unix Operating System. Master's Thesis, Purdue University, Department of Computer Sciences, August 1995
http://ftp.cerias.purdue.edu/pub/papers/taimur-aslam/aslam-taxonomy-msthesis.pdf

[17] M. Bishop: A Taxonomy of Unix System and Network Vulnerabilities. Technical Report CSE-9510, Department of Computer Science, University of California at Davis, May 1995.
http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-95-10.pdf

[18] M. Howard, D. LeBlanc: Writing Secure Code (2nd edition). Microsoft Press, 2002

[19] B. Schneier: Attack Trees. Dr. Dobb's Journal, V. 24, No. 12, pp. 21–29, December 1999
http://www.schneier.com/paper-attacktrees-ddj-ft.html
http://www.softwarekompetenz.de/servlet/is/22033/

[20] G. Hoglund, G. McGraw: Exploiting Software. Addison-Wesley, 2004

[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[22] The Open Group Security Forum.
http://www.opengroup.org/security/gsp.htm

[23] SecurityPatterns.org
http://www.securitypatterns.org

[24] S. Konrad, B.H.C. Cheng, L.A. Campbell, R. Wassermann: Using Security Patterns to Analyze Security Requirements. Second International Workshop on Requirements Engineering for High Assurance Systems (RHAS03), Monterey Bay, California, September 2003

http://ftp.cse.msu.edu/~konradsa/Publications/rhas03.pdf

[25] Jan Jürjens: UMLsec Homepage.
http://www4.in.tum.de/~umlsec/

[26] D.M. Kienzle, M.C. Elder, D.S. Tyree, J.Edwards-Hewitt: Security Patterns Template and Tutorial. February 2002
http://www.modsecurity.org/archive/securitypatterns/dmdj_template_and_tutorial.pdf

[27] Common Criteria for Information Technology Security Evaluation (Version 3.0, Revision 2). CCMB-2005-07-002, July 2005
http://www.commoncriteriaportal.org

[28] A. Zugenmaier, M. Kreutzer, G. Müller: The Freiburg Privacy Diamond: An Attacker Model for a Mobile Computing Environment. 13. Fachtagung Kommunikation in Verteilten Systemen (KiVS 2003), Leipzig, February 2003, VDE Verlag 2003
http://www.inf.ethz.ch/vs/publ/se/FPDinKiVS03.pdf

[29] Jan Jürjens: Developing Safety- and Security-critical Systems with UML. DARP Workshop, Loughborough, May 2003
http://www4.in.tum.de/~juerjens/papers/darp03.pdf

[30] F.B. Schneier, L. Zhou: Implementing Trustworthy Services Using Replicated State Machines. IEEE Security & Privacy, Vol. 3, No. 5, pp. 34–43, September/October 2005.
http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=8013

[31] D. Naccache: Finding Faults. IEEE Security & Privacy, Vol. 3, No. 5, pp. 61–65, September/October 2005.
http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=8013

# Document Information

| | |
|---|---|
| Title: | Systematic Elicitation of Security Requirements in the Context of AmI Systems |
| IESE Report: | 098.05/E |
| Date: | November 2005 |
| Version: | 1.0 |
| Status: | Final |
| Distribution: | Public |