Towards a quantitative assessment of security in software architectures

Artsiom Yautsiukhin[2*] Riccardo Scandariato[2] Thomas Heyman[2] Fabio Massacci[1]
Wouter Joosen[2]

[1] - University of Trento, TN, Italy
[2] - Katholieke Universiteit Leuven, Belgium

Abstract

Software patterns are key building blocks used to construct the architecture of a software system. Patterns also have an important role during the architecture assessment phase, as they represent the design rationale, which is central to evaluation. This work presents a quantitative approach to assess the security of a pattern-based software architecture. In particular, security patterns are used to measure to what extent an architecture is protected against relevant security threats. To this aim, threat coverage metrics are associated to security patterns and an aggregation algorithm is proposed to compute an overall security indicator. The proposed approach helps in comparing design alternatives and choosing the best candidate.

1 Introduction

The problem of assessing the level of security of a software system is still largely open. This is because the complexity of todays systems is becoming cumbersome. Systems are growing bigger and are often the result of tangled composition. They are more interconnected and more heterogeneous in the technologies they adopt. In this complex context, the security discipline lags behind in the area of quantitative assessment methods. Most of the literature focuses on the low end of the spectrum, i.e., on assessing the security posture after deployment.

The software architecture discipline is able to counteract the mentioned limiting factors factors to the assessment of security. It is well known that the software architecture is the key place where software qualities (including security) are embedded in a software system. It is also recognized that the software architecture provides a *manageable* abstraction, in terms of size and complexity, to perform an effective (often critical) quality assessment on the system in a holistic way. In this respect, some approaches exist, but they are qualitative in nature [3]. To the authors' knowledge, no approach exist that quantitatively assesses security at the software architecture level.

Software patterns are key building blocks used to build the architecture of a software-intensive system [1]. Patterns also have an important role during the architecture assessment phase, as they represent the design rationale, which is central to evaluation [3]. The main contribution of this paper is to provide a quantitative approach to assess the security of pattern-based software architectures. In particular, security patterns are leveraged to measure to what extent an architecture is protected with respect to relevant security threats. To this aim, threat coverage metrics are associated to security patterns and an aggregation algorithm

*The work by Artsiom Yautsiukhin was partly supported by the EU-IST-IP-SERENITY and IST-FP7-IP-MASTER projects

is proposed to compute an overall security indicator for the software architecture that instantiates the patterns. The proposed approach helps in comparing design alternatives and choosing the best candidate. In the context of this work, the fit criterion for the best candidate is represented by the best protection against the most dangerous threats.

The rest of this paper is structured as follows. First, in Section 2 we introduce the case study we use as an example throughout the paper. Then, in Section 3 we sketch the results of our previous works on which this paper is based. Section 4 describes how to determine the threat coverage of a security pattern. In Section 5 we show how the values can be aggregated and present an algorithm for that purpose. The results are validated on the case study in Section 6. Section 7 discusses the proposed approach. Related work (Section 8) and concluding remarks (Section 9) close the paper.

2 Case study

We use the following case study as a running example to illustrate our approach. An on-line publishing system has to be designed and the system should be able to receive news articles from *journalists* and advertisements from *advertisers*, allow *editors* to design upcoming editions of the newspaper using the news and advertisements, and provide access to the editions to *customers*. Obviously, next to providing the primary functionalities, there are a number of security-related issues which have to be encompassed in the system design, such as availability of the service, integrity of news items, and authentication of actors (journalists, customers, and so on).

3 Background

Previous work has studied the existing security pattern landscape and has shown how patterns can be used to fulfill security requirements in software design [4, 14, 18]. In this section, some of the previous results that are relevant to the present work are outlined.

Reuse of well-know and time-tested solutions (i.e., patterns) is a widely accepted software engineering practice. Patterns are even more relevant in the *secure* software engineering discipline because of the recognized principle of reusing community resources to avoid reinventing ad-hoc solutions from scratch.

Unfortunately, not all published security patterns are of straightforward use to the software developer, as they do not belong to the proper level of abstraction and lack a sufficiently detailed description. In order to address this problem, a preliminary list of patterns[1] which are directly usable to the software engineer (so-called *core patterns*), have been collected in [18]. The security patterns in this repository were used in selection of patterns for the case study presented in this paper.

To make optimal use of the pattern inventory, a methodology is needed that supports the selection of the right set of patterns for the (security) requirements at hand. Such methodology is presented in [14]. It enables the traceability of selection decisions as far as patterns and requirements are concerned. Such a relation is needed to perform the evaluation of the resulting pattern-based architecture, i.e., to determine how well the requirements are satisfied with the instantiation of a specific set of patterns.

The link from security requirements to security patterns is not obvious, as requirements are domain-dependent while patterns (by definition) are not. To solve this mismatch, security objectives are used as 'intermediaries'. Indeed, security objectives provide an abstraction that is conceptually close to requirements. However, objectives are domain-independent and can be completely enumerated. This allows attaching patterns to security objectives independently from the domain of their application.

[1] The remainder of this paper focuses on security patterns only. We refer to them with 'pattern' for brevity.

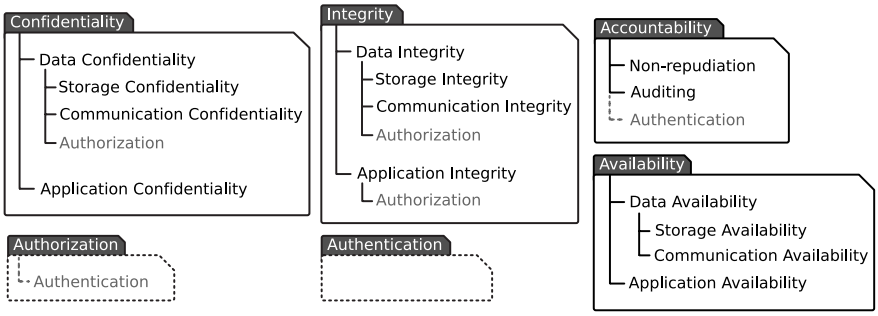

Figure 1: Decomposition of security objectives

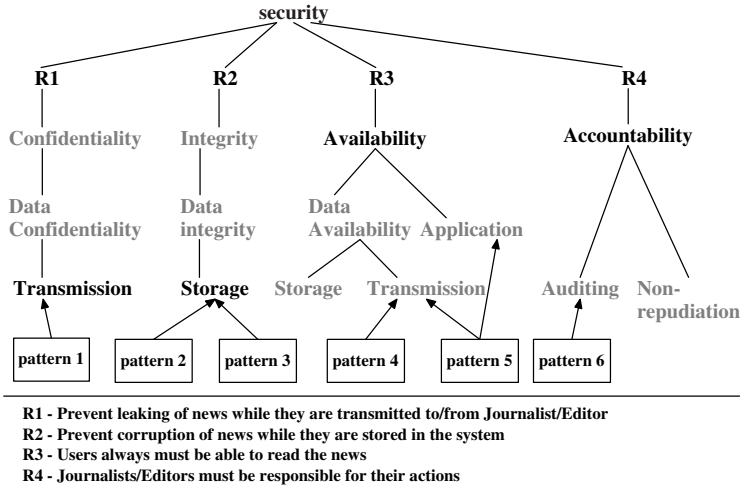

Figure 2: Requirements Tree for the running example.

We use the security *Objectives Decomposition* that is shown in Figure 1, which is based on [14]. There are four top security objectives (authentication and authorization are auxiliary objectives). These four objectives are often used to describe security as a whole [16, 8]. The high level objectives are, in turn, decomposed according to the areas of their applicability (e.g., application, data in transmission, stored data). This level of decomposition helps to sort patterns according to what (and in what context) they should protect. Further decomposition requires more details about the design of the system, which are not available at this high level of the analysis.

Each security requirement (or group of related requirements) has a motivation and rationale that lead to its definition. In general, it is straightforward to interpret such a rationale as a security objective associated to the requirement. By creating an association between requirements and objectives in the problem domain and between objectives and patterns in the solution domain, a link is establish that can be leveraged in two directions. Top-town, it drives the selection of patterns from requirements and bottom-up, it supports the traceability of design solutions back to the requirements.

A visualization of the above concepts is given in Figure 2. For ease of reference, in the following of this paper we refer to this as the *Requirements Tree*. In particular, the figure presents the relationship between requirements, objectives, and patterns for the case study of Section 2. Every requirement is connected with a security objective as follows. The analyst chooses the top objective from Figure 1 (integrity, confidentiality,

etc) to which a requirement belongs, and then the relevant sub-objective, if the requirement is fine-grained enough (see R1 and R2 in Figure 2). If the requirement is very high-level and allows the analyst to specify only a top objective, then all sub-objectives should be chosen implicitly (see R3 and R4 in Figure 2). Finally, the patterns which contribute to the satisfaction of the low level objectives are identified. The patterns are shown just to give the complete picture to the reader and will be explained further on in the article.

Example 1 In Figure 2, requirement *Prevent corruption of news items while they are stored in the system (R2)* is connected to *Integrity of stored data*. On the other hand, *Users always must be able to read the news (R3)* is associated to the *availability* top objective. This means that the availability of news stored in the database, its transmission to the customer, and the provisioning of data should be protected against denial-of-service attacks.

4 Metrics for Security Patterns

In our approach, the assessment of a security-enhanced architecture is performed in two stages. First, we must determine the level of protection each group of patterns provides to the architecture. Second, these low level contributions need to be aggregated, using the Requirements Tree, into an overall security indicator. In this section we deal with the first problem and Section 5 describes the second one.

4.1 Mapping threats to objectives

The level of protection of a software system against various classes of threats represents the result of the implementation of security requirements. In order to measure protection against threats, a reference list of threats must be identified. In this work, we use the threats included in Microsoft's STRIDE methodology [6]. This choice is driven by two reasons. First, the threats mentioned in STRIDE are meant to be used for the threat modelling activities performed during the architectural phase of the software development life-cycle. Therefore, they naturally fit into the context of this work. Second, STRIDE is a well-known and mature technique that has been successfully adopted in several software projects. STRIDE might not be exhaustive concerning the range of potential architectural threats. This issue, however, is not in scope for this work.

In STRIDE, threats are organised into tree structures, called *threat trees*. As shown in Figure 4, each tree has a generic threat class as its root (e.g., "DoS against a process") that is decomposed into more specific threats via AND/OR branches. The decomposition can proceed further as necessary, but is limited, on average, to three levels. The resulting trees are similar to the well-known attack trees [15] although STRIDE threats are more general.

Each threat tree can be unambiguously mapped to a corresponding elementary objective of the Objectives Decomposition as shown in Figure 3[2].

4.2 Rating threat severity

Each threat has a different severity degree. For instance, vulnerabilities leading to some threats can be easier to discover. Similarly, carrying out a specific threat scenario can be costlier than others. We assign weights to each elementary threat to capture this difference.

The weights can be determined by a risk assessment method that assigns severities to threats. In this work our preference went naturally to Microsoft's DREAD [12], the companion method of STRIDE. The DREAD approach considers five factors contributing to the severity of a threat:

[2]There is only one minor exception: to match the accountability objectives (i.e., auditability and non-repudiation), the branches of the "repudiation of data flow" threat (i.e., repudiation of transactions and repudiation of messages, respectively) had to be considered.

STRIDE threat trees		Elementary security objectives
Spoofing an external entity or process	$\Longrightarrow$	Authentication
Tampering with data store	$\Longrightarrow$	Integrity of stored data
Tampering with data flow	$\Longrightarrow$	Integrity of transmitted data
Tampering with a process	$\Longrightarrow$	Integrity of application
Repudiate message	$\Longrightarrow$	Non-repudiation
Repudiate transaction	$\Longrightarrow$	Auditability
Information Disclosure of data store	$\Longrightarrow$	Confidentiality of stored data
Information Disclosure of data flow	$\Longrightarrow$	Confidentiality of transmitted data
Information Disclosure of a process	$\Longrightarrow$	Confidentiality of application
DoS against data store	$\Longrightarrow$	Availability of stored data
DoS against data flow	$\Longrightarrow$	Availability of transmitted data
DoS against a process	$\Longrightarrow$	Availability of application
Elevation of Privileges for processes	$\Longrightarrow$	Authorization

Figure 3: Threats to security objectives

- **Damage potential (D):** how great the damage of exploiting the threat could be.
- **Reproducibility (R)**: how easy it is to repeat the attack.
- **Exploitability (E)**: how easy it is to exploit the vulnerability.
- **Affected users (A)**: how many users can be affected if the attack is successful.
- **Discoverability (D)**: how easy it is to discover for an attack that the threat presents in the system.

At this stage, we only consider the DREAD factors that are related to the "likelihood" of the threats, i.e., Reproducibility, Exploitability, and Discoverability—here we do not consider the factors related to the "impact", i.e., Damage potential and Affected Users. The impact is brought into the picture later on in Section 5, where the impact due to the failure of a requirement is considered.

As shown in Figure 4, for each of the factors of interest, an analyst should rate the specific threats (in gray) on a scale from 1 to 3 (as in the DREAD methodology). The severity s_i of a specific (i.e., leaf) threat t_i of a root threat t is then computed as the normalized sum of reproducibility R_i, exploitability E_i and discoverability D_i:

$$s_i = (R_i + E_i + D_i) / \sum_{\forall t_i \in t} (R_i + E_i + D_i)$$

Example 2 *For the* denial-of-service against a process *threat tree (see Figure 4), the following severities can be assigned to the specific threats:* Consume application-specific resources *s=7,* Consume fundamental resource *s=8,* Input validation *s=5,* Tampering with on-disk process *s=4. The total sum equals 24. Dividing each resulting number by 24, the final weights are 0.291, 0.333, 0.210 and 0.166.*

The only exceptions to this method are the threats with AND-decomposition branches. For each AND-branch the sum (coverage) is calculated separately. Then the *minimal sum* among these alternatives is used to normalize all values (for all branches). The rationale behind this is that the success of exploiting the root threat depends on successful compromising of *all* branches. This means that in order to protect the system we need to reduce the probability to compromise at least one branch. In other words, the level of protection of the elementary objective depends on the level of protection of the best protected branch, i.e., the branch which is less likely to be compromised. This allows us to consider only one (i.e., the strongest) branch. This exception is illustrated in the following example.

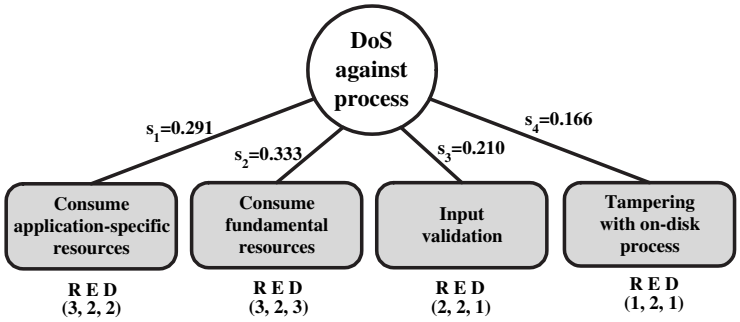

Figure 4: DoS against process tree

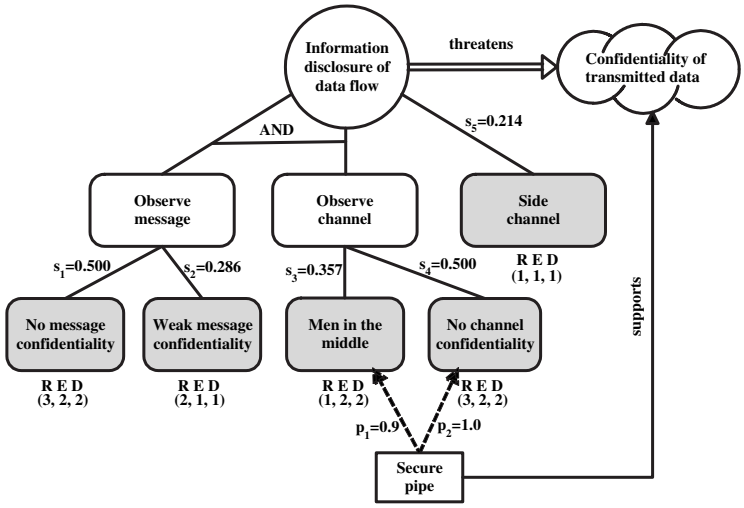

Figure 5: Information Disclosure of data flow tree

Example 3 Consider the threat *Information disclosure of data flow* which violates *Confidentiality of data transmission* in our example (Figure 5). Note that the first two (on the left) and second two elementary threats are AND-branches. That is why the lowest scoring child nodes should be selected when choosing the normalization value and adding the rest values: 7+4=11 vs. 5+7=12. Then adding the score of the another OR branch (i.e., 3) to the minimal sum (i.e., 11) the normalization value becomes: 11+3=14. This results in the following values of 0.500, 0.286, 0.357, 0.500, and 0.213. Note that the sum of all values does not equal 1 (it is greater), since the AND-join implies that only one of the branches should be chosen.

4.3 Determining threat protection

In this section, the security patterns are matched against threats in order to determine the level of protection they offer. This is the building-block metric on which the rest of the approach is based. The level of protection is computed as the prevention of a set of threats associated to the security objectives the pattern contributes to achieve.

Often a pattern provides reasonable protection against a threat but it does not cover it entirely. In some

cases a pattern may offer only marginal protection for a threat. This usually happens when the main contribution of a pattern is to satisfy one objective, and has some side effect on a secondary objective.

The level of protection of a pattern against a threat is determined in a semi-qualitative way, by assigning one of the following values: 0, 0.1, 0.5, 0.9, 1. The idea behind these values is the degree of the attacks (realising a certain threat) that will be prevented if the patterns are applied. These values denote the percentage of threat mitigation, 0 being 'not at all' and 1 being 'completely'. Values 0.1 and 0.9 are preferable to denote side-effects of some patterns and impossibility to satisfy some objectives completely (e.g., availability).

Example 4 *As shown in Figure 5,* SECURE PIPE *provides a 100% protection against* No channel confidentiality *threat of* Information Disclosure of data flow *(1) while cannot give the same certainty for* men-in-the-middle *attack (0.9). Similarly,* REPLICATED SYSTEM *(not shown in the figure) increases the probability to withstand a DoS attacks which are based on* providing invalid inputs *(*DoS against Process*) to a program but does not provide a complete solution (0.5). Finally,* APPLICATION FIREWALL *can provide some side-effect to "access to memory" but this is just a small contribution (0.1).*

In case that several patterns contribute to the same threat, their contributions should be seen as the probability that at least one of the independent protection mechanisms is able to protect the system against this threat, i.e., the 'defense in depth' principle[3]. In other words, if several patterns protect the system against the same leaf threat t_i with coverage $p_{i1}, p_{i2}, ...$ then the overall protection against the threat p_i will be:

$$p_i = 1 - \prod_j (1 - p_{ij})$$

4.4 Computing the overall coverage

Finally, the protection values assigned to leaf threats (via the patterns) can be aggregated to determine the overall coverage for the threat tree. The aggregation can be seen as a weighted function. The computation considers all instantiated patterns supporting a given objective (protecting against corresponding root threat) and produces overall coverage for the objective. The protection values of the patterns (determined according to Section 4.3) are multiplied by the severity of the threats and then summed up:

$$c_t = \sum_{i \in t} p_i * s_i$$

When considering AND-branches, the contributions for each branch are summed up separately, and the difference between the most protected branch (with the lowest sum value) before the implementation of the patterns and after is assigned to the AND-branch. In other words the total coverage of AND branches ($B_1, B_2, ...$) can be calculated as follows:

$$c_{AND} = min(\sum_{i \in B_1} (s_i), \sum_{i \in B_2} (s_i), ...) - min(\sum_{i \in B_1} (s_i * (1 - p_i)), \sum_{i \in B_2} (s_i * (1 - p_i)), ...)$$

The rationale behind this is has been explained in Section 4.2. Note, that after implementation of several patterns another AND branch may become tougher for an attacker.

When the level of protection against the root threat is found, the same number is assigned to the corresponding leaf node at the Requirements Tree.

[3]In general, we acknowledge that the contributions of several patterns to a given objective may not be independent from one another. However, we make this simplification to keep the computation more manageable.

Example 5 For *DoS against a process* threat (see Figure 4), an analyst has defined the following security patters: LOAD BALANCER and REPLICATED SYSTEM. These two security patterns cover *Consume application specific resource*, *Consume fundamental resource* threats 0.5/0.5 and 0.5/0.5 correspondingly. Weights (as it was determined in Example 2) are 0.291, 0.333, 0.21 and 0.166. The overall protection against *DoS against process* threat is the following sum: $0.291 * (1 - (1 - 0.5) * (1 - 0.5)) + 0.333 * (1 - (1 - 0.5) * (1 - 0.5)) + 0.21 * 0 + 0.166 * 0 = 0.468$.

This part of the analysis is hard for non-experts. In particular, the hardest and error-prone steps are: identifying the contributions of patterns to elementary threats, assigning severity degrees to elementary threats, and identifying benefits of patterns (coverage values). On the other hand, these steps are context independent. This means that the results, once achieved, can be (re-)used in any other system. In other words, the analysis can be done thoroughly by highly experienced security experts and then used for analysis in during system development by (non-expert) security designers. Moreover, pattern providers or a Trusted Third Party organisation can do a number of on-site experiments and get realistic statistics which can be used for more thorough definitions of values. This value must be included in the description of the pattern. We assume that the average protection value of a pattern should be the same in all contexts (e.g., being context-independent).

5 Aggregation of metrics

In order to aggregate the values computed for the elementary objectives, we assign weights to the edges in the Requirements Tree. At the top-most part of the tree, weights are assigned to reflect the contribution of each requirement to the overall security goal (see Figure 2). These weights reflect the impact of each individual requirement, for instance, by quantifying the expected monetary loss if the requirement fails.

Example 6 *Consider the four requirements for the on-line publication system (see Figure 2). The following possible losses can be assigned to each requirement, as follows: R1 = 20,000 $, R2 = 80,000 $, R3 = 100,000 $, R4 = 40,000 $ (total = 240,000 $). Thus the normalised weights will be 0.083, 0.333, 0.417 and 0.167, respectively.*

Weights for other edges can be determined in a similar way. The losses caused by failures of objectives are determined and then normalised for every parent-child relationship in the Requirements Tree.

Given this weighted graph, it is possible to verify how well a set of patterns satisfy the security needs of the system-to-be. First of all, we choose a set of patterns which we would like to test. Their contribution towards satisfying the elementary objectives is determined, as shown in Section 4. Using the weights denoting how the selected security patterns impact the satisfaction of elementary objectives, the contributions of each individual pattern can be aggregated. This is done by applying a weighted function to each objective node, starting from the bottom of the Requirements Tree. This process results in a degree of satisfaction of the general security goal (i.e., the value of the fictitious top node in the Requirements Tree). Algorithm 1, built using our previous work on aggregation of security metrics [11], formally describes the core procedure.

6 Experiment

The experiment mentioned in this section is based on a comparison between two student designs of the system described in Section 2. Two lists of selected patterns from each design were compiled (as shown in Figure 6) and are compared with our methodology.

Algorithm 1 Calculation algorithm

Require: $RT = \langle N, E \rangle$: *Requirements Tree: N - nodes(objectives), E - edges*
SL_{leaf} : *protection values of elementary objectives*;
Ensure: $SL[]$ *Satisfaction value of the all security objectives*
Assign received contribution values to elementary objectives
$SL[N_{leaf}] := SL_{Leaf}$;
For each node store number of incoming edges
$SOURCE[N] := |Incoming(E)|$;
Add elementary objectives to a working set
HEAP-insert(N_{leaf});
while HEAPis not empty **do**
take one node from the working set
HEAP-extract(n'); {randomly}
For each outgoing edge from the node
for $\langle n', n \rangle \in Outgoing(n')$ **do**
Reduce the counter for non-traversed incoming edges
decrement(SOURCE[n]);
if all incoming edges are traversed
if $SOURCE[n] = 0$ **then**
compute weighted function for the new node
for all $n'_i.\langle n'_i, n \rangle \in Incoming(n)$ **do**
$SL[n] = SL[n] + L(\langle n'_i, n \rangle) * SL[n'_i]$;
add the new node to the working set;
HEAP-add(n);

Group 1	**Group 2**
Secure pipe	Secure pipe
Secure service Facade	Secure service Facade
Session	Session
Secure Logger	Secure Logger
Audit Interceptor	Audit Interceptor
Authentication Enforcer	Container Managed Security
Authorization Enforcer	Check-pointed System
Application Firewall	Comparator-checked Fault-tolerant System
	Replicated System
	Load Balancer

Figure 6: Tested sets of patterns

The teams have chosen two different strategies in selecting the patterns. The first group focuses on selecting a minimal but sufficient set of patterns to achieve the proposed requirements, given only a limited implementation budget. The second team put more effort in providing the best security solution, ignoring implementation cost. To these ends, the first group decided to use an APPLICATION FIREWALL–which contributes to many objectives–when Group 2 decided to use a CHECK-POINTED SYSTEM and COMPARATOR-CHECKED FAULT TOLERANT SYSTEM to increase the integrity of the software, and combine them with a REPLICATED SYSTEM and LOAD BALANCER to raise the protection against DoS attacks.

In Figure 7, the complete Requirements Tree with attached security patterns selected by Group 1 is depicted. The weights attached to the edges represent the importance of the lower objectives to the higher ones. The weights assigned to the edges connecting patterns and elementary objectives have been calculated by the method proposed in Section 4 using STRIDE threat trees. Details are omitted for the sake of simplicity.

Using our approach, the satisfaction of the overall security goal is calculated to be 33,1% for the set of patterns chosen by Group 1. Applying the same strategy to the second set of patterns, the calculated score is

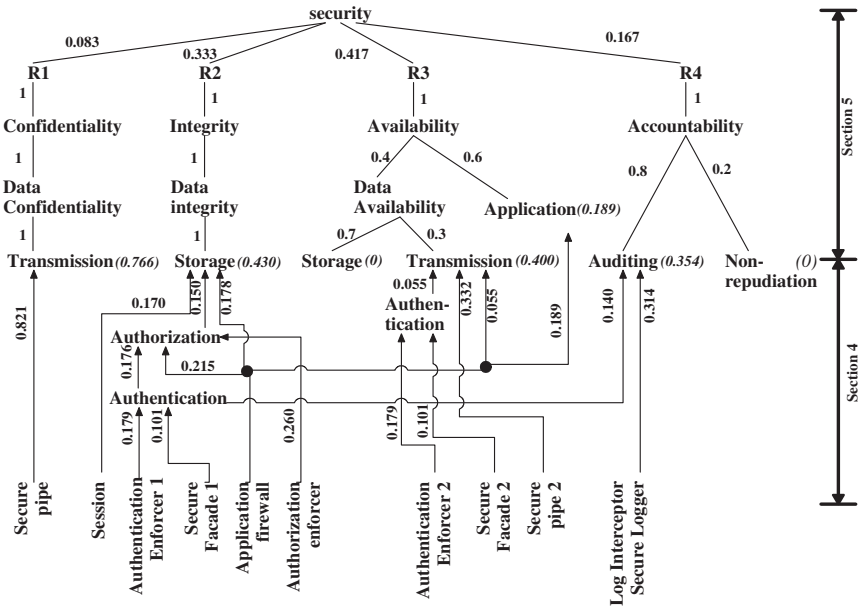

Figure 7: Objectives Decomposition with patterns for the running example.

36,8%. This can be expected, as the second group chose both the REPLICATED SYSTEM and LOAD BALANCER, which contribute greatly to the one of the most important requirement for the publishing system, i.e., *Availability of the application (R3)*. On the other hand, the difference is mitigated somewhat since the APPLICATION FIREWALL has a positive impact on many objectives, including availability. This leads us to the conclusion that, although the second set of patterns provides better protection, the two solutions are comparable in terms of security and the first team designed a more cost-effective solution.

Note that, though the final values are only approximately 35%, this does not indicate that the chosen protection is weak. Even if we assign maximal values to the elementary threats which can be covered on architectural level (i.e., if we assume a 'perfect' architectural solution), the maximum satisfaction level achievable is only 56,8% for this example. Indeed, many threats in STRIDE threat-decomposition trees are only relevant to the detailed design, or even the maintenance phase, of the system. While an argument can be made that the analysis should therefore be conducted without considering design and maintenance threats, the authors are of the opinion that it is important to give a feeling to the software creator about the level of requirements satisfaction at each stage of the software development. A second reason why the computed scores for the examples do not reach 100% quality of protection is that some issues (such as *consume fundamental resources*) cannot be solved perfectly, i.e., there is no such thing as 'perfect security'. A third and final reason is that no patterns are available for many threats which can be solved, in theory, on the architectural level. For example, protection against *Information Disclosure of Data Store* is very poorly represented in the available body of published security patterns.

7 Discussion

The proposed approach applies at the highest level of abstraction of software design in order to identify potential issues as early as possible. In contrast to implemented and deployed software, it is impossible to perform real-world measurements and assess level of protection of the concrete product empirically. For

this reason it is particularly important to reuse context independent measures whose applicability can be extended to several context.

In our approach we used decomposition trees for security objectives and threats. These trees by no means can be proven to be complete, hardly any methodology can provide such guarantee, because the notions of both security and threats are abstract and (most importantly) constantly evolving (e.g., new threats appear). Thus, we have chosen the most complete decompositions that (as of now) suits our purposes.

Further, in the experiments we assigned the contributions and severity levels according to our experience. We must stress that in this paper we propose an approach, but not concrete values. Such values should emerge from either practitioners experience or databases of historical data (which are not available yet).

Finally, further validation is required. However, several challenges are to be faced in this respect. First, in order to get any real-world data one should actually create a software and measure the outcome by comparing several alternatives. Even better, a number of different applications should be tested in order to verify whether the assumptions about context independence really hold. In practice, the software should be exercised in a real setting, which is hard to realise. In alternative, several design variants of a software could be created and evaluated by means of experts. This baseline, then, could be compared with the outcome of the proposed methodology. Another option would be to evaluate existing software application in order to see which one in reality had less design flaws. The problem with this approach is that the design should be based on patterns and the necessary documentation should be. This combination of constraints makes the task of finding suitable case studies particularly tricky (especially in the open source domain).

8 Related work

The problem of security evaluation has received a lot of attention recently. One of the main arguments that makes this problem very important is that the ability to measure security of a system enables choosing the best protection strategy. This leads to the problem of identifying suitable security metrics. Nichols and Peterson [13] proposed and described a number of security metrics which can be used to evaluate the protection against the OWASP top ten vulnerabilities. Although all these metrics are useful for assessment of a specific property, they cannot be used as indicators of the overall security level as they are very specific. Another example of security metric is the attack surface size, which was introduced by Manadhata and Wing [10] as an indicator of the level of security. The method evaluates the functions that the software provides, rather than the security installed to protect the software. Other metrics can be found in [9].

Common Criteria (CC) [7] is a security evaluation method in which a product is evaluated against a specific set of requirements. The product receives an Evaluation Assurance Level (from EAL1 to EAL7) if it satisfies all the requirements for this level. CC is a qualitative method for software evaluation while our approach is based on quantitative assessment though with some subjective values (e.g., level of protection).

CORAS [17] provides a number of ways to model security during software design, and facilitates risk analysis. Risk analysis is more an asset-based methodology, while we focus more on requirements for the software, which allows the application of the approach from the earliest steps of the design.

Also in the area of risk assessment, Clark associates risks to the so-called mission tree, i.e., a tree-based representation of the company specific goals [2]. The evaluation of the risk associated with the mission (i.e., the company top-level goal) is based on the value of the assets at stake.

S. V. Houmb et. al. present a cost-benefit trade-off analysis for aspect-oriented risk-driven development [5]. The idea of the analysis is to use Bayesian Belief Network in order to find the components contributing to Return on Security Investments. Differently from our approach, Houmb proposes a qualitative analysis. Moreover, our main focus is the software base rather than the system and thus we pay more attention to reusable software components (patterns). Also, in our approach we use the same metric (threat coverage) for all nodes of the used tree and thus simplify the analysis.

Finally, the idea of using security patterns to combine security metrics (through security objectives) is also used in [4]. However, the focus of [4] is on assessing and monitoring whether the implemented patterns are working as intended. Our approach evaluates security relative to how well threats are mitigated.

9 Conclusions

In this paper we have presented a novel approach to quantify the security-related qualities of a software architecture adopting security patterns. We use threat coverage as the basic metric for the evaluation. We have also presented an algorithm that aggregates low-level measures associated to these security patterns into an high-level indicator. By applying this algorithm to alternative proposals of software architectures, it is possible to identify the candidate that better satisfies the requirements (in terms of mitigating the threats obstructing the fulfillment of the requirements). The potential of our approach has been illustrated via a case study in the domain of digital publishing.

An interesting research hypothesis is that relationships between patterns (benefit, dependency, impairment, conflict, and alternative) have an impact on the low-level metrics. That is, it is not yet understood how patterns behave, in terms of threat coverage, whenever they are considered in groups. Another interesting issue is the further validation of the approach. These and similar topics are still open to future work.

References

[1] P. Bass et. al. *Software Architecture in Practice*. Addison-Wesley, 2003.

[2] K. Clark et. al. Security risk metrics: Fusing enterprise objectives and vulnerabilities. In *Proc. of IAW*. IEEE, 2005.

[3] P. Clements et. al. *Evaluating software architectures: methods and case studies*. Addison-Wesley, 2002.

[4] T. Heyman et. al. Using security patterns to combine security metrics. In *Proc. of SecSE*. IEEE, 2008.

[5] S. H. Houmb et. al. Cost-benefit trade-off analysis using bbn for aspect-oriented risk-driven development. In *Proc. of ICEECS*. IEEE, 2005.

[6] M. Howard and S. Lipner. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.

[7] International Organization for Standardization (ISO/IEC). *Common Criteria for Information Technology Security Evaluation*. Common Criteria Project Sponsoring Organisations, 2.2 edition, January 2004.

[8] ISO/IEC. *Information technology Security techniques Evaluation criteria for IT security*, November 2001.

[9] A. Jaquith. *Security metrics: replacing fear, uncertainty, and doubt*. Addison-Wesley, 2007.

[10] P. Manadhata and J. Wing. Measuring a system's attack surface. Technical Report CMU-TR-04-102, CMU, 2004.

[11] F. Massacci and A. Yautsiukhin. An algorithm for the appraisal of assurance indicators for complex business process. In *Proc. of QoP*. ACM, 2007.

[12] Microsoft. Improving web application security: Threats and countermeasures. available via `http://msdn2.microsoft.com/en-us/library/ms994921.aspx`, June 2003.

[13] E. A. Nichols and G. Peterson. A metrics framework to drive application security improvement. *S&P*, 5(2):88–91, 2007.

[14] R. Scandariato et. al. Architecting software with security patterns. Technical Report CW-515, Katholieke Universiteit Leuven, Department of Computer Science, 2008.

[15] B. Schneier. Attack trees: Modelling security threats. *Dr. Dobb's journal*, December 1999.

[16] SSE-CMM. *Systems Security Engineering Capability Maturity Model - SSE-CMM Model Document*. CMU, version 3.0 edition, June 15 2003.

[17] K. Stolen et. al. Model-based risk assessment - the coras approach. In *Proceedings of the Norsk Informatikkonferanse*, pages 239–249. Tapir, 2002.

[18] K. Yskout et. al. A system of security patterns. Technical Report CW-469, Katholieke Universiteit Leuven, Department of Computer Science, 2006.