A Reuse-Based Approach to Determining Security Requirements

Guttorm Sindre[1], Donald G. Firesmith[2], Andreas L. Opdahl[3]

[1] *Dept Computer & Info. Science, Norwegian U. Science & Technology*
(on leave at Dept MSIS, Univ. Auckland, New Zealand)
[2] *Software Engineering Institute*
[3] *Dept of Information Science, U. of Bergen, Norway*
guttors@idi.ntnu.no, donald_firesmith@hotmail.com, andreas@ifi.uib.no

Abstract

The paper proposes a reuse-based approach to determining security requirements. Development for reuse involves identifying security threats and associated security requirements during application development and abstracting them into a repository of generic threats and requirements. Development with reuse involves identifying security assets, setting security goals for each asset, identifying threats to each goal, analysing risks and determining security requirements, based on reuse of generic threats and requirements from the repository. Advantages of the proposed approach include building and managing security knowledge through the shared repository, assuring the quality of security work by reuse, avoiding over-specification and premature design decisions by reuse at the generic level and focussing on security early in the requirements stage of development.

1. Introduction

Use cases [1-3] have become popular for eliciting requirements [4, 5]. Many groups of stakeholders turn out to be more comfortable with descriptions of operational activity paths than with declarative specifications of software requirements [6]. As use cases specifically address what users can do with the system, they are most relevant for functional requirements. But lately the application of use cases has also been investigated in connection with security and safety requirements, in the form of *misuse cases* [7-13], a.k.a. *abuse cases* [12, 14].

Misuse cases describe interactions that cause harm to the system or its stakeholders and can be used as an informal front-end to more formal security requirements engineering. A closely related topic of research is that of *security use cases* [15]. Like misuse cases these describe interaction sequences where harm is attempted, but unlike misuse cases, the system ends up preventing or at least mitigating the damage.

In spite of the growing research interest in misuse cases, and promising early applications [10, 11, 13], the approach has yet to be put into large-scale industrial use. Many software development organizations tend to put little focus on security requirements, even if these are increasing in importance [16]. Partly, this may be due to a lacking understanding of security requirements. Indeed, even when they attempt to write security requirements, many developers tend instead to describe design solutions in terms of protection mechanisms, rather than making declarative statements about the degree of protection required [17]. Another reason for neglecting security requirements may be a perceived shortage of time in projects with narrow deadlines. For instance, case studies [18, 19] showed that security requirements were poorly addressed in several e-commerce projects.

To make misuse case analysis more appealing to practitioners *reuse* may be essential – as security requirements could then be specified more rapidly. As pointed out already in the 80's, reuse of requirements could lead to significant savings in development time and cost [20]. Although requirements reuse has attracted some research attention since then, methods suggested from academia have failed to demonstrate practicality or scalability [21] – perhaps with the exception of types of development particularly suited for reuse, such as product family development [22, 23] or ERP systems implementation [24].

The purpose of this paper is to provide a reuse-based methodology for misuse case analysis and the subsequent specification of security requirements. There are two key processes in reuse-oriented development [25, 26]:

- Development for reuse, where reusable artifacts are developed and made available for future

reuse, for instance in a repository / library that facilitates easy retrieval.

- Development with reuse, where end-user applications are developed, partly by reusing artifacts created by the "for" process.

There are interconnections between these two processes. Development with reuse can discover weaknesses of existing components in the reuse repository, or inspire new ideas for reusable components. Development for reuse can steer development with reuse if you are in a position to choose between alternative projects, picking the one where you have the greatest potential for reuse. The rest of this paper is structured as follows: Section 2 deals with development for reuse, discussing what kinds of artifacts should be developed, how the reusability of these artifacts should be ensured, and how the artifacts should be packaged in a repository for future reuse. Section 3 then addresses development with reuse, discussing how to identify candidates for reuse and then adapt them to the specific application. Section 4 discusses related work, and section 5 makes a concluding discussion.

2. Development *for* Reuse

Reuse of systems development artifacts may improve the quality of development processes and products and may reduce development costs if each artifact is reused at least 3–4 times (because it is more expensive to develop something reusable than something which will be used only once [27].) To ensure repeated reuse of security threats and requirements, we must find good answers to the following questions:

1. Which development artifacts should be stored in the repository for reuse?
2. How should the repository be organized to best support reuse?

2.1 The reusable development artifacts

As mentioned in the Introduction, applications are likely to face the same kinds of threats and have similar categories of required security even if they have different functional requirements. The challenge for reusability will be to describe threats and requirements on a sufficiently generic level, so that detailed differences between applications (e.g., in functionality, architecture) do not hamper the possibility for reuse. On the threats and requirements level, we suggest these types of reusable artifacts:

- *Generic threats*, described independently of particular application domains. Here we will only look at threats described as misuse cases, but other forms of representation could also be envisioned.
- *Generic security requirements*, again described independently of the particular application domain. These can be represented as security use cases or "system shall" requirements.
- *Application-specific threats* and *requirements*. Apart from including application-specific terminology, these can be described by misuse cases and security use cases (and/or "system shall" requirements) respectively, much similar to the generic varieties.

In addition to this, there could have been links further on to design level specifications, test cases etc., integrating reuse efforts across more phases, but this is not explored in our work so far.

For an example of a generic threat, Table 1 shows a generic misuse case that represents the threat of *spoofing*, i.e., a misuser gaining access to the system by pretending to be a legitimate user. This is a highly reusable misuse case, covering many different spoofing attacks. It does not matter if authentication is done by username+password, card+PIN, fingerprint scan, voice recognition, human to human recognition of individuals or something else. The interaction sequence is inspired by *essential misuse cases* [2], which focus on the users' intentions rather than concrete actions. In [2] the main motivation for this is to simplify the interaction and avoid premature design decisions, but avoidance of premature design will also increase the reusability of the description.

Table 1: A generic misuse case

Generic Misuse Case: Spoof User Access	
Summary: The misuser successfully makes the system (physical / human / computerized) believe he is a legitimate user, thus gaining access to a restricted system / service / resource / building.	
Preconditions: 1) The misuser has a legitimate user's valid means to identify and authenticate ***OR*** **2)** The misuser has invalid means to identify and authenticate, but so similar to valid means that the system is unable to distinguish (even if operating at its normal capabilities) ***OR*** **3)** The system is corrupted to accept means of identification and authentication that would normally have been rejected. The misuser may previously have performed misuse case "Tamper with system" to corrupt the system.	
Misuser interactions	**System interactions**
Request access / service	
	Request identification and authentication
Misidentify and misauthenticate	
	Grant access / provide service
Postconditions: 1) The misuser can do anything the legitimate user could have done within one access session ***AND*** 2) In the system's log (if any), it will appear that the system was accessed by the legitimate user.	

For an example of a generic requirement, Table 2 shows one path of the security use case "Access Control", more specifically the one requiring the system to reject misusers with valid means of identification but invalid means of authentication.

Table 2: One path of a generic security use case

Generic Security Use Case: Access Control		
Path name: Reject invalid authentication		
Preconditions: Misuser has valid means of user identification but invalid means of user authentication.		
Misuser Interactions	**System Requirements**	
	System Interactions	**System Actions**
	Request user identity and authentication.	
Provide valid user id but invalid authentication.		
	Reject misuser by cancelling transaction.	Attempt identification, authentication & authorization.
Postconditions: 1) Misuser has valid means of user identification but invalid means of user authentication ***AND*** 2) Misuser not authenticated, not granted access ***AND*** 3) Access control failure registered.		

Just like the generic misuse cases, this generic security use case is highly reusable – it makes no design assumptions, and neither does it presuppose any particular application domain. Access control is a feature wanted in a wide range of applications. The above use case could be a representative requirement for accessing an ATM, an internet entertainment service, or a missile control system.

This example may seem ridiculously simple – which it also is. But remember that this is just one of several paths of the security use case, and that the total response to the threat would encompass more than just one security use case. For the particular example threat "Spoof user access" the repository might contain:

- The security use case "Access Control", with several paths (more examples are shown in [15]).
- Other security use cases or normal use cases describing security related functions, e.g., "Cancel means of authentication")
- Requirements described by other means, e.g., "system shall" requirements or *mitigation points* in misuse cases [8].

Several alternative means of representation will be necessary here, as security use cases do have some limitations:

- They are easy to express when the threat is mitigated during the attempt (in the interaction path), not so easy for mitigations relating to the preconditions or postconditions of the threat.
- The are most easy to express for "absolute" requirements. But in many situations a 100% secure solution is impossible or infeasible. It can be noticed that it does not make sense to include a path in the Access Control use case for the situation where the misuser has valid means of identification *and* authentication, since then – to the system – the misuser *is* the legitimate user, and the system cannot be required to do anything else than it normally does, namely granting access.

Examples of other requirements that might be suggested as a response to the spoofing threat:

- "The means of authentication should have a stealability index value of [value]"[1].
- "Upon issue, the user shall sign a contract obliging him to keep the means of authentication safe from misuse, and to report potential compromise within [time limit]"
- A use case "Cancel Means of Authentication", to be applied when users report possible theft or loss of their means.
- "The [user] shall be limited to [maximum action] per [session / time period / …]"

The latter example, with 3 [ ] brackets, may seem so vague that one might wonder whether it has any utility in a reuse repository. But it does serve to remind the stakeholders of a certain mitigation option that could otherwise easily be forgotten. An example of an instantiation will be shown in the next section.

2.2 The organisation of the repository

A meta-model showing Threats and Security Requirements and the links between them is given in Figure **1**. Threats are what misusers try to achieve, causing harm to the system, and security requirements describe the extent to which the system shall be able to mitigate those threats. Key to understanding the diagram are the two classes on the way from Threat to Security Requirement, namely Threat-Requirement Relationship and Security Requirement Bundle. To start with the latter, this is a set of requirements that pull together in mitigating the same threat. It is often interesting to look at

[1] This assumes the definition of a (yet non-existing) stealability index for means of authentication, similar to what exists for cars. Clearly, less precise statements like "The means of authentication shall be difficult to steal" are not useful as requirements.

such bundles rather than just individual requirements, because:

- A security requirement bundle is a bigger and more effective unit of reuse. To the extent that one requirement is a good unit of reuse, it is still possible to define a bundle consisting only of that requirement.
- In many cases, single security requirements provide little or no protection unless accompanied by other requirements. For instance, as observed in [17] identification requirements are seldom of much value alone – in most cases they must be accompanied by authentication requirements.

Indeed, it can be observed that a security use case is in itself a requirement bundle. The example in Table 2 already contains two requirements – a) that the system shall reject access to users without valid means of authentication, and b) that failed access attempts shall be registered. And this is only one path of the bigger use case, so in total it would encompass several requirements.

The Threat-Security link objects will most commonly represent "mitigate" relationships, i.e., that a certain requirement bundle (e.g., the security use case "Access Control") mitigates a threat (e.g., the misuse case "Spoof User Access"). Another possible relationship is "aggravate", i.e., the choice of a requirements bundle may actually increase the risk for a threat. An example is that a bundle of Access Control requirements might increase the risk for Denial of Service (DoS) threats. A classical example is the suspension of console login for a certain user after three failed login attempts – a misuser could then deny access for that user simply by making those three failed login attempts with that username. In general, any requirement that the system should suspend access if sensing an attempted attack might be utilized for DoS purposes.

The Requirement-Requirement Relationship is used to register relationships between requirements, e.g., that they may be overlapping or in conflict. The aggregation from Threat to Threat Specification enables one threat to have several parallel representations in terms of format or language. For instance, the same misuse case could be written both in English, French and Norwegian, or in the same language but with different templates, or there could also be other representations than misuse cases, for instance in more formal languages (not investigated in this paper). The upper right part of the diagram shows an analogous modeling of the requirements side.

The lower left part shows that a Threat can either be a Generic Threat or an Application-Specific Threat, and one Generic Threat may have many Application-Specific instantiations. For instance, the "Spoof User Access" threat of Table 1 may be instantiated to cover illegitimate access to an ATM, a building, or an internet entertainment service. The lower right part of the figure shows that the requirements side is structured accordingly.

Basing a reuse repository on the above meta-model, the following two advantages are achieved:

- Security requirements may be searchable *via threats* that they are meant to mitigate, rather than having to search for requirements "directly". The "direct" alternative is less useful here – to know what to look for the developer must have a pretty clear picture of the requirement already, which reduces the gain from reuse.
- Security requirements can be packaged in bundles that give meaningful protection against commonly seen threats. In most cases this should

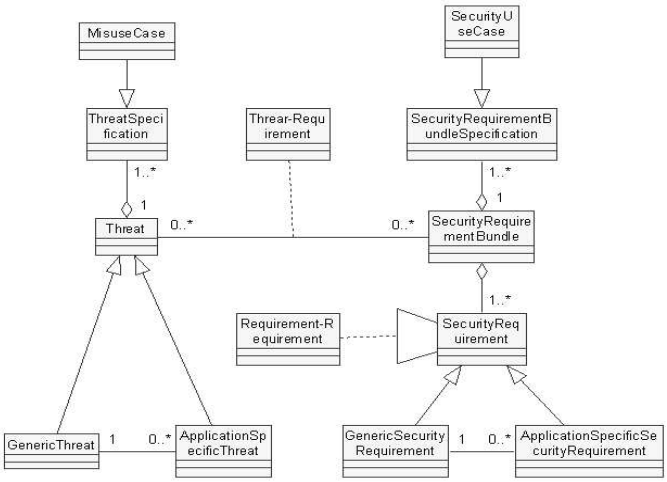

Figure 1: Meta-model for repository

be more effective than reusing requirements one by one and then assembling them in meaningful bundles on a project-to-project basis.

Having observed these advantages we now turn to development with reuse.

3. Development *with* reuse

Figure 2 shoes a UML Activity Diagram that outlines our suggested approach to development with reuse. The steps are as follows:

1. ***Identify critical and/or vulnerable assets:*** Here one must identify all the critical and/or *vulnerable assets* in the enterprise. A vulnerable asset is either information or materials that the enterprise possesses, locations that the enterprise controls or activites that the enterprise performs.[2] The "and/or" should be noticed specifically. It is interesting to look at assets that are critical but not vulnerable because a) further scrutiny may reveal that they were only believed not to be vulnerable, and b) their vulnerability might increase in the future. It is also interesting to look at assets that are vulnerable but not critical, at least if they are of the kind that misusers may use as stepping stones to launch attacks on more critical resources. For example, a server that holds no critical information and runs no critical services might still be used as a zombie in an attack against other computing resources, perhaps also in other companies, causing badwill or even liability to the organization. Starting the security analysis with a focus on assets ensures that the final security requirements are anchored in the protection of materials, information, locations and activities that are of value to the enterprise.
2. ***Determine security goals for each asset:*** For each critical and/or vulnerable asset identified in step 1., select the appropriate security goals for the asset. A security goal is specified in terms of (1) *who* are the potential misusers, (2) the *type* of security breaches the asset is vulnerable to and (3) the security *level* necessary for that type of breach. For example, the potential misusers may be Internet script kiddies, business competitors or disgruntled employees. Examples of security types are violations of, e.g., secrecy or integrity, and several of the security threat classifications in the literature can be used in this step. A possible taxonomy of security breaches is proposed in [17]. The security level to be achieved is specified as a probability that the assets will be kept safe from the particular type of breach from the particular type of misuser. Establishing security goals for all the critical and/or vulnerable assets ensures that the eventual security requirements are derived based on thoroughly identified types of misuers and of security breaches. Also, well-defined security goals are a prerequisite for identifying threats. (If you have no goals, there are no threats either. Even "being killed" is only a threat if you consider "staying alive" as a goal and "life" as an asset.)
3. ***Identify threats to each asset:*** For each security goal identified in step 2, find all the threats that can prevent the goal from being achieved or maintained. This is where the repository is used for the first time. First, find misuse cases in the repository that involve the right types of misusers as specified by the goal and, then, select those misuse cases that threaten the right type of security breach. Finally, assess whether the misuse case poses a threat that is relevant given the security level specified by the goal. For example, misuse cases that involve the breaking of cryptographic codes may be a relevant threat to the security and integrity of banks or military installations with extremely high security levels, but not to the security and integrity of student information in a university information system. In addition to using the repository, it is of course necessary to look for threats that are not directly implied by the determined security goals, because some security goals may indeed have been forgotten.
4. ***Analyze risk for each threat:*** In its most detailed form, the specification of threats must include the risk of the various threats, i.e., the estimated likelihood of occurrence and cost of the damage if the threat occurs. Whereas the description of threats is highly reusable, risks must normally be determined from application to application. For example, although both an Internet entertainment service and a missile control system face the

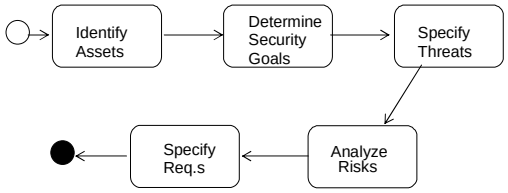

Figure 2: Development for reuse, process

[2] The most important assets of enterprises, the knowledge and skills of its workers, is not directly important in an ICT security context, as they are only vulnerable indirectly, through misuse of the other, more tangible assets.

threat of spoofing, the associated risks may be quite different.

5. ***Determine requirements:*** For each identified threat, and taking its risk into account, determine requirements to mitigate the threat. The repository is used again here. For each threat retrieved from the repository, one or more associated bundles of security requirements may be found. For threats not retrieved from the repository, appropriate security requirements must be determined and specified by other means. Even when threats are retrieved from the repository, additional bundles of security requirements that mitigate the threat may be found by other means. Different levels of mitigation will be needed for different threats, and requirements workers must select requirements bundles that together produce the necessary levels of mitigation for all threats.

When the process is completed, there should be satisfactory requirements specified for all threats, and threats should have been investigated for the security goals of all assets.

In this paper we do not discuss the first two steps any further (although one might envision some reuse even in those steps, for instance by means of asset checklists), neither do we discuss step 4. It is however necessary to show these steps to illustrate the context in which the reuse of step 3 and 5 takes place. The activity diagram of Figure 3 shows the decomposition of the threat specification (step 3). Three possible ways are suggested to identify threats:

- Top down Threat search means that you start from the identified assets and security goals and then try to search the repository for threats relevant to such assets / security goals. This would be best supported if there were attributes pointing to relevant types of assets or security goals in the Threat class, or alternatively there could be separate classes for asset types and goal types which the threats were then associated with.
- Bottom-up threat search, i.e., starting by looking at what you have in repository (without regard for the determined security goals) and then considering whether different threats described there are relevant to your application. This might seem a less systematic approach than the top-down alternative, and if the repository is big it might cause a lot of wasted time looking at irrelevant threats. However, it might be a valuable corrective to a strict top-down development in that it gives an extra check that no threats have been overlooked. As security goals may have been overlooked in the previous stage (or assets before that), a strict top-down approach gives no guarantee that all threats will be discovered.
- Threat brainstorming. This is the option for threats which cannot be found in the repository (whether mandated from determined security goals or not). But of course, whenever a threat has been suggested by brainstorming, one should check to make sure it is indeed not covered by the repository.

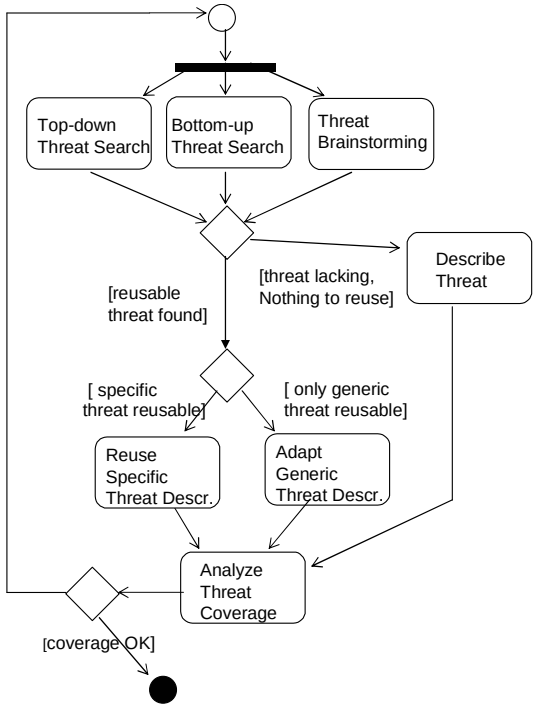

Figure 3: Decomposition of "Specify Threats"

Whatever method a threat has been identified by, one of two situations may occur:

- The repository contains no description that can be reused for this threat. In rare cases this could happen even for threats discovered through the bottom-up approach, i.e., browsing through the repository the developers come upon a threat that is indeed relevant to their application, but the description in the repository is not reusable enough.
- The repository contains a description that can be reused for this threat. In this case there are two new alternatives: Either there is only a generic threat description that can be reused, this must then be adapted to an application specific instantiation. Or there is already a fitting application-specific variety in the repository, then

this can possibly be reused as-is, saving even more work for the developers.

As an example, imagine that the repository contained the threat "Spoof User Access" of Table 1, and that this was retrieved and found relevant in the project at hand – to develop a new ATM system. Then, the generic misuse case could be adapted to the application specific misuse case shown in Table 3 – the only phrases that would have to be rewritten would be the underlined ones. However, if there had also been a previous development project for an ATM system by means of the repository, it might well be that there already was such an application-specific misuse case. Then this could be reused directly.

Table 3: Application-specific misuse case

Misuse Case Name: Spoof Customer at ATM	
Summary: The misuser successfully makes the ATM believe he is a legitimate user. The misuser is thus granted access to the ATM's customer services.	
Preconditions: 1) The misuser has a legitimate user's valid means of identification and authentication ***OR*** 2) The misuser has invalid means of identification and authentication, but so similar to valid means that the ATM is unable to distinguish ***OR*** 3) The ATM system is corrupted, accepting means of identification and authentication that would normally have been rejected.	
Misuser interactions	**System interactions**
Request access	
	Request identification and authentication
Misidentify and misauthenticate	
	Grant access
Postconditions: 1) The misuser can use all the customer services available to the spoofed legitimate user ***AND*** 2) In the system's log (if any), it will appear that the ATM was accessed by the legitimate user.	

Moving on to step 5, the decomposed activity diagram for this can be found in Figure 4. When it comes to requirements, the chance for reuse should be considerable if a threat was reused – then one can follow the repository's links to one or more requirement bundles for that threat. If the threat had to be specified from scratch, there are no directly corresponding requirements in the repository, so the chance for reuse is much smaller. Yet, it could pay off at least to browse briefly for requirements related to similar threats, if any.

Either way, it may happen that no requirement bundles are found satisfactory for reuse, or there may be potential for reuse. Here the situation is quite similar to the reuse of threats: It might be that reuse is only possible with adaptation from the generic level, but one might also be lucky enough to be able to reuse something from the specific level, as is. If we take ATM systems as a concrete example, the Access Control path shown in Table 2 can be utilized almost directly at the specific level, possibly

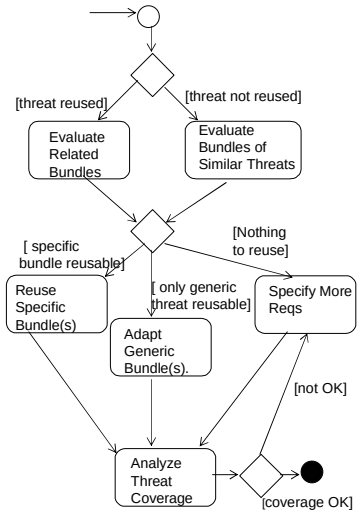

Figure 4: Decomposition of "Specify Req.s"

only with a slight name change to "ATM Access Control". As there is no point in showing the same example twice, we instead show an authorization example. An instantiation of the generic requirement *"The [user] shall be limited to [maximum action] per [session / time period / …]"* could be "The ATM customer shall be limited to withdrawing maximum 1000 USD of cash per week" – not preventing spoofing but at least reducing the damage for the cases when it does occur. It would also be possible to express this as a path of a security use case, and although the "shall" requirement is probably simpler and better to use in this case, we show it for illustration in Table 4.

Table 4: A specific security use case

Security Use Case: ATM Authorization	
Path name: Reject withdrawal beyond weekly limit	
Preconditions: 1) Misuser has gained access to ATM customer services, e.g., by a successful "Spoof User Access". 2) The account has a weekly cash withdrawal limit of USD 1000, of which Y < 1000 has currently been withdrawn. 3) Account balance B > 1000 - Y	
Misuser Interactions	**System Interactions**
Request to withdraw Z1 > X - Y	
	Deny withdrawal as exceeding weekly limit
Request to withdraw Z2 <= X - Y	
	Accept withdrawal, dispense cash
Postconditions: 1) The misuser will max. have been able to withdraw X. 2) New Account Balance B is old B – Z2	

When threats have been analyzed, determining the level of security needed towards various threats, follow the repository links from the threats side to the requirements side, to look at alternative requirements to mitigate the relevant threats – and choose those most appropriate to the needed security levels.

The chosen generic security requirements should then be adapted to application specific ones. In some cases hardly any rewriting is needed, in other examples it may be necessary to change some terms to application specific ones, and to quantify requirements where the generic ones only indicate the possibility to quantify, e.g., changing <time limit> with an actual time limit or X% with a number.

4. Related Work

Reuse of requirements has been investigated by researchers for quite a time, e.g., reusing fragments of domain knowledge through inheritance [28], reuse by analogy of structure [29], semantic matching [30], or by clustering of specification diagrams [31]. Our approach differs from these in its specific focus on security requirements (contrary to, e.g., functional requirements), and the investigation of one particular form of representation as a vehicle for reuse (misuse cases).

Reuse of use cases or scenarios has been investigated by, e.g., [32, 33], through use case patterns and retrieval based on some concept of similarity. Our suggested approach also differs from these in its particular focus on security requirements. Also, the ideas concerning retrieval seem to be different. With use cases / scenarios (typically expressing functional requirements), the idea seems to be that the developer knows to some extent what he is looking for (e.g., being able to partially describe a use case), and then the system will suggest something similar). Our idea with reuse based on misuse cases, however, is that the reuser might not know what he is looking for. Indeed the highest benefits of reuse might be in cases where the developer, browsing the library, becomes aware of a threat to the system that he had no idea of beforehand (and would thus have overlooked). Hence, one can envision the repository structure being used more in a checklist manner, looking at all the various threat categories and considering whether they are relevant for the application to be developed, rather than specifying something vaguely and then searching for it. This means that the reuser's interaction with the repository will be more dominated by taxonomy-supported browsing than by massive automated searches.

A work that deals with reuse of security requirements – therefore being particularly close in topic to ours – is the SIREN approach by Toval et al. [34]. This approach suggests a repository of security requirements initially populated by using MAGERIT, the Spanish public administration risk analysis and management method conforming to ISO15408 (the Common Criteria Framework [35]). Here, it suggests a process with the following 4 steps: i) identify assets, ii) identify vulnerabilities (threats to assets), iii) analyze risks, iv) choose countermeasures. This is quite similar to our process for development with reuse of Figure 2: steps (i) are identical, our step (iii-iv) are similar to their (ii-iii). There are two differences, though:

- We suggest a step 2 of identifying security *goals* for each asset before going on to threats. Our argument for this is that the definition of security goals should precede threats.
- Our step 5 is the identification of *requirements*, while their parallel step 4 talks about *countermeasures*. As argued in [17], the premature specification of design in terms of countermeasures is unlucky – one should first try to express pure requirements (e.g., what level of protection is needed, rather than how to achieve that protection in terms of architectural mechanisms). The examples in [34] do indeed indicate some design tendencies in the suggested requirements (e.g., passwords, firewalls).

In addition to the 4 steps from MAGERIT, SIREN also suggests a larger scale process, based on a Spiral model (but concentrating on requirements engineering, not the entire development). Yet this process is much wider than what is addressed in our paper, the SIREN process deals with all tasks concerning the requirements specification – selection from the repository, elicitation, negotiation, specification, validation. We look more narrowly and in more detail only at the activities directly related to reuse.

The suggested method addresses many things not addressed in our work, e.g., organizing assets in 5 levels, and defining a requirements document hierarchy with 5 different documents (different kinds of requirements specifications and test plans). These are not contradictory with our approach, suggesting that they could supplement each other. When it comes to the SIREN repository structure, requirements can be structured according to *domains* and *profiles* – the former reflecting functional application areas, the latter opening for a possible structuring according to non-functional aspects (e.g., a profile for information systems security). Requirements can be parameterized or non-parameterized. The latter can be reused directly, whereas the former must have, e.g., some values filled in. This does not exactly parallel our difference between generic and application-specific – rather, both parameterized and non-parameterized requirements are on the same level in that respect.

Moreover, SIREN focuses on requirement lists, while we focus on (mis)use cases, but this is clearly a surface difference, as there is clearly nothing in the SIREN approach that excludes the inclusion of use cases in the repository (and neither would requirement lists be excluded from our repository). The requirements in the SIREN repository can be coupled to (or retrieved via) the aforementioned document structure and the MAGERIT asset hierarchy. This is different from our approach, where a) requirements are navigated from threats, not assets, and b) the threat specifications are seen as the principal artifacts of reuse, possibly together with corresponding requirements.

A distinguishing property of our suggestion is thus that we suggest to reuse specifications of both requirements and threats, which are closely linked. Apart from the fact that goals and threats are opposites, their relationship to requirements are quite the same (e.g., that there can be many different choices of requirements to address the same goal or threat). Hence, our approach is also related to goal-oriented approaches. For instance, the *obstacles* discussed in [36] (in connection with the KAOS method) could be likened to our threats, and [19] (in connection with the GBRAM approach) discusses the linking of security/privacy policies and requirements. Also relevant is the work on trust in i* [37], by which threats can also be modeled. Rather than contradicting these, our particular work again looks more narrowly at reuse issues with one particular form of representation (misuse cases).

5. Discussion and Conclusions

The paper has proposed a reuse-based approach to determining security requirements. The main weakness is that our suggested reuse approach has not been tried out in practice – for which a tool would have to be developed the repository populated with threats and requirements to be reused. Yet we contend that the contribution in this paper at least is a good starting point for such demonstrations of practicality, with its suggested models for the repository and reuse process. Development *for* reuse involved identifying security threats and associated security requirements. This can either be done as domain analysis or during application development, and should yield a repository of threats and related requirements. Threats can for instance be expressed as misuse cases and requirements as security use cases. Development *with* reuse involved identifying security assets, setting security goals for each asset, identifying threats to each goal, analyzing risks and determining security requirements, based on reuse of generic threats and requirements from the repository. Advantages of the proposed approach include building and managing security knowledge through the shared repository, assuring the quality of security work by reuse, avoiding over-specification and premature design decisions by reuse at the generic level and focusing on security early in the requirements stage of development. The proposed approach may also save time in the early development phases and produce more complete requirements, as the repository may prevent developers from forgetting important threats or requirements. The generic security requirements show the developers what level their description should be at whereas, otherwise, it would be tempting to jump directly from threats to design mechanisms (or even to mechanisms directly, without completely understanding the threats).

The main difference from related work is a specific focus on the reuse of threats *and* security requirements, both described in terms of use cases. On the other hand, this paper fails to address many issues that are addressed by related work, hence integration with other approaches is an interesting topic for further work.

Work on reuse-based determination of security requirements is still in its early stages, and industrial case studies are called for. To better support development *for* reuse, further work is needed on how to link misuse cases in the repository to relevant security goals, to better prepare for development *with* reuse. The repository should be implemented in a tool and integrated with CASE tools. For example, the tool should support abstraction of application specific threats and security requirements into generic ones. The tool should also enforce a common taxonomy and terminology, e.g., for types of misusers and security breaches, in order to increase search efficiency.

To better support development *with* reuse, further work is needed on method guidance for specifying security goals, in particular on how to best classify security threats. Heuristics for setting security levels would also be helpful. Of course, the tool should support searching for threats according to misuser and type of security breach, both exactly and approximately.

Comparing the present proposal to goal- and agent-oriented approaches to security requirements work is another path for further work. As emphasized also in previous publications, misuse case analysis has never intended to be a full-fledged development approach in its own right, rather the idea is that it must be integrated with other approaches.

References

[1] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Boston: Addison-Wesley, 1992.

[2] L. L. Constantine and L. A. D. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*: ACM Press, 1999.
[3] A. Cockburn, *Writing Effective Use Cases*. Boston: Addison-Wesley, 2001.
[4] J. Rumbaugh, "Getting Started: Using use cases to capture requirements," *Journal of Object-Oriented Programming*, pp. 8-23, 1994.
[5] D. Kulak and E. Guiney, *Use Cases: Requirements in Context*: ACM Press, 2000.
[6] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenario Usage in System Development: A Report on Current Practice," *IEEE Software*, vol. 15, pp. 34-45, 1998.
[7] G. Sindre and A. L. Opdahl, "Eliciting Security Requirements by Misuse Cases," presented at TOOLS Pacific 2000, Sydney, 2000.
[8] G. Sindre and A. L. Opdahl, "Templates for Misuse Cases," presented at REFSQ'2001, Interlaken, 2001.
[9] G. Sindre, A. L. Opdahl, and G. F. Breivik, "Generalization/Specialization as a Structuring Mechanism for Misuse Cases," presented at 2nd Symposium on Requirements Engineering for Information Security, Raleigh, NC, 2002.
[10] I. F. Alexander, "Initial Industrial Experience of Misuse Cases in Trade-Off Analysis," presented at RE'02, Essen, 2002.
[11] I. F. Alexander, "Misuse Cases, Use Cases with Hostile Intent," *IEEE Software*, vol. 20, pp. 58-66, 2003.
[12] J. McDermott, "Abuse-Case-Based Assurance Arguments," presented at 17th Annual Computer Security Applications Conference (ACSAC'01), 2001.
[13] I. F. Alexander, "Modelling the Interplay of Conflicting Goals with Use and Misuse Cases," presented at 8th International Workshop on Requirements Engineering: Foundation for Software Quality, Essen, Germany, 2002.
[14] J. McDermott and C. Fox, "Using Abuse Case Models for Security Requirements Analysis," presented at 15th Annual Computer Security Applications Conference (ACSAC'99), 1999.
[15] D. Firesmith, "Security Use Cases," *Journal of Object Technology*, vol. 2, pp. 53-64, 2003.
[16] R. Crook, D. Ince, L. Lin, and B. Nuseibeh, "Security Requirements Engineering: When Anti-Requirements Hit the Fan," presented at IEEE International Requirements Engineering Conference (RE'02), Essen, Germany, 2002.
[17] D. Firesmith, "Engineering Security Requirements," *Journal of Object Technology*, vol. 2, pp. 53-68, 2003.
[18] A. I. Anton, R. A. Carter, A. Dagnino, J. H. Dempster, and D. F. Siege, "Deriving Goals from a Use Case Based Requirements Specification," *Requirements Engineering Journal*, vol. 6, pp. 63-73, 2001.
[19] A. I. Anton and J. B. Earp, "Strategies for Developing Policies and Requirements for Secure Electronic Commerce Systems," presented at 1st ACM Workshop on Security and Privacy in E-Commerce, 2000.
[20] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment and Directions," *IEEE Software*, vol. 4, pp. 41-49, 1987.
[21] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective," presented at ICSE'2000, Limerick, Ireland, 2000.
[22] M. Mannion, B. Keepence, H. Kaindl, and J. Wheadon, "Reusing Single System Requirements from Application Family Requirements," presented at ICSE'99, Los Angeles, CA, 1999.
[23] R. R. Lutz, "Towards Safe Reuse of Product Family Specifications," presented at SSR'99, Los Angeles, CA, 1999.
[24] M. Daneva, "Measuring Reuse of SAP Requirements: a Model-based Approach," presented at SSR'99, Los Angeles, CA, 1999.
[25] E.-A. Karlsson, "Software Reuse: A Holistic Approach," in *Wiley Series in Software Based Systems*: John Wiley & Sons, 1995.
[26] G. Sindre, R. Conradi, and E.-A. Karlsson, "The REBOOT Approach to Software Reuse," *Journal of Systems and Software*, vol. 30, pp. 201-212, 1995.
[27] W. Tracz, "Software Reuse Myths," *ACM SIGSOFT Software Engineering Notes*, vol. 13, pp. 17-21, 1988.
[28] H. Reubenstein and R. Waters, "The Requirements Apprentice: Automated assistance for requirements acquisition," *IEEE Software*, vol. 17, pp. 226-240, 1991.
[29] N. A. M. Maiden and A. G. Sutcliffe, "Exploiting Reusable Specifications through Analogy," *Communications of the ACM*, vol. 35, pp. 55-64, 1992.
[30] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks," presented at 3rd International Conference on Requirements Engineering, Washington DC, 1997.
[31] O. Lopez, M. A. Laguna, and F. J. Garcia, "Reuse-based Analysis and Clustering of Requirements Diagrams," presented at 8th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'02), Essen, Germany, 2002.
[32] W. Lam, "Scenario reuse: A technique for complementing scenario-based requirements engineering approaches," presented at 4th Asia Pacific Software Engineering and International Computer Science Conference (APSEC'97 / ICSC'97), Hong Kong, 1997.
[33] H. G. Woo and W. N. Robinson, "Reuse of Scenario Specifications Using an Automated Relational Learner: A Lightweight Approach," presented at IEEE Joint Conference on Requirements Engineering (RE'02), Essen, Germany, 2002.
[34] A. Toval, J. Nicolas, B. Moros, and F. Garcia, "Requirements Reuse for Improving Information Systems Security: A Practitioner's Approach," *Requirements Engineering Journal*, vol. 6, pp. 205-219, 2002.
[35] CCIMB, "Common Criteria for Information Technology Security Evaluation," Common Criteria Implementation Board, Technical Report CCIMB-99-031, August 1999.
[36] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering," *IEEE Transactions on Software Engineering*, vol. 26, pp. 978-1005, 2000.
[37] E. Yu and L. Liu, "Modelling Trust in the i* Strategic Actors Framework," presented at 3rd Workshop on Deception, Fraud and Trust in Agent Societies, Barcelona, 2000.