ABSTRACT

Background: Software developers are facing increased pressure to lower development time, release new software versions more frequent to customers and to adapt to a faster market. This new environment forces developers and companies to move from a plan based waterfall development process to a flexible agile process. By minimizing the pre-development planning and instead increasing the communication between customers and developers, the agile process tries to create a new, more flexible way of working. This new way of working allows developers to focus their efforts on the features that customers want. With increased connectability and the faster feature release, the security of the software product is stressed. To develop secure software, many companies use security engineering processes that are plan heavy and inflexible. These two approaches are each others opposites and they directly contradict each other.

Objective: The objective of the thesis is to evaluate how to develop secure software in an agile process. In particular, what existing best practices can be incorporated into an agile project and still provide the same benefit if the project was using a waterfall process. How the best practices can be incorporated and adapted to fit the process while still measuring the improvement. Some security engineering concepts are useful but the best practice is not agile compatible and would require extensive adaptation to integrate with an agile project.

Method: The primary research method used throughout the thesis is case studies conducted in a real industry setting. As secondary methods for data collection a variety of approaches have been used, such as semi-structured interviews, workshops, study of literature, and use of historical data from the industry.

Results: The security engineering best practices were investigated through a series of case studies. The base agile and security engineering compatibility was assessed in literature, by developers and in practical studies. The security engineering best practices were group based on their purpose and their compatibility with the agile process. One well known and popular best practice, automated static code analysis, was toughly investigated for its usefulness, deployment and risks of using as part of the process. For the risk analysis practices, a novel approach was introduced and improved. As such, a way of adapting existing practices to agile is proposed.

Conclusion: With regard of agile and security engineering we did not find that any of the investigated processes was agile compatible. Agile is reaction driven that adapts to change, while the security engineering processes are proactive and try to prevent threats before they happen. To develop secure software in an agile process the developers should adopt and adapt key concepts from security engineering. These changes will affect the flexibility of the agile process but it is a necessity if developers want the same software security state as security engineering processes can provide.
Developing Secure Software
- in an Agile Process

Dejan Baca
Developing Secure Software
- in an Agile Process

Dejan Baca

Doctoral Dissertation in
Computer Science

Blekinge Institute of Technology doctoral dissertation series
No 2012:05

School of Computing
Blekinge Institute of Technology
SWEDEN
"Do not worry, it is non-destructive security testing. Your week long stress test will not be effected by it."

My last words before getting banned from the test lab.

–Dejan Baca
Abstract

Background: Software developers are facing increased pressure to lower development time, release new software versions more frequent to customers and to adapt to a faster market. This new environment forces developers and companies to move from a plan based waterfall development process to a flexible agile process. By minimizing the pre-development planning and instead increasing the communication between customers and developers, the agile process tries to create a new, more flexible way of working. This new way of working allows developers to focus their efforts on the features that customers want. With increased connectability and the faster feature release, the security of the software product is stressed. To develop secure software, many companies use security engineering processes that are plan heavy and inflexible. These two approaches are each others opposites and they directly contradict each other.

Objective: The objective of the thesis is to evaluate how to develop secure software in an agile process. In particular, what existing best practices can be incorporated into an agile project and still provide the same benefit if the project was using a waterfall process. How the best practices can be incorporated and adapted to fit the process while still measuring the improvement. Some security engineering concepts are useful but the best practice is not agile compatible and would require extensive adaptation to integrate with an agile project.

Method: The primary research method used throughout the thesis is case studies conducted in a real industry setting. As secondary methods for data collection a variety of approaches have been used, such as semi-structured interviews, workshops, study of literature, and use of historical data from the industry.

Results: The security engineering best practices were investigated through a series of case studies. The base agile and security engineering compatibility was assessed in literature, by developers and in practical studies. The security engineering best practices were group based on their purpose and their compatibility with the agile process. One well known and popular best practice, automated static code analysis, was toughly investigated for its usefulness, deployment and risks of using as part of the process. For the risk analysis practices, a novel approach was introduced and improved. As such, a way of adapting existing practices to agile is proposed.

Conclusion: With regard of agile and security engineering we did not find that any of the investigated processes was agile compatible. Agile is reaction driven that adapts to change, while the security engineering processes are proactive and try to prevent threats before they happen. To develop secure software in an agile process the developers should adopt and adapt key concepts from security engineering. These changes will affect the flexibility of the agile process but it is a necessity if developers want the same software security state as security engineering processes can provide.
Acknowledgements

First and foremost, I would like to thank my supervisors PhD Lars-Ola Damm, Professor Bengt Carlsson and Professor Lars Lundberg for their support, especially for valuable feedback on papers and other research related advice. Especially Bengt Carlsson for giving me a swift kick when I needed to write papers instead of starting new experiments.

Ericsson AB gave me the chance of conducting research and at the same time staying in touch with the industry. The industrial Ph.D student position gave me a unique opportunity to identify problems and evaluate them in a practical real world environment instead of a test lab. A special thanks to my unit managers, whose budget I was constancy breaking; Maria Larsson, PerOlof Bengtsson, Sven Johansson and Daniel Borg.

Blekinge Institute of Technology provided me with colleagues and friends from all over the world that create a diverse environment of ideas and culture. I would especially like to thank Kai Petersen whom I wrote many of my papers with. Martin Hylerstedt who always agreed to read my papers and help me with proofreading. Anton Borg, Petar Jersick, Martin Boldt, Samireh Jalali and Svetlana Zivanovic for being my friends and colleagues at BTH during my time there.

Finally, I would like to thank my family and friends for putting up with me despite neglecting them when having a high workload.

This work was funded jointly by Ericsson AB and the Knowledge Foundation in Sweden under a research grant for the research school SAVE-IT from Mälardalens university.
Overview of Papers

Papers that are included in this thesis. Each paper answers part of a research question and the papers are included as appendix in the thesis.


For each of the included paper I was the main author and responsible for conducting the study and writing the major part of the paper, while the co-authors aided in research design, general feedback and in finalizing the papers.
Papers that are related to but not included in this thesis.


# Table of Contents

1. **Introduction** ...................................................... 1

2. **Concepts** .......................................................... 3
   2.1 Software Development ........................................ 3
   2.1.1 Waterfall Processes ....................................... 4
   2.1.2 Security Engineering ...................................... 6
   2.1.3 Agile Process .................................................. 7
   2.2 Security .......................................................... 8
   2.2.1 Software Vulnerabilities ................................... 8
   2.2.2 Early Vulnerability Detection ............................ 10
   2.2.3 Design Vulnerabilities ..................................... 11
   2.2.4 Implementation Vulnerabilities .......................... 11
   2.2.5 Configuration Vulnerabilities ............................ 11
   2.3 Security Engineering Best Practices ...................... 12
   2.3.1 Common Activities ......................................... 12
   2.3.2 Risk Analysis ................................................ 13
   2.3.3 Automated Static Code Analysis ........................ 13

3. **Research Approach** .............................................. 17
   3.1 Research Questions ........................................... 17
   3.2 Research Methods .............................................. 20
   3.3 Research Validity ............................................... 21
   3.4 Research Environment ......................................... 23
   3.4.1 Waterfall Process .......................................... 23
   3.4.2 Agile Processes ............................................. 24
   3.4.3 Products ....................................................... 26
   3.5 Vulnerability Taxonomy ....................................... 26
4 Outline and Results

4.1 Prioritizing Countermeasures through the Countermeasure Method for Software Security (CM-Sec) .................................................. 29
4.2 Countermeasure Graphs for Security Risk Assessment: An Action Research ................................................................. 30
4.3 Static Code Analysis to Detect Software Security Vulnerabilities: Does Experience Matter? .................................................. 31
4.4 Improving Software Security with Static Automated Code Analysis in an Industry Setting .................................................. 32
4.5 Agile development with security engineering activities .................................................. 34
4.6 Integrating security engineering in an Agile process .................................................. 35

5 Discussion and Conclusions .................................................. 37

A Prioritizing Countermeasures through the Countermeasure Method for Software Security (CM-Sec) .................................................. 41
A.1 Introduction .................................................. 41
A.2 Background and Related Work .................................................. 43
A.3 Countermeasure Method for Software Security (CM-Sec) .................................................. 44
   A.3.1 Countermeasure Graphs: An Extension to Attack Trees .................................................. 44
   A.3.2 Process .................................................. 46
A.4 Application .................................................. 49
   A.4.1 System Description and Development Environment .................................................. 49
   A.4.2 Result of Applying ACM-Sec .................................................. 50
A.5 Discussion .................................................. 55
   A.5.1 Practical Implications: .................................................. 55
   A.5.2 Research Implications: .................................................. 55
A.6 Conclusion .................................................. 56

B Countermeasure Graphs for Security Risk Assessment: An Action Research .................................................. 57
B.1 Introduction .................................................. 57
B.2 Related Work .................................................. 58
B.3 Countermeasure Graph .................................................. 60
   B.3.1 Process .................................................. 62
B.4 Evaluation .................................................. 65
   B.4.1 Research Conduct .................................................. 66
   B.4.2 Results .................................................. 69
B.5 Countermeasure Graph .................................................. 75
### C Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?  
C.1 Introduction .......................................................... 81  
C.2 Background and Related Work ................................. 83  
C.3 Research Method .................................................... 83  
C.3.1 Variables .......................................................... 83  
C.3.2 Subjects ........................................................... 84  
C.3.3 Instrumentation .................................................. 85  
C.3.4 Operation .......................................................... 86  
C.3.5 Threats to Validity ................................................. 86  
C.3.6 Static analysis tool output examples ....................... 87  
C.4 Data Analysis .......................................................... 88  
C.5 Discussion ............................................................. 91  
C.6 Conclusion ............................................................. 93  

### D Improving Software Security with Static Automated Code Analysis in an Industry Setting  
D.1 Introduction .......................................................... 95  
D.2 Related Work .......................................................... 96  
D.3 Research Method .................................................... 98  
D.3.1 Units of Analysis and Case Study Context ............... 98  
D.3.2 Research Goal .................................................... 100  
D.4 Data Collection Methods ............................................ 101  
D.4.1 Tool capabilities ................................................... 101  
D.4.2 Deployment .......................................................... 104  
D.4.3 Usage by Developers .............................................. 104  
D.5 Results ................................................................. 106  
D.5.1 Tool capabilities ................................................... 106  
D.5.2 Deployment .......................................................... 112  
D.5.3 Usage ............................................................... 113  
D.6 Discussion ............................................................. 117  
D.6.1 Tool Capabilities ................................................... 117  
D.6.2 Deployment .......................................................... 118  
D.6.3 Usage ............................................................... 119  
D.6.4 Validity threats and study limitations ..................... 120  
D.7 Conclusions ............................................................ 121
Table of Contents

F.4 Research Method ........................................... 157
  F.4.1 Case Study Context .................................... 157
  F.4.2 Research Context ....................................... 157
  F.4.3 Case Study ............................................... 159
  F.4.4 Threats to validity ..................................... 161
F.5 Evaluating waterfall activities ................................. 161
F.6 Integrating with agile ....................................... 166
F.7 Security activities .......................................... 170
  F.7.1 Excluded activities ..................................... 171
F.8 Discussion ................................................ 173
F.9 Conclusions ............................................... 175
F.10 Future works ............................................ 176

List of Figures .................................................. 185

List of Tables .................................................. 187
Chapter 1

Introduction

Most software development organizations have some desire or goal to create secure software that insure their products’ availability and robustness. How software security is achieved varies greatly in terms of commitment, approach and actual result. Common for all are the possible consequences of poor software security. While some companies suffer direct financial loss all would suffer from poor reputation. A stained security reputation can follow a company for a long time, creating a negative image for all future products. Microsoft’s products were early labeled as insecure and the company had to invest large sums to improve its products and repair its reputation (Howard and Lipner 2003). Whatever the reason, companies are today looking to improve their products’ software security. Unfortunately, the demand of quick release and constant flow of new products has forced companies to move away from methodical and big design upfront waterfall processes which includes most Security engineering processes. Companies and developers have instead adopted agile processes, which are iterative and incremental, and can easier adapt to new changes and release small release to customers at a faster rate. The agile process creates new problems for the security engineer that need to be addressed.

Using an agile process and developing secure software can create problems for developers (Wyrynen et al. 2004). The planning and strict structure that security engineering processes use has been replaced with a flexible, easy to adapt processes that values quick developer interaction instead of official meetings and documented decisions. This philosophical differences between security and agile creates different problems for security best practices, e.g. the practical activities that are performed in security engineering process. Risk analysis is an important step in identifying threats to the product before it has been completed and its design determined (Verdon and
Chapter 1. Introduction

McGraw 2004). However, since risk analysis is such an early preventive measure its effectiveness is determined by how much of the end result is already determined. In an agile process the end result and the used design might not be known until the developers start writing or finish writing the code. Therefore, it is necessary to adapt the risk analysis concept so it iterates and better integrates in a flexible agile process. Static code analysis is also a security best practices that instead focuses on implementation vulnerabilities (Chess and McGraw 2004a). While static code analysis in it self can easily integrate with an agile process there are many ways of doing so. It is therefore necessary to evaluate and study the different ways the tool can be used and integrated to determine their advantages and problems. Since one of the goals of an agile process it to release early and often to customers it is especially important that security is built into the process and not performed as a separate task at the end of development. Numerous studies have shown that correcting software faults increases in cost with every development phase (Boehm 1981)(Damm and Lundberg 2007)(Boehm 2002). As such, a software development organization that aims to produce secure software and use agile, should focus on early vulnerability detection. With an early detection approach the most effective method or tool should be deployed at the phase of development where it does the most good and can detect or prevent vulnerabilities at the lowest possible cost. What kind of method or tools should be used is not always obvious. Indeed, there are several different security development processes that all try to make security a part of the entire development lifecycle (De Win et al. 2008).

This thesis evaluates the interaction between an agile development process and development of secure software. Several studies are conducted to determine which current security engineering best practices can be used in an agile process, how they can be integrated, modified or new methods created to fulfill the different security engineering concepts. This thesis research questions are explained in greater detail in Section 2.
Chapter 2

Concepts

In this chapter the concepts of software vulnerabilities, development processes and
security engineering are explained. These key concepts are used throughout the thesis
and it is assumed the reader understands the following information.

2.1 Software Development

During development of software faults and flaws are introduced either from the imple-
mentation or from the design of the software. During runtime these faults and flaws can
propagate into failures that can result in vulnerabilities if the right conditions are met.
Failures and especially vulnerabilities increase the cost for the developers and require
them to spend time on maintenance instead of new features. Many software developers
rely on testing to reduce their maintenance cost and to create software with high avail-
ability. Unfortunately, testing mainly focuses on verifying the intended functionality
and not on detecting vulnerabilities.

Figure 2.1 illustrates the problem with most implemented software. The original
requirements or design does not always map correctly with the actual implementation
of the product. The implementation might be missing some features, labeled as A in
Figure 2.1, or the implementation might add unwanted functionality, labeled as B in
Figure 2.1. Testing and validation mostly focus on bugs and verifying requirements
and spends little time on detecting any extra functionality that might have been added,
i.e. vulnerabilities. The main exception is penetration testing that focuses entirely on
searching for unknown vulnerabilities. Therefore, many developers depend on pene-
tration testing to improve product security. However, while penetration testing does
find vulnerabilities it has to be performed late in development after all the functions have been verified as no more functionality is permitted to be added after the penetration testing has passed. Therefore, it is expensive to do penetration testing and it adds lead-time to the development cycle. An alternative would be to use a security development process that includes quality concerns with a security focus during the entire process, instead of trying to add it at the end of development. Some attempts like Microsoft’s Security Development Lifecycle have been studied with encouraging results (Lipner 2005). Figure 2.3 shows a layout of the security touchpoint development process (McGraw 2006b). However, these security process relay on stable project goals and waterfall development.

2.1.1 Waterfall Processes

The waterfall model used at the company runs through the phases requirements engineering, design & implementation, testing, release, and maintenance. Between all phases the documents have to pass a quality check, this approach is referred to as a stage-gate model. An overview of the process is shown in Figure 2.2.

We explain the different phases and provide a selection of checklist-items to show what type of quality checks are made in order to decide whether the software artifact developed in a specific development phase can be passed on to the adjacent phase.

Requirements: In this phase, the needs of the customers are identified and documented on a high abstraction level. Thereafter, the requirements are refined so that they can be used as input to the design and implementation phase. The requirements (on high as well as low abstraction level) are stored in a requirements repository. From
this repository, the requirements to be implemented are selected from the repository. The number of requirements selected depends on the available resources for the project.

As new products are not built from the scratch, parts from the old product are used as input to the requirements phase as well. At the quality gate (among others) it is checked whether all requirements are understood, agreed upon, and documented. Furthermore, it is checked whether the relevant stakeholders are identified and whether the solution would support the business strategy.

**Design & Implementation:** In the design phase the architecture of the system is created and documented. Thereafter, the actual development of the system takes place. The developers also conduct basic unit testing before handing the developed code over to the test phase. The quality gate checklist (among others) verifies whether the architecture has been evaluated, whether there are deviations from the requirements compared to the previous quality gate decision, and whether there is a deviation from planned time-line, effort, or product scope.

**Testing:** In this phase the system integration is tested regarding quality and functional aspects. In order to make a decision whether the system can be deployed, measures of performance (e.g., throughput) are collected in the test laboratory. As the company provides complete solutions (including hardware and software) the tests have to be conducted on a variety of hardware and software configurations as those differ between customers. The outcome of the phase is reviewed according to a checklist to see whether the system has been verified and whether there are deviations from previous quality gate decisions in terms of quality and time, whether plans for hand-over of the product to the customer are defined according to company guidelines, and whether the outcome of the project meets the customers’ requirements.

**Release:** In the release phase the product is brought into a release ready state. That
Chapter 2. Concepts

is, release documentation is finalized (e.g. installation instructions of the system for customers and user-guides). Furthermore, build-instructions for the system have to be programmed. Build-instructions can be used to enable and disable features of the main product line to tailor the system to specific customer needs. At the quality gate (among others) it is checked whether the outcome meets the customers’ requirements, whether the customer has accepted the outcome, and whether the final outcome was presented in time and fulfilled its quality requirements.

**Maintenance:** After the product has been released to the customer it has to be maintained. That is, if customers discover problems in the product they report them to the company and get support in solving them. If the problems are due to faults in the product, packages for updating the system are delivered to the customers.

### 2.1.2 Security Engineering

![Figure 2.3: Cigital Software Security Touchpoints development process.](image)

Security engineering processes support software development engineering in delivering solutions that prevent misuse and malicious behaviour. The processes improve the end products security by adding formal methods, requirements and predefined best practices for the developers. By having guidelines and concrete steps the developers
have to perform the processes tries to force the developers to focus on security. There are several different development processes that have a security focus and guide developers in different ways, some of the most common are:

Cigital Software Security Touchpoints (McGraw 2006b) has often been described as a lightweight security engineering process that integrates core activities in an existing development process and improves the quality and security aspect of the end product (Steven 2006).

Common Criteria (Keblawi and Sullivan 2006) is mature and well used security engineering principal that is ISO certified. A previous study has examined Common Criteria from an agile perspective and identified activities similarly to ours (Mellado et al. 2006).

The Microsoft Security Development Lifecycle (SDL) (Howard and Lipner 2006) is a software development process used and proposed by Microsoft to reduce software maintenance costs and increase reliability of software concerning software security related bugs. It is based on the classical waterfall model but there are attempts in making it more agile friendly (Sullivan 2008).

Other development processes are the SEI’s Team Software Process (TSP) (Humphrey 1999), Correctness by Construction (Hall and Chapman 2002), and several other security development processes (De Win et al. 2008).

Unfortunately, most of these processes are restricted to a specific software development method or are intrusive and hard to integrate into an already existing development process. However, some common parts, like the source audits, should be easily integrated onto an agile process. Just as it is not possible to test quality into software, it is impossible to add security features onto code and expect the result to be a secure product. Security must be part of the product's entire development lifecycle (McGraw 2006b).

### 2.1.3 Agile Process

Over recent decades, while market forces, systems requirements, implementation technology, and project staff were changing at a steadily increasing rate, a different development style showed its advantages over the traditional one. This agile style of development directly addresses the problems of rapid change. A dominant idea in agile development is that the team can be more effective in responding to change if it can:

- reduce the cost of moving information between people.
- reduce the elapsed time between making a decision to seeing the consequences of that decision.
To reduce the cost of moving information between people, the agile team works to place people physically closer,
- replace documents with talking in person and at whiteboards.
- improve the team’s sociability, its sense of community so that people are more inclined to relay valuable information quickly.

To reduce the time from decision to feedback, the agile team
- makes user experts available to the team or part of the team.
- works incrementally.

Making user experts available as part of the team gives developers rapid feedback on the implications to the user of their design choices. The user experts, seeing the growing software in its earliest stages, learn both what the developers misunderstood and also which of their requests do not work as well in practice as they had thought.

The term agile, coined by a group of people experienced in developing software this way, has two distinct connotations. The first is the idea that the business and technology worlds have become turbulent, high speed, and uncertain, requiring a process to both create change and respond rapidly to change. Agile development focuses on the talents and skills of individuals and molds process to specific people and teams, not the other way around. Agile teams are characterized by self-organization and intense collaboration, within and across organizational boundaries. Agility requires that teams have a common focus, a collaborative and quick decision-making process and the ability to deal with ambiguity and abstract goals. We distinguish collaboration from communication. Communication is the sending and receiving of information. Collaboration is actively working together to deliver a work product or make a decision. Scrum projects are characterized by intense 15 to 30 minute daily meetings in which participants constantly adjust to the realities of high-change projects. All too often, project teams are given the ultimate accountability for product delivery, while project management are given decision power. In order to innovate and react to change, agile teams reflect a better alignment of accountability and responsibility. It is again an emphasis, at the team level, on competency rather than process.

2.2 Security

2.2.1 Software Vulnerabilities

This section explains how a fault is classified as a vulnerability. This will make it possible to determine what the most common effect of reported vulnerabilities is. In security
terminology, this thesis is primarily interested in vulnerabilities. Vulnerabilities are a weakness in a software system. Typically, vulnerabilities have two possible origins, faults in the implementation of the software and flaws in the software’s design (Viega and McGraw 2002a). However, source code based vulnerabilities are still software faults; the faults can just propagate into a specific type of failure, a security vulnerability. In testing terminology, vulnerabilities are faults and faults are defined as the static origin in the code that during runtime dynamically expresses the unwanted behaviour or error. While the static analysis tool is searching for the fault or bug, testers are looking for the propagation of the fault. Figure 2.4 explains graphically how a vulnerability can have two possible origins, it also shows that while some vulnerabilities are first detected as failures, some are never detected and remain in mature code waiting to be exploited.

![Figure 2.4: Origin and propagation of vulnerabilities.](image)

However, not all failures are vulnerabilities, there are certain requirements that first have to be met to classify a possible failure as a vulnerability. A failure is labelled as security vulnerable if it, under any conditions or circumstances could result in a denial of service, an unauthorized disclosure, an unauthorized destruction of data, or an unauthorized modification of data. These represent the propagation or effect of an exploit. Below these effects are explained in greater detail.

- **Denial of Service** - Preventing the intended usage of the software. The most common Denial of Service attacks today are network based that try to exhaust system resources. Source code that does not always properly release a system resource might be exploited in the same manner, resulting in an exhaustion of
resources.

- **Unauthorized disclosure** - Extracting data from the software that was meant to be secret, e.g. customer data or passwords. Most common attacks today are disclosing data from databases or web sites, often in the form of encrypted, or worse, plain text pass-words.

- **Unauthorized destruction of data** - Destroying the data and preventing others from using it. Besides destroyed user data, configurations may be used to force the system to enter a default/unsafe state that later on can be exploited.

- **Unauthorized modification of data** - The data is not destroyed, but instead altered to fit the need of the attackers. This is often the most serious result and often requires that the attacker gains full access to the system.

### 2.2.2 Early Vulnerability Detection

The concept of early vulnerability detection is similar to early fault detection. The intention is to detect any anomaly resulting in a vulnerability in the product that would require effort to be corrected after the product has been released to customers. Efforts in detecting a specific type of vulnerability should also be focused in the phase and method that is most cost effective for that type of vulnerability. Studies in early fault detection have shown that detecting the fault earlier in development reduces the development cost (Boehm 2002). Because implementation vulnerabilities are also faults the same benefit should be there when using static tools. However, one characteristic of vulnerabilities is that they are harder to detect than regular faults. It might therefore be necessary to specialize the detection method to specific types of vulnerabilities. In addition, it is likely that for some vulnerabilities it might be more cost effective to detect them later in development. As an example, a complete manual source code audit with highly specialized security developers should, in theory, detect all implementation vulnerabilities. However, if the source code were larger than a few thousand lines of code the audit would require many experts and be very time-consuming (Porter et al. 1995). As such, it would be economically more sound to use penetration testing on release ready code instead of expanding the implementation phase to incorporate the enormous audit. For an early vulnerabilities detection method or tool it is therefore important to know what types of vulnerabilities are most likely detected and to what cost, then a strategy to detect the vulnerabilities effectively can be created.
2.2.3 Design Vulnerabilities

When designing a product it is possible to add requirements to the design that would affect the end products security. On such common design vulnerability is the locking of users after a predetermined amount of failed authentication attempts. This requirement is often labeled as a security requirement to prevent bruteforce password guessing. However, if a product implements this requirement they create design vulnerability were malicious users can lock other users and create a Denial of Service attack. These vulnerabilities can be costly to correct as they often require a rewrite of the source code. It is therefore beneficial to detect the threats as early as possible and most Security Engineering processes have some form o Risk Analysis to try and detect them.

2.2.4 Implementation Vulnerabilities

All software projects produce at least one common artefact, the products’ source code. At the code level, the focus is placed on implementation faults, especially those that are detectable by static analysis tools. However, knowing the implementation faults that are not detected is also interesting as it can be used to guide where other more expensive detection methods should focus their analysis on. Implementation vulnerabilities differentiate themselves from the design vulnerabilities because they only exist in the source code and are not part of the original design or requirements. The implementation vulnerabilities are also very language specific, especially the C and C++ coding languages are infamous for their ease of creating implementation vulnerabilities. The languages memory control is both its strength and weakness. The control the developer has creates the opportunity to create optimized and fast software but also insecure code that can easily be exploited. Some of the most common causes of implementation vulnerabilities that we have observed are, buffer overflows, format string bugs, integer overflows, null dereferences and race conditions.

2.2.5 Configuration Vulnerabilities

A finished product will eventually be deployed on a customer’s network. The setup of the product can be crucial for security. A product that has been designed safe and implemented correctly might still be vulnerable due to for configuration security. For example default users might be active on the product or the customer does not use the product as intended. To prevent configuration vulnerabilities developers’ aim at creating safe by default programs that have a default state that is secure and often featureless.
2.3 Security Engineering Best Practices

This section explains some of the most common security engineering best practices. The two most relevant activities for this thesis are further explained in greater detail in their own subsection.

2.3.1 Common Activities

Early in the development processes there are security activities such as, Security Requirements that cover both overt functional security and emergent characteristics that are best captured by Abuse Cases and attack patterns. The purpose is to identifying and documenting security and functionality for a given software project. Abuse Cases describe the system’s behaviour under attack; building Abuse Cases requires explicit coverage of what should be protected, from whom, and for how long. Developers can also identifying all possible user roles and their access level to the software by creating a Role Matrix. Security risk analyses uncover and rank architectural flaws so that mitigation can begin. Disregarding risk analysis at the early phases leads to costly problems down the road. Risk must normally be determined from application to application. Requirements Inspection is carried out in order to validate all the generated artefacts and it is generated as a Validation Report. Its aim is to review the quality of the team’s work and deliverables as well as assess the security requirements engineering process. Finally developers improve the design by performing Attack Surface Reduction to reduce the design and end products attack surfaces, by limiting entry points and simplifying interfaces.

With the end of the design phase the security engineering activities switch the focus towards the product source code. At the code level, the focus is on implementation bugs, especially those that static analysis tools, that scan source code for common vulnerabilities, can discover. A process should have mandatory Static Code Analyses with predefined rules and priorities. Coding Rules are helpful in clarifying the rationale for the deprecation of unsafe functions and help identify and recommend alternatives for these unsafe functions. While performing source Code Reviews on source code before the code change is committed to the source code repository increase the chance of detecting the vulnerability early (Rigby et al. 2008). An agile concept where developers code in pairs. Solving and reviewing problems directly as the code is written (Williams et al. 2000).

With the product created the security engineering processes address the verification and validation of the product. Penetration Testing provides a good understanding of fielded software in its real environment. It does this by simulating real world working conditions and attack patterns. Manual risk-based security testing based on attack
patterns is an alternative to Penetration Testing. The risk analysis is used to guide and focus the test cases towards the most important parts of the product. Fuzzy testing uses tools to automatically generate test cases that explore the products input validation.

At the end a final security review can be conducted to; reviews all threat models, validates security tool results, reviews all outstanding/deferred security bugs, reviews all exception requests as part of the security program. Furthermore, the model elements already in the repository could be modified in order to improve their quality.

### 2.3.2 Risk Analysis

The importance of security risk analysis in software projects can be observed since most security engineering process include the practice and it is viewed as essential for the compliance of the security engineering process. Although, different process might use different terms and names for the practice, the core concept stays the same. The vulnerable areas covered under the course of a risk analysis are:

- Assessment of risk in the product and new features.
- Characterization of risk based on likelihood of successful attack and damage caused.
- Communication of the risk to effected parties.
- Risk management, the process of handling and mitigating the risks.

Risk analysis is a technique employed to identify and assess various factors, which may jeopardize the security of a product. Thus risk analysis covers the process of scientific assessment of such threats that can create vulnerabilities in the software and which should be prioritized. Risk analysis techniques are helpful in defining preventive measures to reduce the probability of software design vulnerabilities. It includes identification of various countermeasures to successfully deal with possible threats.

### 2.3.3 Automated Static Code Analysis

With the term static analysis we mean an automated process for assessing code without executing it. Because the technique does not require execution, several possible scenarios can be explored in quick succession and therefore obscure vulnerabilities might be detected that would otherwise be very hard to detect. A contrast to static analysis is dynamic analysis that does its analysis during runtime of a system. However, dynamic analysis requires that the code is executed with sufficient test cases to execute all possible states and it slows down the test cases substantially (Ernst 2004). Static
analysis does not have these shortcomings and is theoretically easier to integrate into development, as it does not require a complete working product before analysis can begin. Today most security aware analysis tools are static while performance analysis tools are dynamic. In this thesis, we will focus on static analysis as we believe it provides the better results early in software’s development. Static analysis can aid penetration testing but it does not replace security specific testing, it should be seen as a complement that can detect some of the vulnerabilities early and save time and money for the developers.

To detect the vulnerabilities, static analysis tools used predefined rules or checkers that explain how vulnerabilities look. However, both the technique and the checkers can report incorrect warnings that do not cause any problem during execution; these are referred to as false positives. The precision of the analysis determines how often false positives are reported. The more imprecise the analysis is, the more likely it is to generate false positives. Unfortunately, precision usually depends on analysis time. The more precise the analysis is, the more resource consuming it is and the longer it takes. Therefore, precision must be traded for time of analysis. This is a very subtle trade-off, if the analysis is fast it is likely to report many false positives in which case the alarms cannot be trusted. This is especially true for instant feedback tools that perform fast analysis. With a high number of false positives, developers would often spend more time excluding false warnings compared to correcting faults. On the other hand, a very precise analysis is unlikely to discover all anomalies in reasonable time for large programs. One way to avoid false positives and shorten analysis time is to filter the result of the analysis, removing potential errors that are unlikely and pruning unlikely paths. However, this may result in the removal of positives that are indeed defects. These are known as false negatives, an actual problem that is not reported. False negatives may occur for at least two other reasons. The first case is if the analysis is too optimistic, making unjustified assumptions about the effects of certain operations. The other case which may result in false negatives is if the analysis is incomplete; not taking into account all possible execution paths in the program. There are a number of well-established techniques that can be used to trade off precision and analysis time.

A framework for static analysis is said to be sound if all instances of an examined defect type are reported, i.e. there are no false negatives but there may be false positives. Traditionally, most frameworks for static analysis have aimed for soundness while trying to avoid excessive reporting of false positives. However, most commercial systems today (Coverity Prevent, Klocwork K7 and Fortify) are not sound and they will not find all instances of a defect. Commercial tools also focus more on lowering the number of false positives then research tools do.

Because of the focus of this thesis in a specific type of faults, vulnerabilities, we cat-
egorize the output or lack of it, from static analysis tools in the following four groups:

- **False Positive:** Warnings that do not cause a fault in the software or state and un
- true fact. These are often caused be either weak verification or incomplete/weak checkers.

- **True Positive:** Correct reports of faults within the software. However, the fault does not allow a software user to create a failure that would result into any of the four security consequences stated in section 2.2.1.

- **Security Positive:** Warnings that are correct and can by exploited into any of the four effects in section 2.2.1.

- **False Negative:** Known vulnerabilities that the static analysis tool did not re
- port. Either because the analysis lacked the precision required to detect them or because there are no rules or checks that look for the particle vulnerability.
Chapter 3

Research Approach

In this section, we explain the different research methods used to produce our contributions in this thesis.

3.1 Research Questions

Software development departments in Ericsson AB aim to improve their software’s security by integrating security in their ways of working and not add security as an after thought on the finished product. Integrating security in an entire development process is not a small task and this thesis does not answer all the questions or address all possible recommendations.

The contribution of this thesis is to understand how software development methods and tools can aid developers to create security software while working in an agile process. There are three main questions this thesis tries to answer that are then divided into sub-questions to better understand the contribution.

The first study, with the sub questions in Figure 3.1, starts with Paper A that combines elements of well-known security risk analysis methods to create a risk analysis method that could be easier for non-security developers to use and still result in a useful risk analysis. In Paper B the risk analysis method is enhanced and adapted to the agile process through several prototypes and trails. These two papers answer our first research question (Q1), the purpose of this question is to identify how an iterative risk analysis can prevent design vulnerabilities and create a secure by default product that requires less hardening for configuration management security.

Most of current risk analysis methods depend on having a clear goal and plan for
Chapter 3. Research Approach

Figure 3.1: The thesis first research question and its three sub-question.

Q1: How can risk analyses be performed in an iterative way and still provide developers with an overview security state and a plan forward?

1.1 What information can be reused in an iterative risk analysis? (Paper A)
1.2 How can developers create a prioritized list of threats? (Paper A & B)
1.3 What visualization and usage improvements can aid in the creation process? (Paper B)

Figure 3.2: The thesis second research question and its four sub-question.

Q2: What are the benefits and pitfalls in using static code analysis as part of a development process?

2.1 Do any developer expertise’s aid in the classification of static analysis warnings? (Paper C)
2.2 How well can developers correct static analysis warnings? (Paper D)
2.3 Does the tool deployment strategy affect the end result? (Paper D)
2.4 What results can be expected after using a static analysis tool for a prolonged time? (Paper D)

The second meta study, Figure 3.2, focuses on the automated static analysis tools and its ability to detect implementation vulnerabilities. The goal of this study is the second research question (Q2). In Paper C, an experiment was performed to determine
how well developers of different experience levels could correctly classify warnings from the static analysis tool. Then in Paper D the results from several case studies are combined to examine the effect of long term static analysis usage and a new experiment to examine how developers correct the warnings.

Many studies have identified the benefit of static code analysis and other studies have improved the technology used in static code analysis. However, there has been a distinct gap between technology and actual usage research of the tool. This research question answers the possible practical implications of use static analysis in a large industrial setting.

Q3: How well do existing security engineering methods and tools integrate in an agile process?

- 3.1 What security engineering activities do developers initially exclude from the agile process? (Paper E)
- 3.2 Which security engineering concept can be adapted to the agile process? (Paper F)
- 3.3 What is a security enhanced agile process and what security concepts does it contain? (Paper F)

Figure 3.3: The thesis third and final research question and its three sub-question.

Finally, in Figure 3.3 the last meta study examines how existing security engineering practices integrate in an agile development process. Answering research question three (Q3). Paper E uses an open-ended interview study to determine what security engineering practices have a low probability of integrating with agile and do not warrant further investigation. Then in Paper F, practical case studies and workshops evaluate the security engineering methods and tools in a live setting to determine if they;

1. Can be used as is.
2. Require modification.
3. Are not agile compatible.

The agile security research has mostly focused on literature reviews and document comparisons. By involving industry and performing practical case studies

This thesis answers the above question through case studies at Ericsson AB. By
Chapter 3. Research Approach

answering the above questions, this thesis provides a stepping stone and method of reaching the final goal of incorporating security in an agile development process.

3.2 Research Methods

The research objective of this thesis is to study and evaluate methods and tools in practical industrial context, in this case security engineering activities that can be integrated with agile. We therefore believe that case studies and experiments are suitable methods to archive this goal (Wohlin et al. 2003). Below is a list of methods used during this thesis to answer the question that arose.

**Action research:** In action research the goal is to introduce an intervention in a realworld setting and then observe what the affect of the intervention is. The researcher is actively involved in introducing the intervention and making the observations, in fact the researcher takes an active part in the organization. The steps of action research are planning of the intervention and the collection of the data to capture the intervention effect, the implementation of the actual intervention, and the collection of data and their interpretation (Somekh 2006). As pointed out by Martella et al. (Ronald C. Martella and Marchand-Martella.W 1999) much can be learned by continuously observing the effect of a change after inducing it. However, as the researcher is actively involved in the team work action research is an effort intensive approach from the researchers point of view.

Action research was the main drive in Paper B to prototype and improve the risk analysis concept to better integrate and provide a benefit to the agile process.

**Survey:** A survey studies a phenomena for a population by surveying a sample that is representative for that population. In order to collect the data from the sample questionnaires and interviews are often used. Questionnaires are sent out to a widely distributed sample of people and is therefore surveys are seen as research-in-the-large. The surveys usually contain fixed questions that provide quantitative answers that are easy to analyze. The surveyed people are often a sample group that represents the general population (Creswell 2003).

Surveys were primarily used to examine the human factor, in Paper D surveys were used to collect developer experience data and in Paper E surveys were used to determine how the different projects had deployed and used the static analysis tool in a real setting. Also in Paper E, developer interviews are used to get their options on what security engineering activities are agile friendly and worthy of integrating.

**Experiment:** Experiments are controlled studies that are designed to test one spe-
cific impact or variable while at the same time controlling all other factors that might influence the outcome variable. Experiments are referred to as research-in-the small because they typical address a limited scope (Kitchenham et al. 1995).

Controlled experiments were used in both Paper D and Paper E to examine a specific variable. In Paper D the developers experience was compared to their ability to correctly classify warnings reported by the tool. In Paper E the experiment was extended to include the developers’ capability to correct the warnings.

**Case study:** A case study is often used to study industry projects and is considered research-in-the-typical. This normally makes it easier to plan the experiments but the results are harder to generalize and sometimes to analyze (Wohlin et al. 2003). The cases are objects of the real world studied in a natural setting, in software engineering this means they are real software organizations, software projects, software developers, etc. Case studies are conducted by defining the case to be studied, the units of analysis, as well as a data collection strategy. With regard to case studies a number of decisions have to be made. The first one being whether the study is of confirmative or exploratory nature. In comparison to controlled experiments there is much less control in an industrial case study.

In Paper B and in Paper C, case studies were used to determine the capabilities of static analysis tools. The case studies examine the output of the tools and classified the results into a taxonomy. Also in Paper F case studies re used to practicably verify the different security engineering activities.

**Post-mortem analysis:** Post-mortem analysis are often used in conjunction with case studies, they are used to collect historical data. Therefore post-mortem analysis are similar to surveys but have the same scope as case studies (Wohlin et al. 2003).

Post-mortem analysis was used in Paper E with the purpose of the analysis was to determine how static analysis had been used in an industry setting, what vulnerabilities had been detected and how they were corrected.

### 3.3 Research Validity

There are four types of validity: internal, external, construct and conclusion validity (Wohlin et al. 2003). Because the studies were performed in an industry setting, they have a high probability of being realistic and having a real impact. There are however, some more validity threats that need to be assured.

The *Internal validity* "concerns the causal effect, if the measured effect is due to
changes caused by the researcher or due to some other unknown cause” (Wohlin et al. 2003). As an example, there might be unknown factors that affect the measured result. Internal validity generally becomes a significant threat to industry case studies as their environments often changes. However, because our research observed and interacted with the projects closely any change to the environment that would affect our result could easily be detected and examined. The author was also always present in any analysis so that the human factor would not change over time.

The external validity concerns the possibility of generalizing the findings (Wohlin et al. 2003). Generalization becomes a problem because most of the studies done in the course of this thesis are case studies or experiments in a fixed setting. The case studies are often only valid in the context they were performed and do not isolate the measured attributes as an experiment would. However, through-out the thesis the case studies have tried to keep as generalized context as possible. Several different projects have been examined, developers from four different countries have participated and both commercial Ericsson software and open source has been examined. This was all done to ensure some generalization of the results. Conducting industry studies has benefits over student experiment but at the same time there is a risk that the results can not be generalized away from the need of the company towards the general good. During the studies of this thesis some precautions were taken to improve the generalizability of the results. By using such large company for the studies several of the studies could use results from several different development units and products. These development units and products were located in different countries and utilized different practical ways of developing software, in a sense while all the studies were done at Ericsson the diversity of the environment makes results much more generalizable.

However, the content of the product are all the same. They are all telecom product that have a strict server architect and focus on availability and throughput. While the studies have focused on server architecture the methods used would be similar for client side software. The outcome of the studies could differentiate with client software. However, the scientific results would be similar. For the identification of vulnerabilities the difference of server and client software would be in the type of threats that are detected and not as much importance in the methods used to detect them. The process oriented measure used to detect and prevent a threat is the same for client or server software. It is none less a weakness that all studies have focused on server software; these are however the systems that often require higher security standards as server software is expected to listen to other software and react on the data they receive.

The construct validity “reflects our ability to measure what we are interested in measuring” (Wohlin et al. 2003). In most studies, we measure solid data that is not open for interpretation and shows what we are actually interested in measuring. How-
ever, in Chapter C we measure developer experience that can be subject to interpretation. By measuring experience in years instead of perceived expertise we reduce the chance that interpretation does not sway the results.

The conclusion validity concerns the correctness of the conclusions the study has made (Wohlin et al. 2003). Three typical threats to conclusion validity are reactivity, participant bias and researcher bias (Robson 2002). These threats are largely avoided by examining real data instead of specially generated lab code. The research bias is avoided by using proof of concepts to prove vulnerabilities or external validation sources like bug reports. Developers also interacted in the studies and stated their opinion if a warning was a vulnerability or not. Also to prevent research bias most the studies were conducted with the aid of developers and by letting the developers determine the meaning of the outcome in a workshop. However, research bias could not be removed complete because it was the researcher that conducted the studies and providing the area expertise. A researcher bias was mitigated partly by having clear rules on what information the researcher could give to the developers and how much interaction was allowed. The level of interaction depended on the different studies.

3.4 Research Environment

At the beginning of these thesis the company used an waterfall model that was design upfront and had a heavy focus on solving problems before the arias, this later changed to an agile process that was instead more flexible and could adapt to change. We begin with explaining the waterfall process, then the agile processes followed by simplified explanation of the products.

3.4.1 Waterfall Process

The waterfall model used at Ericsson AB runs through the phases requirements engineering, design & implementation, testing, release, and maintenance. Between all phases the documents have to pass a quality check, this approach is referred to as a stage-gate model. An overview of the process is shown in Figure 2.2. Products are developed in a number of consecutive releases. The product development cycles are in large and last from one to two years, also the scope of the project is defined at the beginning of a development cycle. The scope is set by selecting a number of requirements from the requirements repository. These requirements are refined into a requirement specification that should cover an entire project. Based on this requirement specification, a new version of the system is developed and released to the customers. Due
3. Research Approach

to the length of the projects, it often happens that customers’ needs change when the project is still ongoing. This means that some of the requirements specified in the beginning become obsolete. Therefore, some requirements need to be added, some need to be changed, and some need to be deleted to better match the customer expectations. These changes affect the security measures that have been performed on the design and would therefore be required to be redone.

In the design phase, the architecture of the system is created and documented. Thereafter, the actual development of the system takes place. It is during the implementation phase that the early vulnerability detection tool should be used. Penetration testing would be performed at the end of testing as a quality door and has therefore a higher cost in correcting the vulnerabilities. The static analysis tool studies in Paper C & D were conducted in this phase of development.

During the course of this thesis, the development process at Ericsson changed from a longer waterfall based process to a shorter more iterative Agile development process. This development process is fully explained in a paper written by Tomaszewski (Tomaszewski et al. 2008a). This section focused only on the design and implementation phase of the old process as it is the only one that is relevant for this thesis.

3.4.2 Agile Processes

The process model used at the company is described and thereafter its principles are mapped to the incremental and iterative development; SCRUM, and Extreme Programming (XP). The model is primarily described to set the context for the case study, but the description also illustrates how a company has implemented an incremental and agile way of working. Due to the introduction of incremental and agile development at the company the following company specific practices have been introduced:

The first principle is to have small teams conducting short projects lasting for at most three months. The duration of the project determines the number of requirements selected for a requirement package. Each project includes all phases of development, from pre-study to testing. The result of one development project is an increment of the system and projects can be run in parallel.

The packaging of requirements for projects is driven by requirement priorities. Requirements with the highest priorities are selected and packaged to be implemented. Another criterion for the selection of requirements is that they fit well together and thus can be implemented in one coherent project.

If a project is integrated with the previous baseline of the system, a new baseline is created. This is referred to as the latest system version (LSV). Therefore, only one product exists at one point in time, helping to reduce the effort for product maintenance. The LSV can also be considered as a container where the increments developed by the
projects (including software and documentation) are put together. On the project level, the goal is to focus on the development of the requirements while the LSV sees the overall system where the results of the projects are integrated. When the LSV phase is completed, the system is ready to be shipped.

Figure 3.4: An overview model of the agile processes used by the examined projects. Three distinct roles (SPM, DP and LSV) are relevant to our study and shown in the model.

The anatomy plan determines the content of each LSV and the point in time when a LSV is supposed to be completed. It is based on the dependencies between parts of the system developed which are developed in development projects, thus influencing the time-line in which projects have to be executed.
Chapter 3. Research Approach

If every release is pushed onto the market, there are too many releases used by customers that need to be supported. In order to avoid this, not every LSV has to be released, but it has to be of sufficient quality for a potential customer release. The release project in itself is responsible for making the product commercially available, performing any changes required to alter a development version into a release version.

In figure F.1 an overview of the development process is provided. The requirements packages are created from high priority requirements stored in the repository. These requirements packages are implemented in projects resulting in a new increment of the product. Such a project is referred to as a Development Project and has a duration of approximately three weeks (time-boxed). Each DP contains design, implementation and testing of the requirements. It might happen that a DP is started in one sprint to later be released in another. When a project is finished developing the increment, the increment is integrated with the latest version of the system, referred to as last system version (LSV). The LSV has a predefined cycle and no components dropped after a new sprint will wait until the next sprint to be tested. From the LSV there can be different releases of the system. These are either potential releases or customer releases.

3.4.3 Products

In the Ericsson case studies all the examined products were server software that operate on a client - server basis, e.g. the end user never logs onto the server but instead communicates via a client software or middleware. Therefore, most of the detected vulnerabilities are of remote exploitation interest and the focus lies on server vulnerabilities. But local exploits are also explored because company employees have local access to the servers. Because the servers provide functionality and in some cases handle financial data, we can not exclude the local threat to the products.

Most of the mature products in the study were written in C and C++ language. While all new products use Java and Java Enterprise. This was due to a company wide directive that new products should use Java instead. In some cases old products were migrated to Java. Since the static analysis papers were written before the other papers the products in those studies are written in C and C++ while later papers use Java products instead.

3.5 Vulnerability Taxonomy

We used the taxonomy (Tsipenyuk et al. 2005) where eight different groups are used to classify the vulnerabilities. This taxonomy focuses on implementation faults and is especially good for static code analysis tools. The taxonomy explains the cause
of vulnerabilities, but not necessarily the effect of it. The following eight groups are defined in the taxonomy:

- **Input validation and representation** - Meta-characters, alternate encodings, and numeric representations cause input validation and representation problems.

- **API abuse** - An API is a contract between a caller and a callee: the most common forms of API abuse occur when the caller fails to honor its end of the contract.

- **Security features** - Incorrect implementations or use of security features, e.g. not correctly setup encryption.

- **Time and state** - Distributed computation is about time and state, e.g., for more than one component to communicate, states must be shared, which takes time.

- **Errors** - Errors are not only a great source of ”too much information” from a program, they’re also a source of inconsistent thinking that can be exploited.

- **Code quality** - Poor code quality leads to unpredictable behavior, and from a user’s perspective, this often manifests itself as poor usability. For an attacker, bad quality provides an opportunity to stress the system in unexpected ways.

- **Encapsulation** - Encapsulation is about drawing strong boundaries around parts of the system and setting up barriers between them.

- **Environment** - Environment includes everything outside the code that is still critical to the security of the software. This includes the configuration and the operating systems environment.
Chapter 4

Outline and Results

This section presents the results of this thesis. Each subsection represents a publication and its contributions. In short, Subsection

4.1 Prioritizing Countermeasures through the Countermeasure Method for Software Security (CM-Sec)

Software security is an important quality aspect of a software system. Therefore, it is important to integrate software security activities throughout the development lifecycle. In this paper we answer research question 1.1 and 1.2 by proposing a novel method focusing on the end product by prioritizing countermeasures. The method provides an extension to attack trees and a process for identification and prioritization of countermeasures. The approach has been applied on an open-source application and showed that countermeasures could be identified. The purpose of the countermeasure graph is to determine what security countermeasures would be useful to include in the product and at the same time identify what countermeasures already exist. At the same time we determine what the greatest threat to the product is and how well we prevent them. To achieve this we need to discover what attacks can be made and how they are stopped, with traditional threat analysis or attack trees this process is direct and tries to immediately identify attacks. We instead divide the work in four distinct steps and then use hierarchical cumulative voting to calculate the impact of every step.

Goals: The first step is to understand why anyone would attack the product and what their gain would be. This is often the same as the intended usage of the product.
and is used as a guideline to see the big picture.

Agents: Thereafter the Agents are identified. Agents are users, roles and software that interact with the products. Preferably the entire range of possible users should be presented in this step, form end users to administrators and outside software that interact with the product.

Attacks: Combining the Agents with a Goal we then look for Attacks. This step of the threat analysis is made easier because we have a more clear idea who and why the attack would accrue. We do not focus on if the attack would work, only if it is a possible route for the Agent to take to achieve his desired Goal.

Countermeasures: The final step is identifying what Countermeasures could be used to prevent an attack. A countermeasures can prevent one or more attacks.

### 4.2 Countermeasure Graphs for Security Risk Assessment: An Action Research

Risk and threat analysis are an important part of early vulnerability prevention. Risk analysis, threat models and abuse cases are different methods of identifying and preventing risks. However, none of these methods integrated well with agile process that the examine company used. Because of the small iterations and incomplete designs many of the traditional methods did not aid the risk assessment process. To answer research question 1.2 and 1.3 this paper evaluates and improves our novel risk analysis method. Through an action research approach and with the aid of several industrial projects, this paper has improved a novel risk analysis method. The countermeasure graph focuses on identifying and ranking protective measures against attacks product. By creating a model of the attack surface against the product it aids the developers in identifying attacks and ways of preventing them. The model also allows for incremental work and makes it easier to transfer the knowledge to future increments and reuse previous work.

When comparing the results of the improved agile risk analysis to a traditional Peltier analysis it was shown that the developers could much easier reuse the existing analysis and focus their efforts on the few new requirements for the next development iteration. This resulted in new risk and important information to the developers that would implement the new feature. The importance of general security risk, that were common in the traditional risk analysis, diminished with each iteration.

The presentation of the risk analysis played in large role in the developers acceptance of the tool. While post-it notes were used in the initial version of the method it quickly became necessary to use a tool to create a easy overview of the current assets.
The tool also aided in the complex math that prioritizes the threats and countermeasures for the developers. The final benefit of using a tool was the ability to easily save the results and share them with other developers. As such, it was determined that to meet our goals for a successful risk analysis a tool is necessary, even though agile processes prefer developer interaction before using tools.

In the process of creating the agile friendly risk analysis, the paper also proposes a structured way of adapting a security engineering activity to an agile process. By using action research and utilizing industry project with real developers it was possible to focus the integration on their needs and create a risk analysis that is continued to be used after the study was completed.

4.3 Static Code Analysis to Detect Software Security Vulnerabilities: Does Experience Matter?

In this chapter, we examine one of the major pitfalls when using static code analysis as an early vulnerability detector and contribute to research question 2.1. Because the tool requires that developers to examine the result from the tool there are potentials for human error, similar to code reviews were developers have to read and understand the code. This is especially true for static analysis tools due to the way they are often used. The process of classifying a warning and correcting it is often separated. This separation is not desirable, but was observed to be very common. Because of this, it might be important that the initial classification of warnings is correct, or else vulnerabilities might be ignored.

To determine what impact the human factor played in the tools’ usefulness a group of 34 developers were asked to use the tool and classify a pre-selected random sample of warnings. This was done in a controlled environment where the developers’ experience and knowledge could be compared to their ability to correctly use the tool. We wanted to determine if all developers could use the tool as an early vulnerability detector or if only developers with a specific knowledge should use the tool.

The developers could classify the warnings as false positives, true positives or security warnings. They also specified how confident they were in their classification. From the experiment, some issues became clear. First, the developers could not judge by themselves how correct their classification was, there was no correlation between their confidence and how correctly they answered. This is not good because the developers cannot determine when they should ask for help when using the tool. When examining the classification we saw that very few false positive warnings were classified correctly. In the majority of cases the developers would classify a false positive
Chapter 4. Outline and Results

as a bug that should be corrected. This increases the cost in using the tool and introduces a new source of potential faults. Neither development experience nor specialized skills helped the developers in classifying the false positives and no group, with the data we had, could be created that had more than random chance in classifying false positives correctly. On the other hand, classifying the true positives was easier and the majority of developers would have corrected the true positives as expected. The security warnings initially followed the same pattern as the false positives, very few developers correctly saw that the warning was security related and needed extra attention. However, developers that had used a static analysis tool prior to the experiment had a better than random chance in classifying the security warnings. Combining experienced groups and only looking at developers with both security experts and static analysis experience created a group that would correctly classify the security warnings 67% of the time. However, this group consisted only of 6 developers and it is concerning that only 6 out of the initial 34 got acceptable results. It also puts in question how useful the tool would be if so few had the necessary skill-set to effectively use the tool.

4.4 Improving Software Security with Static Automated Code Analysis in an Industry Setting

This chapter examines two years of industry experience with static analysis tools (Coverity Prevent) and contributes to research question 2.2-2.4. The deployments of several projects are examined post-mortem and three distinctly different deployment strategies are identified. From these projects, some are further examined and their historical data is collected to determine what types of vulnerabilities have been corrected. The developers’ usage is investigated by first letting them classify warnings, and then correct the warnings in the code, and lastly their historical actions are examined. This is done to identify any success factors or problems in using static code analysis tools as early vulnerability detectors.

During deployment we identified those three strategies which were distinctly different from each other. Least successful was a very open approach where the tool was provided and developers were free to use it. These projects seldom had any data or had never used the tool. The second strategy was a champion approach where a developer was responsible for the tool and its promotion. This was moderately successful, but it was imperative that the champion was dedicated and stayed in the project. The most successful strategy was to integrate the tool into the projects configuration management system, in particular the bug tracking system. This deployment strategy produced the most data and had a larger group of developers that used the tool as an early vulnera-
While examining the historical data from four different projects it is clear that the tool’s strength lies in memory related faults. In every project, the largest groups of vulnerabilities were either buffer or reference operations that could be exploited. While there were some race conditions, they were few and most of the tool’s checkers focused on memory operations.

The classification from the previous chapter was expanded in this study and the developers were asked to also correct the code and not only classify the warnings. The false positives could be divided into two groups, one that often resulted in harmless corrections, meaning a correction that would not affect the product and would run correctly even after the unnecessary correction. The second group consisted of harmful corrections that would either break the code so that test cases would fail. Even worse, in seven cases the correction of a false positive actually created a new vulnerability in the code. This result is very unfavorable and puts the entire idea in question. However, the correction of the security warnings showed better result, the majority of warnings were corrected in such a way that the code would become safe. This was true even for the developers that did not classify the warnings as a security threat. This indicates that the initial classification does not matter as much as first assumed, and that independently of development experience the static analysis tool is still useful for all developers. While this was true for the majority of warnings, probably because they often only had one logical correction, there was one large exception. A file base race condition had more unsafe corrections than it had correct ones. This was probably caused by lack of knowledge of how race conditions work because most of the developers that answered correctly had security experience. Most of the memory related vulnerabilities, a majority, were corrected as they should to create safe code.

While examining the product’s historical data some negative trends were observed in the usage of static analysis tools. Developers often classified warnings as false positives if the warning did not disappear after they thought they had corrected it. The data showed that their fix was not correct and that the fault still remained, but developers instead saw it as a false positive. Some complex warnings that required specific data-flows were often fixed incorrectly so they would break test cases. These fixes were then reverted and the warning classified as a false positive. Because of these trends the number of false positives in deployed static analysis projects was more than twice the predicted 20% that earlier studies had indicated. The last negative trend was that developers often viewed the results from the tool as minor faults; this was even true for the warnings that had been identified as solutions to known vulnerabilities. The most probable cause is in the way the information is presented. In bug reports the failure is often described vaguely and developers have to spend time finding the cause of the failure. With the static analysis tool the cause of the fault is directly identified, this creates
the illusion that the fault is unimportant and would never cause a serious failure.

### 4.5 Agile development with security engineering activities

Agile software development has been used by industry to create a more flexible and lean software development process, i.e. making it possible to develop software at a faster rate and with more agility during development. There are however concerns that the higher development pace and lack of documentation are creating less secure software. To answer research question 3.1 we have therefore looked at three known Security Engineering processes, Microsoft SDL, Cigatels touchpoints and Common Criteria and identified what specific security activities they performed. We identified 31 security activities that was performed in well established Security Engineering processes. We then compared these activities with an Agile development process that is used in industry. Developers, from a large telecommunication manufacturer, were interviewed to learn their impressions on using these security activities in an agile development process.

All three traditional processes scaled badly to the chosen agile security processes because of too high costs or not being enough beneficial for the agile conditions, i.e. time constraints and reiterated increments of the product. Preliminary findings, based on the interviews, show that especially the design and testing phase scaled badly.

Microsofts SDL had the largest negative effect on an Agile development process. Developers where particularly negative to Cost analyses and the purpose of a Cost analyses is to focus the security push on the area that creates potentially the greatest cost and damage. However, developers did not believe that it would be possible to perform a sound Cost analyses at that early stage of development and that the benefit is no longer as good at the later phases of development.

Cigatels Touchpoints did not have any negative activities during requirement, design or implementation phase. However, during testing there are several test methods that raise the cost without providing a large benefit. The Touchpoints testing methods all relies on having a ready product and then conduct a, mostly manual, test phase where the implementation code is not changed. This requirement by the tests methods makes them very Agile unfriendly, more specifically test driven unfriendly. An alternative would be to adapt or device a new test method that has the same benefit but uses test-driven and a large automated test suit instead of manual after release test methods. Lastly the concept of having an external, out of project, review is very hostile to a process that is depended on no bottlenecks or other blocking issues.
Common Criteria that is more of a metrics and measurements scheme does not have any implementation or testing activities. However, its activities for the products architecture assumes that there is an initial and final design created and then resulting in a implementation phase. For Agile projects the design phase if often very short and any design problems are solved during implementation by the team in a group effort. The group collaboration and daily stand-ups could replace the concept of a final architectural design.

We produced a security enhanced Agile development process that we present in this paper. This new Agile process use activities from already established security engineering processes that provide the benefit the developers wanted but did not hinder or obstruct the Agile process in a significant way. In a future work these proposed security processes will be integrated and tested in a practical experiment with the existing development process.

4.6 Integrating security engineering in an Agile process

In this paper we examine how the software development transition, from a former waterfall model to a more flexible agile software development process, expounds security issues. It answers research question 3.2 and 3.3. In the studied industrial case, software development projects are investigated involving 5 different phases (requirement, design, implementation, test and release). Within originally a waterfall model three external tools (Cigatel Touchpoints, Common Criteria and Microsoft Security Development Lifecycle Process) for improving quality and security aspects, are investigated. Three different teams within an agile setting were identified for improving the security of the end product; the software product manager (SPM), the development project teams (DP) and latest stable version test team (LSV). Developers working within an agile development process were interviewed and the results were implemented forming a new agile software engineering process. Some activities, compared to the waterfall model, are rejected as too time consuming or being too restricted in a simplified agile setting.

In all 31 activities were investigated originally identified from different waterfall processes. For the agile setting 20 of these activities were used, 6 directly and 14 in a modified version representing 7 different agile activities, i.e. in all 13 different agile activities (research question 3.2). Some of the activities were repeated or they used different methods for solving the same problem, so not all activities are relevant to use. The issue is instead to find the appropriate combination regarding security activities. Moreover this should be done in an agile environment, i.e. with restricted time and limited overall resources for the teams involved. In total, we found 3 SPM activities
covering requirement and design, 7 DP activities within design, implementation, test and release and finally 3 LSV activities within test and release. Instead of having a pro-active waterfall model the agile activities are reactive, i.e. development instead of trying to solve very complex problems before the full understanding of the product (research question 3.3).

The agile process is good at dividing and conquering the known problems in the software and focusing software development towards the real business needs. The short iterations make it easy to accept changes and re-prioritize the requirements. However, secure code development is more about coding principles and process activities that lead to better security, often depending on traditional top-down, document-centric approaches which are agile unfriendly. In the same way agile cannot help enforce code style, it won’t help with secure coding guidelines but it will hamper the integration of mature security activities. This was most clearly seen in the design activities that often relied on a finished product design that does not exist in the agile process. It therefore become very hard to reuse existing security practices and instead requires rethinking security for the agile process.
Chapter 5

Discussion and Conclusions

Research Question 1: How can risk analyses be performed in an iterative way and still provide developers with an overview security state and a plan forward?

Through prototyping and developer feedback an novel approach was devised that has the potential to better integrate in an agile process then traditional risk analysis methods. By creating a mind map that focuses on the prevention of threats, instead of attacks, it makes it easier for developers to begin a risk analysis. By using hierarchical cumulative voting the method creates a prioritized list of protective measures that developers should implement. The cumulative voting also aids ranking of threats as the developers compare the same type of issues with each other instead of an abstract scale that might or might not fit with the current case. By having a tool and the same model in each risk analysis it was possible for the developers to iterative and reuse the previous results. The model provided a better security sense and metric for each reuse, making it easier to plan and evaluate new security features and threats.

To answer this research question we used an interactive action research method and borrowed concepts from agile development. By having constant feedback from our customers, the developers, and by including them in our studies, we made sure that the results were inline with their needs. There is however a risk that the method would be too narrow and not be generalizable for other projects. To mitigate this risk several projects took part in the studies, used the method and provided feedback. The method has also been adopted by the projects and is continuously used even though the study has ended and the author is no longer present. This indicates that the authors affect on the methods results were not to severe, which was a risk because of his involvement in
Chapter 5. Discussion and Conclusions

the risk analyses.

Research Question 2: What are the benefits and pitfalls in using static code analysis as part of a development process?

A large scale industry study that consisted of several smaller studies investigated how developers deployed and used static analysis tools. There was a clear benefit because the tool was capable in detecting known vulnerabilities that had been reported by testers or customers. The tool was also effective in identifying unknown potential threats that cost very little to correct. However, these results also had some requirements. If developers used the tool outside the process it was often neglected and the benefit squandered. For a successful deployment it was necessary to integrate the tool in the process and put requirements on the developers to use them. While the tools did provide useful warnings it also reported false positives, warnings that are not a fault. In an experiment developers were asked to correct warnings with just the information provided by the tool. Due to the complexity of the code several developers damaged the system when trying to correct the false positives. In the worst cases the developers removed input validations due to the false positives opening the product to new vulnerabilities.

Since static code analysis tools analyze source code and detect measurable vulnerabilities in the product, it was made possible to determine how effective the tool is in these cases. By performing the same studies on several different projects these results should provide a general trend on how static code analysis tool affect the products security.

Research Question 3: How well do existing security engineering methods and tools integrate in an agile process?

Case studies were performed so that development teams could evaluate the different security engineering practices. From these studies some activities were immediately dismissed because they either:

- Go against the agile way of working.
- Are already covered by other activities or the process itself and were therefore not practically evaluated.
- The developers believe did not provide enough benefit and do not have agile replacement.

The majority of activities could be used in the agile process but required some modification to better integrate. The modified activities were either joined together with other activities for a new altered activity or they were moved from their intended
phase of development to a more suitable phase in the agile process that might not be ideal for its effectiveness. Lastly a small group of activates could be used as is in the agile process with any modification. In the end it was possible to create a secure agile process that covered most of the security concepts from established security engineering processes.

The initial literature and interview study relied on developer opinions. While this is a good guideline the outcome is not connected to any measurable result. Therefore the practical study allowed the developers to try the different Security Engineering concepts and experience them in a real environment. Because of this some results could be measured by the developers and they could base their answers on their own experience of using the security activity. By using developers from different projects and making sure that several developers used the same activities the results are more general and not specific for on project. To avoid author bias the developers were instructed to perform and evaluate their own activities and any feedback on the results or the success of an activity are all the developers’ options.

Final Thoughts

The contribution of this thesis is to understand how software development methods and tools can aid developers to create security software while working in an agile process. The thesis has examined what existing security engineering processes can integrate in an agile process (Q3). Then the thesis evaluates a security activity that could be used without modification to integrate in the process (Q2). The activity, static code analysis, was examined in several ongoing development projects to determine how it should be deployed, used and integrated in the process. Lastly a security engineering concept, risk analysis, was adapted to better fit the agile process (Q1). Providing a method of how future activities can be modified with the aid of action research to create a new activity that achieves the same goal.
Chapter 5. Discussion and Conclusions
A.1 Introduction

Software security has recently become an important business case for companies to protect information availability (Frühwirth 2009). Therefore, it is important to make security an integral part throughout the software development process which has been done through security touch-points introduced by Gary McGraw (McGraw 2006a). During the development of software, bugs and vulnerabilities are unintentionally introduced into the end product. The cost of removing these vulnerabilities increases the further the product has reached in its development cycle. It is therefore preferable to detect them as early as possible in the development process. Vulnerabilities are introduced into two different phases of development, namely architecture/design and implementation. Implementation vulnerabilities such as buffer overflows are added to the source while the developer writes it. For these types of vulnerabilities, static code analysis
tools can be used for early detection (Baca, Carlsson, and Lundberg 2008; Baca, Petersen, Carlsson, and Lundberg 2009). Design vulnerabilities on the other hand can be introduced into the product before there is any source code to examine. Early vulnerability prevention therefore has to focus in the architect of the product instead of the implementation, and hence security touch points have to be introduced early.

Different approaches have been introduced to detect threats/attacks on the system, namely risk analysis, and attack trees. Many if these methods are not specific for software development and some require UML diagrams or other information of the end product that might not be available (see e.g. (Howard and LeBlanc 2003)). This if often done with risk analysis and other threat modeling techniques. Attack tree are specifically proposed in the software development context and only require an understanding of the intended end product. Hence, they can be used as a security touch point early, even before implementation has been started. However, there are a few challenges related to the use of attack trees:

- From our experience of working with software security in industry we know that it can be difficult for non-security professionals to create an attack tree.

- The focus of attack trees and risk analysis is on identifying and prioritizing attacks, but not on the protection on the end-product (cf. (Buldas, Laud, Priisalu, Saarepera, and Willemson 2006; Moore, Ellison, and Linger 2001; Mauw and Oostdijk 2005)).

- The way of prioritizing the attacks is done by assigning estimated risks, which are dependent on many factors not known to the software developer (cf. (Buldas, Laud, Priisalu, Saarepera, and Willemson 2006)).

In response to these challenges we present a novel method to prioritize countermeasures to prevent security vulnerabilities, and by that focus the prioritization on the protection of the end-product. The Countermeasure Method for Software Security (CM-Sec) is an extension to attack trees. The method has the following features:

- It aids non security developers in identifying and prioritizing attacks and countermeasures by providing a process that guides them to the solution step by step. The introduction of why the system could be attacked and who could attack the system is used as a trigger to identify attacks and countermeasures.

- Instead of evaluating the risk of an attack for each individual attack ACM-Sec focuses on the prioritization of attacks through cumulative voting, i.e. a fixed number of points is distributed, attacks having a higher threat level receiving more points.
The method has been applied on an open-source system by the authors to demonstrate its applicability. The first author has good knowledge of the system as he is participating in the open source project developing it. The application showed that a broad range of countermeasures could be identified and linked to attacks, and that the distance of the criticality of the countermeasures can be consistently determined.

The remainder of the paper is structured as follows: Section B.2 presents the background and related work. Thereafter, Section A.3 presents the CM-Sec method. The application of the method on an open source system is illustrated in Section A.4. Section D.6 discusses the application of the method and Section D.7 concludes the paper.

A.2 Background and Related Work

The related work focuses on approaches to identify and prioritize attacks on software systems early in development, i.e. before coding starts. The motivation for doing so is that the earlier problems in software development are discovered, the easier and less costly they are to fix (Damm, Lundberg, and Wohlin 2006).

A traditional way of identifying threats to a system would be a quantitative risk analysis like that described in Peltier (Peltier 2001). The first step in the risk analysis would be to enumerate all assets and include their values for the system and then determine the threats that might exist for these values. The second step is to estimate the probability that a threat will occur and the damage it could cause. By combining these two values you get a priorities list of the top risks for the system. Determining what an asset is for a software product can be problematic, also immediately identifying threats to that assets often requires security expertise.

Attack trees were introduced by Bruce Schneier (Schneier 1999) and are structured by stating (1) the goal of the attack (e.g. obtaining a key); (2) what type of attack to be used (e.g. obtaining a private key from a user); and (3) how to implement the attack (e.g. break into the users machine and read the key from the disc) (Viega and McGraw 2002b). The implementations of the attack can be connected to conditions, e.g. one could choose either one of two or more implementation (OR) or several implementations are to be done together (Viega and McGraw 2002b). Attack trees have been applied on examples of software systems (cf. (Saini, Duan, and Paruchuri 2008)). After identifying the attacks should be evaluated based on their risk, i.e. the likelihood of their occurrence multiplied with the damage done (Buldas, Laud, Priisalu, Saarepera, and Willemsen 2006; Moore, Ellison, and Linger 2001; Mauw and Oostdijk 2005). However, Buldas et al. (Buldas, Laud, Priisalu, Saarepera, and Willemsen 2006) point out that the damage could be determined, but the risk of the occurrence is hard to estimate. Therefore, they propose to model the probability of the attack as a game for
the attacker taking the following parameters into account: gains for the attacker, costs of the attack for the attacker, success probability, probability of getting caught, and penalty of getting caught. A potential drawback of the proposal by Buldas et al. is that the software developers need to have a good understanding of the motivations of the attacker. Furthermore, attackers are different based on each individual’s attitude towards risk (see (Hederstierna 1981) for the distinction between risk averse vs. risk loving).

The related work shows that the solutions for addressing security issues prior to implementation have been focusing on the identification and prioritization of attacks and threats. However, from a software development perspective it is of interest to know which actions should be taken to avoid the occurrence of attacks, i.e. the prioritization of countermeasures. In order to prioritize the countermeasures we (1) propose an extension to attack trees; and (2) provide a process of how to arrive at a prioritization of countermeasures.

A.3 Countermeasure Method for Software Security (CM-Sec)

The method consists of two parts, the first part is an extension of attack trees. The second is a process for conducting the analysis to arrive at the prioritized list of countermeasures. For the second part a tool has been implemented which supports practitioners in conducting the prioritization.

A.3.1 Countermeasure Graphs: An Extension to Attack Trees

Three extensions are made to attack trees, which only focus on attacks and describe conditions under which the attacks can occur. The extensions are made explicit in the meta model shown in Figure B.1. The model consists of goals (why to attack), actors (who attacks), attacks (how to attack), and countermeasures (how to avoid attack), each described as a comment in the Figure.

The first extension is the relationship between goals, actors, and attacks. In attack trees there exists a 1..* relationship between the classes as they are trees, i.e. one goal is related to several attacks while one attack is only related to one goal. However, in practice an attack could be executed by several actors, or an actor could pursue more than one goal. Hence, these relationships were extended to *..*.

The second extension is the inclusion of priorities assigned to goals, actors, attacks, and countermeasures. The priorities are an integer number assigned by the per-
son conducting the prioritization. The prioritization method is further explained when presenting the process of creating the countermeasure graph (see Section B.3.1).

The third extension is the inclusion of countermeasures which are actions by the developers to avoid vulnerabilities to be introduced into the software product in the first place. Hence, from an implementation perspective it is important to prioritize the countermeasures which is done by considering the priority of attacks, as well effort to realize the countermeasure. With the inclusion of countermeasures the protection of the end-product is supported.

![Diagram of Metamodel of the Extension to Attack Trees](image)

**Figure A.1: Metamodel of the Extension to Attack Trees**

An example of the countermeasure graph is shown in Figure B.2. The nodes of the graph show the goals, actors, attacks, and countermeasures. The edges connect:

- Goals to agents if the agent pursues the goal.
- Agents to attacks if the agent is likely to be able to execute the attack.
- Attacks to countermeasures if the countermeasure is able to prevent the attack.

Furthermore, the priorities are assigned to each of the nodes. As can be seen agent two is higher prioritized than agent one, meaning that the agent is more likely to execute the attack and hence is a higher threat to the system. The agent has two attacks where attack one is prioritized higher as it does more damage to the system. The figure also
Chapter A. Prioritizing Countermeasures through the Countermeasure Method for Software Security (CM-Sec)

shows the prioritization of the countermeasures, countermeasure two being the most efficient in preventing the attack. It is important to observe that the prioritization does not only allow to prioritize the order of attacks and countermeasures, but also shows the distance between them.

A.3.2 Process

Identify Goals, Actors, Attacks and Countermeasures

The purpose of the countermeasure graph is to determine what security countermeasures would be useful to include in the product and at the same time identify what countermeasures already exist. At the same time we determine what the greatest threat to the product is and how well we prevent them. To achieve this we need to discover what attacks can be made and how they are stopped, with traditional threat analysis.
or attack trees this process is direct and tries to immediately identify attacks. We instead divide the work in four distinct steps and then use hierarchical cumulative voting (Berander and Svahnberg 2009a) to calculate the impact of every step.

**Goals:** The first step is to understand why anyone would attack the product and what their gain would be. This is often the same as the intended usage of the product and is used as a guideline to see the big picture.

**Agents:** Thereafter the Agents are identified. Agents are users, roles and software that interact with the products. Preferably the entire range of possible users should be presented in this step, form end users to administrators and outside software that interact with the product.

**Attacks:** Combining the Agents with a Goal we then look for Attacks. This step of the threat analysis is made easier because we have a more clear idea who and why the attack would accrue. We do not focus on if the attack would work, only if it is a possible route for the Agent to take to achieve his desired Goal.

**Countermeasures:** The final step is identifying what Countermeasures could be used to prevent an attack. A countermeasures can prevent one or more attacks.

Having identified the goals, agents, attacks, and countermeasures we are interested in which countermeasures are most effective.

### Prioritization of Countermeasures

The attack graph consists of sets of goals, agents, attacks, and countermeasures. In the following we present the prioritization questions asked to the developers, and also provide further explanation of the voting on each hierarchy.

- **Goals:** To what degree would the achievement of the goal damage the system from either a cost or stability perspective? These are at first general goals that the different Agents might be interested in achieving. While the first iteration of the attack tree has general goals further iteration will focus the goals on specific requirement and features and are abuse cases based on them. The Goals are also connected to what Agents would be interested in them. Voting on the Goals is focused on the damage that specific goal would sustain on the product. Depending on the goal the damage could be economical or system stability.

- **Agent:** How large is the threat of the agent for the system? With the Agents we determine how threatening the different roles are to the product. Votes are based on how ”scared” the product should be from that Agent. As an example a server product would consider end user more threatening then system administrators while privacy software might be the opposite.
Chapter A. Prioritizing Countermeasures through the Countermeasure Method for Software Security (CM-Sec)

- Attack: How likely is the success of the attack for the agent it belongs to? The attack voting is per Agent and determines how likely it is for that Agent to succeed with the attack.

- Countermeasures: How efficient is the countermeasure in preventing an attack? When voting on Countermeasures the focus is on their ability to prevent the attack compared to each other. In some cases two Countermeasures might be equally effective, but one of them might aid in preventing other attacks as well. The attack graphs would then put higher priority on that Countermeasure.

Each of the sets is prioritized according to the rules of hierarchical cumulative voting (HCV) (Berander and Svahnberg 2009a). Cumulative voting in itself has the benefit of (1) that it is simple to do, and (2) the distance of importance between two items is visible. The voting is done by providing a number of points (e.g. 100 dollars) and then distributing them between a set of items. In HCV the prioritization is done on each level of the hierarchy (in this case goals, agents, attacks, and countermeasures).

Figure B.3 shows the principle of HCV. The prioritization is done on all level, in this case on level 0 (L0) no prioritization is necessary as there is only one node on that level. On level H1 three nodes compete against each other for the points, i.e. nodes two, three, and four. On the lowest level H2 nodes within a group (H2.1, H2.2, and H2.3) compete. When the votes are completed the value of a node is calculated by multiplying the path of its parent nodes. For example, to know the value of node 11 we calculate the product of points assigned to node 11, 4, and 1. As there are different number of nodes within the groups the votes have to be adjusted by the number by the number of nodes in each group (cf. (Berander and Svahnberg 2009a)).

![Figure A.3: Cumulative Hierarchical Voting](image_url)

In the case of our analysis an n to n relationship exist between nodes on different levels of the hierarchy. If an attack is related to several agents then this should raise the
prioritized value of the attack. As an example we calculate the value of countermeasure
two in Figure B.2. First, we calculate the value for counter two ($C_2$) related to attack
2 (remaining zeros removed):

$$C_2 = 500(normalized) \times 75 \times 320 \times 250 = 30000 \quad (A.1)$$

Thereafter, counter two is calculated for attack 1 ($C_1$), here observe that the values
for agent one (150) and two (320) are added as both have an interest in doing the attack,
and hence the threat level increases:

$$C_1 = 750 \times 220 \times (150 + 320) \times 250 = 193875 \quad (A.2)$$

Overall, the value of the attack is the sum of $C_1 + C_2 = 223875$. The two are
added because the countermeasure prevents two attacks and hence is more effective.
Knowing the prioritization of the countermeasures with regard to effectiveness allows
to combine them with costs in a matrix. The matrix shows different areas for the
interaction of effectiveness and cost that can be used as a support in deciding which
countermeasures to focus on. The bottom-right area contains countermeasures that are
effective and have low costs. Hence, these should be implemented first. The top-left
area contains countermeasures that are ineffective and at the same time costly, hence
they should be avoided. In the middle of the two areas the borderline-cases are shown,
which could be implemented if there are enough resources available.

### A.4 Application

To demonstrate the method we applied it on an open source system, called Code 43.

#### A.4.1 System Description and Development Environment

We examined an open source product, which is an online first person shooter game.
An overview of the architecture of the system is provided in Figure A.5. In the cen-
ter of the system is the master server providing server lists to clients, storing clients
authentication information (e.g. authorization data), and client statistical data showing
the performance of the players. Connected to the master server are the servers hosting
the games. The clients are the ones logging into the servers that they receive through
the server list provided by the master server. Because servers are setup by users they
can not be trusted and information from them can be corrupt. Only the master server
is in the trusted zone, all other actors in the system are outside this zone (servers are
untrusted and clients are considered unsecured).
The game is a complex product consisting of 400,000 lines of code not including blanks or comments. This project has matured from other products, i.e. it is reusing source code. In total there has been 26 major active developers.

A.4.2 Result of Applying ACM-Sec

In this example we have identified four Agents that interact with the system. They have five distinct goals that can be archived via six attacks. From these six attacks we devised eleven countermeasures that would in different effectiveness prevent one or several attacks. An overview of the countermeasure graph is shown in Figure A.6. Inside the nodes the value assigned during the prioritization is shown. All values have been normalized according to the approach illustrated in Section B.3.1. If there is more than one number in a node then this is for the different ages (see, for example, nodes ask for password and fake statistics).

In the following we provide the details for the goals, agents, attacks and countermeasures. This includes a detailed description of each of them.

Table A.1 provides an overview of the goals attackers might have. The majority
of the goals are focused on cheating, like alter statistics and cheat on server. The
motivation is either to steal an edge (steal an account), to gain an edge (client-site cheating), or to edit and modify ones edge (alter statistics). Thus, its primarily about the gaming experience.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1: Steal user accounts</td>
<td>All clients have a private account that stores their statistics and aliases. Other users and server operative could steal their clients’ identities.</td>
</tr>
<tr>
<td>G2: Compromise master</td>
<td>The master server is the only trusted source in the network architect. It is therefore a lucrative target for malicious users.</td>
</tr>
<tr>
<td>G3: Alter statistics</td>
<td>Clients compete with each other. All users are therefore interested in faking their success and cheat at their rankings.</td>
</tr>
<tr>
<td>G4: Cheat on server</td>
<td>Clients compete with each other. Client might try to alter the game rules to their benefit to gain an edge against other clients.</td>
</tr>
<tr>
<td>G5: Create fake server</td>
<td>Clients receive server lists from the trusted master. Servers that want more clients might create fake servers that all point to the same servers. Therefore showing up several times in the server list and creating a larger exposure to clients. This is unfair to other servers.</td>
</tr>
</tbody>
</table>

As can be seen in Figure A.6 the players are the agent with the highest threat, the reason being that the goals are all related to the gaming experience. Server operators also can have an interest as sometimes they are players themselves or can be influenced by gamers (e.g. due to ties to other gamers).

<table>
<thead>
<tr>
<th>Agents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1: Players</td>
<td>These are clients and the end users of the system. They are also the least trusted source and present the greatest threat.</td>
</tr>
<tr>
<td>A2: Server Operators</td>
<td>Any client can create and add servers to the system. As such the servers can not be trusted and server operative can have the same motives as players.</td>
</tr>
<tr>
<td>A3: Servers</td>
<td>Being open source software the server can be altered to behave differently than the master server expects. It is therefore also a threat to the system.</td>
</tr>
<tr>
<td>A4: Master Server Operators</td>
<td>The operator of the master server has total command over the system. While he/she is an agent he/she does not provide any threats as the master server operator already can do whatever he/she wants.</td>
</tr>
</tbody>
</table>

An overview of the attacks is provided in Table A.3. The attacks focus on either stealing other users accounts or cheating to improve ones own statistics. In both cases the threat does not only come from client but also from servers that are run by other clients. The administrators of these servers might have the same interest in other clients’ account and statistics, just as other clients might.
Table A.3: Attacks

<table>
<thead>
<tr>
<th>Attack</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>At1: Ask for the password</td>
<td>Social engineering attack where either other clients, server operators or servers send fake login request to other users in an attempt to get the clients password.</td>
</tr>
<tr>
<td>At2: Eavesdrop the password</td>
<td>Servers can intercept the clients’ login information as it passes thought the server to the master server. Servers also need to know that the client as passed the login procedure.</td>
</tr>
<tr>
<td>At3: Fake statistics</td>
<td>Clients and server can send any statistics to the master server and thus increasing their own ranking.</td>
</tr>
<tr>
<td>At4: AutoAim / Wallhacks</td>
<td>There exists several client side cheats where players use unauthorized software to gain an edge against their competitors.</td>
</tr>
<tr>
<td>At5: Malicious statistical data</td>
<td>A large source of user input to the master servers comes from statistical data. As such it is a big threat and way in for attackers. The attacks can vary from buffer overflows to injection attacks on the master servers database or webpage that presents the statistical data.</td>
</tr>
<tr>
<td>At6: Authenticate fake server</td>
<td>Any server can authenticate to the master server. They therefore can create multiple entries of their server in the master server list.</td>
</tr>
</tbody>
</table>

The countermeasures which are the main outcome of this analysis are shown in Table A.4. As can be seen in Figure A.6 one countermeasure is able to prevent several attacks, which raises its value in the prioritization (see, for example, C2).

After having prioritized the effectiveness of the countermeasures using our prioritization approach introduced in Section B.3.1 the countermeasures are combined with cost. Cost is the estimated effort required to implement them. The result is shown in Figure A.7.
Table A.4: Countermeasures

<table>
<thead>
<tr>
<th>Agents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1: Single logon</td>
<td>Have a single point of login for the user during game start. With a specific login window that can not be replicated or requested by servers or other users.</td>
</tr>
<tr>
<td>C3: Avoid servers</td>
<td>Use direct client and master server communication for client authentication. However, servers still need verification from master that the client has authenticated.</td>
</tr>
<tr>
<td>C4: Encrypt</td>
<td>Encrypt the password with a master server public key before sending it. Would require a sequence number or timestamp to prevent replay attacks.</td>
</tr>
<tr>
<td>C5: Several sources</td>
<td>Verify statistical data by comparing it from several sources. From both client in the game and the server hosting the game.</td>
</tr>
<tr>
<td>C6: Master spies</td>
<td>User random samples and make the master join servers to verify that the servers are not duplicates and that the statistical information is correct.</td>
</tr>
<tr>
<td>C7: Limit info.</td>
<td>Limit the information sent to the client so automated processes can not aid the player unfairly. For example do not send other players location unless they are within the clients’ field of view.</td>
</tr>
<tr>
<td>C8: Detection algorithm</td>
<td>Nonhuman actions can be detected by the server by analyzing the clients behavior and comparing it with normal dataset.</td>
</tr>
<tr>
<td>C9: Input sanitization</td>
<td>Cleaning all input strings and removing any harmful characters.</td>
</tr>
<tr>
<td>C10: Input policy</td>
<td>Having a strict input protocol that drops any incorrect input before it is processed for storage.</td>
</tr>
<tr>
<td>C11: IP and Port</td>
<td>Verify unique servers by examining both IP address and the destination Port. The same server can however run several games on different ports.</td>
</tr>
</tbody>
</table>

From this the following interpretations can be made: Countermeasures C2, C5, and C6 should be implemented as they are in the zone of countermeasures that are effective and have low cost. As can be seen in Figure A.6 C2 affects the same attack as C3 and C4. Hence, the implementation of C2 would reduce the effectiveness of these two countermeasures. Countermeasure C8 is very costly to implemented, but is highly effective. As often project resources are limited this countermeasure should be taken into consideration for future releases, i.e. it should be handled as a requirement instead of a quick fix. Countermeasure C11 should not be implemented as it has very low effectiveness, i.e. its implementation does not make a difference. The Figure also shows that C9 and C10 are equally effective, but have different costs. Hence, the analysis also allows prioritization on cost whenever countermeasures are close to each other with regard to effectiveness. In this example we did not find any countermeasures with high cost and low effectiveness. However, in a different context such countermeasures might very well be identified.
A.5 Discussion

A.5.1 Practical Implications:

Focus on End-product: We would like to stress that it is important to prioritize countermeasures as those are directly related to the end-product. That means they result in actions that could be taken early in the development process to avoid the introduction of vulnerabilities in the first place. Furthermore, it is important to mention that our method can be applied throughout the whole development life-cycle, independently of whether a new product is implemented, or an existing product is analyzed. When an existing product is analyzed it is important to take into account the countermeasures already implemented, this needs to be taken into account when doing the re-prioritization.

Problems of objectivity: Risk analysis and attack trees assume that developers have a good understanding of how an attacker would think as probabilities have to be estimated for many different factors (cf. (Buldas, Laud, Priisalu, Saarepera, and Willemson 2006)). Furthermore, it is hard for developers without security experience to identify attacks without guidance. Hence, the proposed methods addressed this problem in two different ways: First, the attack trees were extended by agents as this allows the developers to put themselves in the shoes of the attacker, knowing who the attacker might be. Secondly, a process is proposed that the developers can follow. Furthermore, the process focuses on comparing different attacks and countermeasures with each other. Having the comparison makes it easier to value the attacks and countermeasures.

Implementation of Triangulation: When conducting the prioritization we recommend that it should be done by several developers. The averages and variances of the points assigned provide an understanding of to what degree the developers agree on voting. If there is a large discrepancy the data will show the need for discussions and further investigations.

Tool support: To support the developers in the prioritization process we are developing a tool. The main feature of the tool is to (1) work in groups during the identification of the goals, agents, attacks, and countermeasures (shared canvas), (2) support in the prioritization. The support should be handled thorough sliders which allow easy re-prioritization and show the impact of the change of one prioritization in real-time. This makes the method particularly suited for agile development as changes in the system can be very easily prioritized for each iteration.

A.5.2 Research Implications:

The presented method focuses on countermeasures instead of only attacks and is novel with this regard. Hence, research needs to focus on testing the approach in an industrial
Chapter A. Prioritizing Countermeasures through the Countermeasure Method for Software Security (CM-Sec)

application. We plan to conduct case studies in industry to evaluate the method with empirical data.

A.6 Conclusion

This paper presented a novel method to identify and prioritize countermeasures to increase the security of software systems. The method provides an extension to attack trees, as well as a process for the identification and prioritization of the countermeasures. We applied the method on an open source system, the application showing that several countermeasures could be identified. Furthermore, an analysis of the quantitative results is presented, showing that the proposed method has a potential in guiding managers in choosing the most effective and cost-efficient countermeasures. In future work empirical evaluations of the proposed method in industry are needed.
Paper B

Countermeasure Graphs for Security Risk Assessment: An Action Research

Dejan Baca and Kai Petersen

B.1 Introduction

Risk and threat analysis are an important part of early vulnerability prevention. Risk analysis, threat models (such as (Peltier 2001) and (Verdon and McGraw 2004)) and abuse cases (McGraw 2006a) are different methods of identifying and preventing risks. We proposed a combination of the user perspective (misuse and abuse cases) with the threat and risk perspective (Peltier and attack trees) in previous work (countermeasure graphs) (Baca and Petersen 2010), as it is important to look at both of them. For example, when not considering a user group, we might miss their goals, and with that potential threats. Furthermore, the previous models were general and did not consider the specific needs of agile development processes with their nature of having small iterations and incomplete designs. Hence, we evaluated countermeasure graphs in that context. By doing that, we also address a research gap of using risk assessment in the context of agile software development, as no case studies evaluated the use of the previous approaches in that context. Because of the small iterations and incomplete
designs many of the traditional methods might not aid in risk assessment, as at their time of development systems were built by first making a complete design and gather the major part of all requirements prior to starting implementation.

In response to that challenge we introduced an approach that allows to easily update the risk-analysis. The approach has been introduced in Section B.3 (referred to as countermeasure graph). The countermeasure graph focuses on identifying and ranking protective measures against attacks product. By creating a model of the attack surface against the product it aids the developers in identifying attacks and ways of preventing them. Based on the need of industry to be able to work in an agile way being able to respond to changing customer requirements we introduced our approach in the company and evaluated it in different projects. Our proposition before evaluation was that the new approach (including tool support) is better suited to support agile risk assessment as (1) it stores traceability links between agents, threats ...; (2) the efficiency and effectiveness of a countermeasure in relation to the risks identified is automatically updated and allows to capture the effect of small changes immediately. As a reference for comparison to our approach we used the Peltier risk assessment approach (Peltier 2001).

The contribution of this chapter is the evaluation and iterative improvement of the approach presented in Chapter B.4.1. That is, we used action research with several industrial projects. Action research is characterized by taking an active role in implementing an intervention in practice (e.g. as a team member) and working iteratively by taking feedback and lessons learned into account. Our choice for action research as a research method was motivated by the need of incorporating feedback into the approach and to evaluate the improvement. Furthermore, the authors are employed at the company working with security related issues, which allowed to easily participate actively in ongoing projects at the company.

The remainder of the chapter is structured as follows: Section B.2 presents the related work of risk assessment approaches for software security. Section B.3 illustrates how the baseline approach (countermeasure graph) works. Thereafter, the evaluation is presented in Section B.4. The evaluation includes the presentation of the action research approach, and the results that were learned in each step of the evaluation.

### B.2 Related Work

The related work provides an overview of risk assessment approaches for software security, followed by a discussion of their limitations.

Peltier (Peltier 2004) defines a risk assessment process for information security. The process consists of the steps asset definition, threat determination, determine prob-
ability of occurrence of the threat, and assess impact of the threat, recommended controls, and result documentation. By combining probability and impact into a matrix and placing the threats into that matrix a decision support is given to determine for which threat an action has to be taken (referred as controls, or in this chapter referred to as countermeasures).

Schneier (Schneier 1999) introduced attack trees, which is a notation to describe a system with respect to security. The root node of the tree describes what should be achieved (e.g. the goal is to access system) and the child nodes describe attacks of how that could be done (e.g. learn password). A child nodes can also have further child nodes of how to realize that attack (e.g. bribe administrator). Furthermore, a distinction is made between OR nodes (either choose one attack or the other) as well as AND nodes (do both attacks together). Each node (attack) can have a variety of attributes, such as its cost, equipment needed, and possibility to conduct the attack. From these one can then e.g. derive the most likely and cheapest low-risk attack, cheap high-risk attacks, etc. These then can guide which countermeasures to implement.

Misuse cases (Sindre and Opdahl 2005) are defined based on UML use case definitions, misuse case corresponding to use case in UML, and actor corresponding to misuser. A misuse would correspond to an attack (in attack trees) or a threat (in Peltier’s method), such as flood system, get privileges, etc. Alternatives of how to attack the system can be expressed in the form of misuse case variants. However, from an empirical standpoint a study using controlled experiments has shown that attack trees are more effective than misuse cases, even though people had similar opinions about them (Opdahl and Sindre 2009).

Peltier’s approach as well as Schneier’s approach do not explicitly consider the stakeholders in their approach, which might lead to ignoring specific groups, hence missing their motivations, and with that their goals in attacking the system. The same also applies in Peltier who proposes to start from the system scope and then identifying the threats, but does not provide a guide of how to identify the threats. Misuse cases take the user explicitly into account, but does not contain an assessment for risk assessment. As a consequence, there is a need to combine the ideas from Schneier (Peltier 2004), Peltier (Peltier 2004), as well as Sindre and Opdahl (Sindre and Opdahl 2005) and take misusers, goals, attacks, as well as countermeasures into consideration. Furthermore, the prioritization of threats is done on different scale types independently of each other (e.g. value of 1 to n where 1 is unimportant and n is very important). However, it is of value to know by how much more important one goal/threat is than another. Consequently, approaches with trade-off are required, such as AHP (Berander and Svahnberg 2009b) or cumulative voting (Sawyer and MacRae 1962). AHP does a pairwise comparison and the stakeholder has to decide by how much one item is more important than the other. Cumulative voting (also called 100 dollar method) requires
the user to assign dollars to items depending on their importance. If one item is twice as important as another item, it shall receive twice as many dollars. AHP only allows for a very small number of comparisons, hence it is not feasible in most cases (Berander and Andrews 2005). Cumulative voting is also efficient when many items are to be prioritized (Berander and Svalnberg 2009b). Further details on the prioritization approach are provided in Section B.3. In consequence, we proposed a new approach (countermeasure graphs) that (1) combines the ideas mentioned above, and (2) allows for relative prioritization (Baca and Petersen 2010).

The above mentioned approaches are proposed as general risk assessment approaches, independently of any development methodology. Recently, Williams and Shipley (Williams, Meneely, and Shipley 2010a) proposed an approach that allows to assess the security risk associated with individual requirements. The approach is focused on project level and is a simple approach that uses the concepts of planning poker in agile teams. For risk assessment Peltier’s approach of combining risk and impact was used. The difference to countermeasure graphs is that they are not driven by individual requirements, but system level perspective. However, a change of requirements might e.g. add new malicious actors, and hence change the relative risk of other identified threats/attacks.

In the following section countermeasure graphs, which are the subject of evaluation in this paper, are further explained. In particular, we explain the baseline with which we started the evaluation at the company.

### B.3 Countermeasure Graph

We combined the ideas of misuse cases, agile attack trees, and Peltier by considering actors and connecting them to goals and attacks. That is, we consider the relationship between goals, actors, and attacks. In attack trees there exists a 1..* relationship between the classes as they are trees, i.e. one goal is related to several attacks while one attack is only related to one goal. However, in practice an attack could be executed by several actors, or an actor could pursue more than one goal. Hence, these relationships were extended to *..*.

Furthermore, we include priorities assigned to goals, actors, attacks, and countermeasures. The priorities are an integer number assigned by the person conducting the prioritization. The prioritization method is further explained when presenting the process of creating the countermeasure graph (see Section B.3.1).

We incorporate Peltier’s ideas by including countermeasures (in his words called conduct), which are actions to avoid vulnerabilities to be introduced into the software product in the first place. Hence, from an implementation perspective it is important...
to prioritize the countermeasures which is done by considering the priority of attacks, as well effort to realize the countermeasure. With the inclusion of countermeasures the protection of the end-product is supported.

Figure B.1: Metamodel of the Extension to Attack Trees

An example of the countermeasure graph is shown in Figure B.2. The nodes of the graph show the goals, actors, attacks, and countermeasures. The edges connect:

- Goals to agents if the agent pursues the goal.
- Agents to attacks if the agent is likely to be able to execute the attack.
- Attacks to countermeasures if the countermeasure is able to prevent the attack.

Furthermore, the priorities are assigned to each of the nodes. As can be seen agent two is higher prioritized than agent one, meaning that the agent is more likely to execute the attack and hence is a higher threat to the system. The agent has two attacks where attack one is prioritized higher as it does more damage to the system. The figure also shows the prioritization of the countermeasures, countermeasure two being the most efficient in preventing the attack. It is important to observe that the prioritization does not only allow to prioritize the order of attacks and countermeasures, but also shows the distance between them.
Chapter B. Countermeasure Graphs for Security Risk Assessment: An Action Research

Figure B.2: Example of a Countermeasure Graph

B.3.1 Process

Identify Goals, Actors, Attacks and Countermeasures

The purpose of the countermeasure graph is to determine what security countermeasures would be useful to include in the product and at the same time identify what countermeasures already exist. At the same time we determine what the greatest threat to the product is and how well we prevent them. To achieve this we need to discover what attacks can be made and how they are stopped, with traditional threat analysis or attack trees this process is direct and tries to immediately identify attacks. We instead divide the work into four distinct steps and then use hierarchical cumulative voting (Berander and Svahnberg 2009a) to calculate the impact of every step.

Goals: The first step is to understand why anyone would attack the product and what their gain would be. This is often the same as the intended usage of the product and is used as a guideline to see the big picture.
Agents: Thereafter the Agents are identified. Agents are users, roles and software that interact with the products. Preferably the entire range of possible users should be presented in this step, form end users to administrators and outside software that interact with the product.

Attacks: Combining the Agents with a Goal we then look for Attacks. This step of the threat analysis is made easier because we have a more clear idea who and why the attack would accrue. We do not focus on if the attack would work, only if it is a possible route for the Agent to take to achieve his desired Goal.

Countermeasures: The final step is identifying what Countermeasures could be used to prevent an attack. A countermeasures can prevent one or more attacks.

Having identified the goals, agents, attacks, and countermeasures we are interested in which countermeasures are most effective.

Prioritization of Countermeasures

The attack graph consists of sets of goals, agents, attacks, and countermeasures. In the following we present the prioritization questions asked to the developers, and also provide further explanation of the voting on each hierarchy.

- **Goals**: To what degree would the achievement of the goal damage the system from either a cost or stability perspective? These are at first general goals that the different Agents might be interested in achieving. While the first iteration of the attack tree has general goals further iteration will focus the goals on specific requirement and features and are abuse cases based on them. The Goals are also connected to what Agents would be interested in them. Voting on the Goals is focused on the damage that specific goal would sustain on the product. Depending on the goal the damage could be economical or system stability.

- **Agent**: How large is the threat of the agent for the system? With the Agents we determine how threatening the different roles are to the product. Votes are based on how "scared" the product should be from that Agent. As an example a server product would consider end user more threatening then system administrators while privacy software might be the opposite.

- **Attack**: How likely is the success of the attack for the agent it belongs to? The attack voting is per Agent and determines how likely it is for that Agent to succeeded with the attack.

- **Countermeasures**: How efficient is the countermeasure in preventing an attack? When voting on Countermeasures the focus is on their ability to prevent the attack.
compared to each other. In some cases two Countermeasures might be equally effective, but one of them might aid in preventing other attacks as well. The attack graphs would then put higher priority on that Countermeasure.

Each of the sets is prioritized according to the rules of hierarchical cumulative voting (HCV) (Berander and Svahnberg 2009a). Cumulative voting in itself has the benefit of (1) that it is simple to do, and (2) the distance of importance between two items is visible. The voting is done by providing a number of points (e.g. 100 dollars) and then distributing them between a set of items. In HCV the prioritization is done on each level of the hierarchy (in this case goals, agents, attacks, and countermeasures).

Figure B.3 shows the principle of HCV. The prioritization is done on all level, in this case on level 0 (L0) no prioritization is necessary as there is only one node on that level. On level H1 three nodes compete against each other for the points, i.e. nodes two, three, and four. On the lowest level H2 nodes within a group (H2.1, H2.2, and H2.3) compete. When the votes are completed the value of a node is calculated by multiplying the path of its parent nodes. For example, to know the value of node 11 we calculate the product of points assigned to node 11, 4, and 1. As there are different number of nodes within the groups the votes have to be adjusted by the number by the number of nodes in each group (cf. (Berander and Svahnberg 2009a)).

In the case of our analysis an n to n relationship exist between nodes on different levels of the hierarchy. If an attack is related to several agents then this should raise the prioritized value of the attack. As an example we calculate the value of countermeasure two in Figure B.2. First, we calculate the value for counter two \( C_2 \) related to attack 2 (remaining zeros removed):

\[
C_2 = 500 \text{(normalized)} \times 75 \times 320 \times 250 = 30000
\]
Thereafter, counter two is calculated for attack 1 ($C_2^1$), here observe that the values for agent one (150) and two (320) are added as both have an interest in doing the attack, and hence the threat level increases:

\[
C_2^2 = 750 \times 220 \times (150 + 320) \times 250 = 193875
\]  

(B.2)

Overall, the value of the attack is the sum of $C_2^1 + C_2^2 = 223875$. The two are added because the countermeasure prevents two attacks and hence is more effective. Knowing the prioritization of the countermeasures with regard to effectiveness allows to combine them with costs in a matrix. The matrix shows different areas for the interaction of effectiveness and cost that can be used as a support in deciding which countermeasures to focus on. The bottom-right area contains countermeasures that are effective and have low costs. Hence, these should be implemented first. The top-left area contains countermeasures that are ineffective and at the same time costly, hence they should be avoided. In the middle of the two areas the borderline-cases are shown, which could be implemented if there are enough resources available.

![Figure B.4: Combination of Effectiveness with Cost](image)

B.4 Evaluation

The evaluation is conducted in an iterative way using action research. In total we evaluated four different products in six steps to arrive at a new and improved version of countermeasure graphs.
B.4.1 Research Conduct

Research Context

Security and Risk Assessment Prior to Introduction: Prior to the introduction of our approach Ericsson conducted risk analysis as part of its development process. The risk analysis is modelled using Peltier risk analysis. Figure REF shows the guide the developers use to rank the threats they discover. The threats are rated with respect to likelihood of success and the severity of a successful attack. By multiplying these two values a risk rating is produced. Each rank has a specific explanation for the fulfilment of that rank. After the threats have been ranked the most sever are analyzed to determine what action can be taken to mitigate them. The conduct of this risk analysis is a requirement on all products at Ericsson and an essential part of the company’s strategy to prevent vulnerabilities and have early detection. It is recommended that the risk analysis is conducted at the beginning of a new software version.

Since the introduction of the risk analysis some projects in Ericsson have moved from a waterfall development process to system in system based agile process. The change of development process has meant that each development cycle has become shorter and changes to the original product plan are more frequent. These changes have affected the risk analysis method as it relies on a fixed product plan that can be evaluated early in the process and then stay fixed until release. Several other problems with the risk analysis method were observed so a study to determine the weakness of the method was conducted. (postmortem looking at old results, workshop asking developers about the risk analysis method).

Products: Four products have been investigated in a series of projects. In the following the systems are shortly characterized. All products are developed using Java Programming Language.

- Product A: Has 5 development teams with 5-7 developers per team. One team specialises in verification and validation. The product had been in development for several years prior to the study but risks analysis were only performed prior to major release version.

- Product B: Has 3 development teams with 5-6 developers per team. With on of the teams focusing on verification and validation. The product is a spin-off from product A and uses the same base framework.

- Product C: Has 3 development teams with 4-6 developers per team. All tasks are divided between the teams. Product development started during the study, meaning that no legacy risk analyses exists.
• Product D: Has 8 development teams with 6-8 developers per team. One team focuses only on requirements and one only on verification and validation. Has been in development for a year prior to the study.

Research Question

The research goal is to evaluate countermeasure graphs in the context of practitioners conducting security threat assessment in agile software development for large-scale systems with respect to their ability to identify security threats and countermeasures.

The following research question is in the center of analysis: How should countermeasure graphs be modified to be useful in agile software development? The answer to this question focuses on what is working well with a version of the approach and what is not working well. Hence, the research allows to provide explicit feedback on lessons learned while evolving the approach.

Action Research Design

Figure B.5 provides an overview of the four products under investigation. The vertical lines represent the timeline of investigation. The boxes represent projects run for the products, for which the risk assessment was conducted. Looking at the timeline, we distinguish between phases and steps. In phase 1/step 0 the baseline is evaluated on products A, B, and D. At that stage Peltier’s approach was used as a method. The assessment was done by the developers and the result was reported in a document outlining the results of the different steps shown in related work. In our analysis we focused on the outcome, i.e. the probability/impact matrix to determine the threats identified.

In phase 2/step 1 a prototype phase was initiated where pilot was conducted. Here the very initial version of countermeasure graphs (prototype) as presented in Section B.3 was evaluated on product A in two projects related to that product. Thereby, the author was the moderator supporting the team in conducting the risk assessment using countermeasure graphs. The outcome (countermeasure graph and matrix) were used for analysis, as well as the observations and the feedback we received from the team. The team consisted of several developers that fulfilled different roles in the project. At a minimum one developer for each of these roles were present, requirement engineer, programmer, tester. Often there were more then one developer per role and the risk analysis consisted of 5-6 developers including the author that was moderating risk analysis.

In phase 3 (Improvement) several improvements were made to countermeasure graphs. In step 1 countermeasure graphs were redesigned and evaluated in two projects,
Figure B.5: Action Research Approach
one related to product A and one related to product B. After the redesign we also saw a need for expansion, which was evaluated in three projects for products A, B, and C. Finally, having a good understanding of how to design countermeasure graphs we used the experiences gained to define a way of working. That way of working was then evaluated in four projects on three products (A, B, and D). During these risk analysis the developers themselves decided when to conduct the analysis and which developers should be part of it. The author was no longer moderating the analysis but was instead one of the developers providing expertise to the analysis. But since the author was still part of the analysis it was possible to observe problems and easily discuss improvements to the method and tool.

In phase 4 we achieved a matured version of our approach, which then was extended in step 5 by providing better support for cooperation. In this phase the authors were not present during the risk analyses and any feedback sent by the developers afterwards. However, the risk analyses continued after the study had concluded and is part of the development process of the projects.

### B.4.2 Results

#### Baseline (Peltier)

Product A, B and D had performed at a minimum one Peltier based risk analysis prior to the introduction of the Countermeasure Graph.

Using Peltier 13 known threats with the highest risk factor of 4 were identified. The threats focused on internal threats posed by the system administrator and hardware failures. Only system threats are reported in the results, no product specific features are mentioned.

Product B had 11 threats with the highest risk factor of 5. The threats were exclusively internal fraud by the system administrator and social engineering attacks from the outside.

Product D had 29 threats with the highest risk of 8. The threats were system level and focused on the user accountability and shared account access.

**Reflection:** The highest risk factor that one could actually find is 25 (i.e. multiplying 5 (probability) * 5 (impact)). Overall, this showed that the Peltier approach only identified low risk/impact threats. However, it was likely that the systems had more significant security issues given its nature (complexity, exposed to networks, many different users). One potential reason for not finding more significant threats was likely that Peltier did not seem to steer the people in identifying the most critical one. One potential reason for this is that when combining the ideas from the different approaches, a more comprehensive picture of the threat situation is given. This was done by piloting
the countermeasure graphs.

By observing the results of previous risk analysis and conducting a workshop with the developers to identify the problems with the method we created the following list of problems that need to be addressed:

**Only identified Low risk threats:** The results from previous risk analyses had often not yielded any risk with a higher rating then Low (5) and only in one occasion a Medium (8). A root cause analysis reviled that the identified risks focused on OS level threats instead if new functionality and that the developers that performed the risk analysis had problems to steer the risk analysis towards business logic threats instead of general system threats.

**Few business logic threats:** The risks analyses very seldom identified any business logic threats, e.g. threats that are related to features of the product. Instead most of the threats were based on the OS level or deployment of the system. The developers attributed this to the lack of expertise and the timing of the risk analysis.

**Risk analyses are performed at the end of the development cycle:** While risk analyses should be performed early to identify threats before they have been implemented, this was seldom the case in the examined products. The main reason for the late risk analysis was because it was not sure what requirements would be implemented. The team responsible for the risk analysis did not know if the requirements would be implemented or how. So the risk analyses instead performed before a major release on the already implemented features.

**Pilot**

For the first risk analysis during the pilot we used post-it notes instead of a tool. The idea was to use the method in an as agile friendly way as possible; depending on developer interaction before tool usage. However, as the risk model grew it because time consuming to calculate the risk of a threat or the effectiveness of a countermeasure. Combined with the need to share and reuse the risk model it was descried that a tool was needed. Therefore a prototype was created and a new risk analysis was conducted on product A.

As these two risk analysis were the first instance with the new model the focus was on adding the existing knowledge to the new model and to create a base line for further analysis. As such the greatest focus was put on identifying all possible Agents that interact with the product and any threatening Goals they might have.

The first two risk analysis identified 23 threats to the product. 7 of these threats were already handled by the system and no new countermeasures were required. For the remaining threats 37 countermeasures were identified, several of them prevent the same threats in different degrees. As such there were many more proposed counter-
measures then attacks. Each countermeasure was also weighed on its effectiveness to prevent an attack and its cost to implement. Because of this it was possible to generate a list of the most important countermeasures.

The top identified threats were:

1. The lack of product security testing, test cases that verify input validation, look for race conditions and so on
2. Internal fraud by the system administrator and the lack of accountability
3. Social engineering and user based attacks
4. Weak file system access control

The top prioritized countermeasures based on their effectiveness compared to the cost of implementing them:

1. Node hardening; securing the file system and accountability between different system users
2. Using static code analysis tools and fuzzy testing
3. Creating a mandatory input validation class that all input has to use
4. Using SE Linux

Redesign

For the second phase of our action research we looked at how to improve the method and to do this we used developer feedback from the risk analysis sessions. For our first improvement attempt we have introduced the tool to a second product and allowed them to reuse the previous risk analysis. Since product A and B share the same base platform it was very easy to reuse the model, only minor changes to the Agents were required. So for both products the base line was already present and the risk analysis could focus on new implementations instead of legacy functionality.

The main improvement to the model was to decouple the Agent and Goal. In the initial model a Goal was always connected to an Agent. But during this interaction of risk analyses there were scenarios were then strict Goal-Agent-Attack connection was not always true, in some cases an Agent might have Goals that were not connected to an Attack at all. It was therefore necessary to alter the model slightly and connect Goals and Agents to Attack instead of just Agents as the initial model had intended.
The change did not affect any weighting or require any of the previous risk analysis, as they did not have any attacks that required the specific case.

The main focus of these risk analysis was the authorization and authentication system. 11 new threats were identified and 13 new countermeasures were proposed. It was also the first time a suggested countermeasure created a new threat. In this case a suggested countermeasure against a Denial of Service attack meant that an ip address would be blocked, this made it possible for attackers to force the system to block ip addresses and thus creating a new Denial of Service attack.

The countermeasures from these risk analysis often solved the same problem. Because of this the developers could experiment within the tool and see what happened with the countermeasure prioritization list if they would implement one of the countermeasures. This aided the developers in deciding what countermeasure should be implemented and in what order. As an example, the third highest recommended countermeasure in the list would solve several of the other threats but at the same it would be costly do to, therefore lowering it in the list.

The top identified threats were:

1. Brute forcing a user’s password would lock that user after 3 failed attempts. This made it possible for outsiders to lock any user.
2. User can set any simple password they wish
3. Because of faulty salt implementation leaked passwords can be decrypted more easily then they should.

The top prioritized countermeasures based on their effectiveness compared to the cost of implementing them:

1. Place a delay on login attempts from the same ip instead of locking the user.
2. Correct the salt implementation
3. Use third party authentication system

Expansion

With the base method working as intended and producing requirement based risk. We instead looked at how to improve the tool and the feedback we received about working with such a large model. By the end of this risk analysis iteration the different models consisted of 10+ Goals, 10+ Agents, 30+ Attacks and 40+ Countermeasures. The old data is still useful and need in both the analysis and the calculation for prioritization but at the same time it was cluttering the tool.
Developers therefore recommend that the attack can be group together and that they can be placed as prevented. By placing an attack as implemented it is no longer used in the calculation for its countermeasures. Therefore if an implemented countermeasure has completely prevented an attack the risk analysis can now disregard the countermeasures that got a higher rating because they would have prevented the high risk attack. At the same time the any countermeasure that was only protecting against that attack could be hidden from the graph and free up screen space. Attack groups were also implemented so that the developers could hide a group and all the Attacks and Countermeasures of those Attacks would hide and free up screen space.

In this iteration a third product has joined the study. This new product does not share and base platform or framework as product A and B does. But it is still a telecom, server/client system using Java EE technology. As such it was still possible for the developers to reuse most of the Goals, Agents and some of the Attacks of previous risk analyses, even if they were done for a different product. Therefore, the startup time for product C was considerably shorter then it was for product A. The first risk analysis with product C therefore immediately produced features based risks instead of just system level threats.

The threats from this iteration focused on the new multivendor support for product A and B while product C that was a completely new product had several risk in its base requirements. All the newly identified risks were based on the product features instead of system level threats.

The top identified threats were:

1. Stealing other users programs and configurations by using their publicly known identification string
2. Resetting other users device by spoofing a reset request with their Mac address
3. Altering the synchronization log so the system would think the cluster is out of sync and perform unauthorized operations
4. Alter another vendors db tmp files and affect their user database

The top prioritized countermeasures based on their effectiveness compared to the cost of implementing them:

1. Using private/public keys to identify and verify user devices
2. Sandboxing and file system separation
3. Deploy all devices with system public key for integrity checks
Ways of Working

In this iteration the fourth and final product has joined the study. Because the product had already conducted several major risk analyses it first needed to move the results from the Peltier analysis to the tool. After that initial, startup analysis it also produced features based risks instead of the general system level threats.

With the inclusion of several projects it became necessary to consolidate and determine exactly what a Goal, Agent, Attack and Countermeasure was. It was therefore recommended by the developers to include the description of the components in the tools and to create a guide that the developers can follow when conducting the risk analysis.

Since product D was a relatively new product and had high priority, it had a large amount of developers working with it. There was an interest in dividing the work of the analyses and not only focusing on a single analysis per sprint or release. Instead the general Goals, Agents and Attacks were performed in the beginning of a new development cycle while the different development teams would take over the risk analysis and expand the model with new Attacks and Countermeasures based on the features they were implementing in that sprint. The tool and method supported this way of working and it was encouraged to the other products but they did not see the need since they had so few teams.

Another recommendation by the developers was to use the data in the model to create a metric of the current security state of the product. By comparing what attacks have been prevented compared to the weighting of those that have not been prevented it is possible to get a security metric based on the risks in the product.

Several new threats were identified and with the introduction of a new product some severe system level threats were discovered as well.

The top identified threats were:

1. Stealing admin and user credentials though a unencrypted network channel
2. Reading private information on remote transferred log files
3. Retrieving other user private information though weak transaction history verification
4. Several roles use shared accounts to perform different tasks

The top prioritized countermeasures based on their effectiveness compared to the cost of implementing them:

1. SSL encrypt all connections
2. Create annotation type for sensitive data so it cannot be logged
3. Audit log operations and remove sensitive data
4. Create guides for SSH tunneling of vulnerable connections if internal fraud is a threat

**Mature**

At the end of the study each product continued to use the Countermeasure Graph method as part of its development process. While the method itself has become stable there have been some improvements in the way of working with the tool.

Since it was easy for product to share and reuse some parts of each other’s risk analysis it was decided to have a general repository that the products could use to share and safely store their risks analysis. Since the tools uses a simple xml format to store the risk analysis it was possible to generate change logs between the risk analyses and verify that they have been performed at each release. It also made it possible to tag the results the same way the source code is so that a risk analysis is associated with a certain source code release.

Because of the size and multi-site development of some of the products it was necessary to add network support to the tool. With the shared screen, result repository and the iterative way of working with the tool it was possible to perform risk analysis with teams that were not located in the same place.

The threats identified by the method have also focused on the new features instead rediscovering the same system threats that still are not important enough to correct.

**B.5 Countermeasure Graph**

After a few iterations the model started to stabilize and fewer changes were done to the method instead and the changes instead happened to the tools layout and the way developers user the method. Figure B.6 shows how the old metamodel, Figure B.1, has evolved over time to the new model.

The nodes of the graph show the goals, actors, attacks, and countermeasures. The edges connect:

- Goals to Attacks if the purpose of the attack is the goal.
- Agents to Attacks if the agent is likely to be able to execute the attack.
- Attacks to countermeasures if the countermeasure is able to prevent the attack.
Chapter B. Countermeasure Graphs for Security Risk Assessment: An Action Research

Figure B.6: Metamodel of the Countermeasure Graph

Goals have an extra monetary value that can be used to create an extra prioritization variable. Some developers wanted to divide the damage an goal could do to a system and the possible monetary losses.

Attacks can be grouped so it is possible to focus in a certain type of attacks. If an attack was prevented it was made possible to deactivate it so that non-implemented countermeasures would not be prioritized if they protected against an already prevented attack.

The finished tool is shown in Figure B.7. The values bellow the Goal and Agent names are their weighting compared to each other. The values in the arrows show the weighting for that particular edge.

B.6 Discussion

An risk analysis is only as useful as its results. The major problem with the legacy analyses we examined was the lack of interesting findings. The developers performing the risk analysis reported that they had problems identifying threats and that they did not always know how or what would be implemented at the time of the analysis. Some developers tried solving this by performing the risk analysis after a release, but then
Figure B.7: Open Source tool that implements the Countermeasure Graph

they lost the entire benefit of detecting threats before implementing the design. With the Countermeasure Graph we have created a method that solves the problem, the main benefit was achieved by dividing the task and making it iterative so it can easily be reused. While the method and tool provide several other improvements to traditional risk analysis the main contribution is providing the developers with the possibility to focus the risk analysis on the new features and ignore the legacy threats that are already
in the model. By having the old threats in the calculations and reusing Goals and Agents it was easier for developers to focus and create threats for the new features instead of trying to identify new threats from nothing.

In this study we examined a security concept, risk analysis, that had work well with the company before but did not produce sufficient results after the projects switched from long term waterfall development with 1-1.5 years of development between release to a quick agile development process that had 3 weeks development iterations. To investigate and create a solution we performed action research were the author first examined the problem by evaluating the results and then proposing a solution. Because the solution might be faulty it was iterated though several risk analysis were the author received feedback and observed how the method was used. By solving the problem in a iterative way with the developers the solution was easily accepted by the developers and it continued to be used after the study. To minimize the risk that the solution would be to specific to the one case, we used several different projects to create a generalisable solution.

B.7 Conclusion

We have investigated how developers use risks analysis in an agile development process. The developers reported that they were not happy with the results of the risk analysis or the way the analysis was conducted. The short development cycles and uncertainty of the final design made it hard for them to conduct a useful risk analysis. In an attempt to solve the problem we conducted action research in unisons with the developers to prototype and improve a new risk analysis method and tool. We improved the tool in a iterative way with the aid and feedback from the developers that conducted the risk analysis as part of their development process. The created risk analysis method divides the work and makes it possible to reuse the results. By first identifying roles that interact with the system and goals those roles might be interested to achieve it be easier for the developers to create new attacks. If a new features was added to the product the developers could re-examine the existing Goals and Agents and see if any new attacks are possible with the new feature. Also by focusing on the Countermeasure, how the products prevents the Attack, it was more important to know if the attack is prevented then if it is possible to do. By listing Countermeasures the method also created a prioritized list the developers could immediately use in their planning of the short development cycle. The prioritization was done using hierarchical cumulative voting so that the developers were comparing the threats against each other instead of trying to place them on a scale. After several projects had joined the study a matured method and tool emerged that was generalisable enough to work for all of the project
and continued to be used after the study ended.
C.1 Introduction

In recent years software companies have concerned themselves with security related threats connected to their products source code. This is related to customers demanding high security in software products and that software products are used in networks where software vulnerabilities can be exploited (e.g., for intrusion) (Hunter et al. 2007). With these new threats, software companies have to improve the security of their products by preventing code vulnerabilities. In consequence, software companies now have to develop products with secure code and verify that legacy code is secure. This is a task that can slow ongoing development and is very expensive if the product has a large legacy code base.

To achieve this task, one has to identify the defects that cause source code vulnera-
abilities. Two main semi-automated alternatives are possible, namely static and dynamic
detection of faults. The dynamic detection of faults is done late in the process as it re-
quires the software system to be complete and executable. If the system fails, then the
location of the cause of the failure has to be identified. The static analysis does not
require that the system is executable. Instead, it can be run on incomplete parts of the
system (e.g., just a small part of the overall code base) and highlights the cause of a
possible failure. This leads to two main benefits: 1) Faults can be detected early in de-
velopment and thus are much less expensive to fix compared to late detection (Boehm
2002), and 2) the location of a failure does not have to be identified.

From a security perspective, the system is not able to determine which of the iden-
tified problems are security vulnerabilities. Instead, this judgment has to be made by
developers who use the output of the static code analysis tools. In order to select
the right people for the task, one has to know which experience is necessary to suc-
ceed in judging the output of the static code analysis tool. That is, if certain types of
experience (e.g., knowledge in security, knowledge in a programming language, and
experience with the static code analysis tool) matter, then they have to be considered
when assigning people to the task. However, several studies and articles have shown
that the SAT are indeed useful tools with lots of potential (Chess and McGraw 2004b)
(Ayewah et al. 2008) (Zitser et al. 2004a), but all previous studies have been done by
experts assuming that developers using the tool will have the same results. That is, no
empirical evidence is provided on the impact of experience on the results of static code
analysis usage.

To address this research gap the aim of this study is to determine the impact of
experience on the correct identification and classification of faults identified by a static
code analysis tool (SAT). The study has been conducted as an industry experiment with
34 developers. Furthermore, the perceived confidence in the answers from a single
developer has been asked for to control whether the answers can be considered as not
random.

We therefore want to answer if SATs are useful for average developers as a vul-
nerability detector, if developers with certain experience get better results and if the
developers can identify when they need aid in interpreting the SAT.

In section 1 the introduction and research problem is presented. Section 2 explains
the background and related work In Section 3 our research method is explained, fol-
lowed by the results in Section 4. In Section 5 we discuss the impact and possible
reasons of our result followed by our conclusions in Section 6.
C.2 Background and Related Work

The first static analysis tools (SAT) were simple program checkers (Viega et al. 2000b) that were no more sophisticated than a simple grep or find command. These simple SATs were soon followed by several commercial tools that for a variety of coding languages tried to achieve an automated source code review. These tools capture the most common faults in a particular coding language and help the developers to create more stable code. The tools represent a varying degree of sophistication and some utilize complicated techniques such as an abstract interpretation to achieve their goal of better accomplished automation (Hallem et al. 2003). A coding fault also represents a large quantity of software vulnerabilities where SATs have the capability to detect these faults. SATs are therefore good security touch points that can be introduced early in the development process (Mead and McGraw 2005). Experiences from industry show worse than expected results (Baca et al. 2008) and, even with a SAT, faults that should have been detected have slipped through the static analysis process.

A SAT is used early in development process to prevent fault slip through (Damm et al. 2006), i.e. catching the failures before testing is cheaper than finding them during testing and then having to correct and retest faults. The SAT is mostly automated. Thus, it is fast and inexpensive. Still there is a human factor when examining if a warning should be corrected; it is needed because the SATs have false positives (Hudepohl 2006). A warning is considered false if its statement is considered wrong or if the developer does not believe it needs correcting. The tool Coverity Prevent has a 20% false positive rate (Baca et al. 2008). The same study also suggests that about 5% of the tool generated warnings have been security faults that need correction.

C.3 Research Method

C.3.1 Variables

Two different types of variables are usually considered when conducting experiments, namely independent variables and dependent variables. The independent variables (or treatments) are what the researcher is controlling, and the dependent variables are measured outcomes. In this case, the variable experience is controlled. As outcome variables we consider the ratio of different fault types classified correctly by the developers (see Figure C.1).

The dependent variables are divided into three groups depending on the warning type.

- False positive An incorrectly reported warning.
Chapter C. Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?

Figure C.1: Independent and dependent variables.

- **True positive** A correctly identified fault.
- **Security** A correct warning that can propagate into one of the four security vulnerabilities that are explained in section 3.3.

Security and true positives are divided because the warnings can be interpreted and corrected in different ways depending on how they are reclassified.

The independent variables examine how experienced the developers are with: the coding language, software development, the product, software security and the SAT. The developers answers are rated in a four grade scale, from guessing to very confident. The experience data are then used for further analysis of different groups compared to their ability to classify a warning.

### C.3.2 Subjects

The software developers where randomly selected from the list of developers at Ericsson AB, a major telecom company, from different development sites (Sweden and India). The developers vary in experience and have or will use the SAT as part of their daily work. The study was voluntary and all the developers had the same briefing prior to the study. In total 34 developers answered the questionnaire. From the total group, 15 had experience in software security and 14 had used a SAT before the study. The developers were divided into three groups depending on their general software engineering experience. To determine the developers experience we examined how long they have worked in software development and how long they have experience of development in the products programming language. The low experiences group had 10 developers and less then 2 years development experience. The medium experience
group had 14 developers while the high experience group had the remaining 10 developers. The high experience group consisted of developers with more than 6 years of development experience. Specific knowledge groups consisting of security and SAT experience were also examined. Specific knowledge was not measured in time, instead developers answer a simple yes or no question.

C.3.3 Instrumentation

The instruments used in the experiment are written guidelines, forms used by the subjects to record their results, and the tools and systems used.

Written guidelines: For the experiment, each subject received one page explaining their task, including explanations of classification types (false positives, true positives, and security fault). Security faults are further explained in the instructions following the definitions of vulnerabilities (Krsul 1998a). This definition includes the coding faults that cause a Denial of Service attack, unauthorized disclosure, unauthorized destruction of data, and unauthorized modification of data.

Forms: The evaluations form was created from a random sample of warnings to create eight false positives, eight true positives and six security warnings. All developers then classified the same randomly generated sample. At the time of the investigation the product had an approximated ratio of 25 true positives and 8 false positive for every vulnerability, i.e. true positives are underrepresented and security warnings are highly overrepresented in the questionnaire due to the need of sufficient data points. The original classification was done by developers working on the product. The random sample was also examined further on to insure that the original classification was correct. In some cases the vulnerabilities were also practically verified for being accurate. The accuracy had been confirmed as follows: The warnings were taken from a mature product still under development. The over a year old source code in use had some known flaws both detected and undetected by the SAT. The results from the developers were compared to a correct template, i.e. a previous study that methodically examined and/or practically confirmed those warnings when doubt arose. Also, some warnings were confirmed by trouble reports from testers and end users. The sample contained 13 different checkers that looked for a specific type of fault. These checkers can be grouped based on the characteristics of the fault they detect. The following groups of warnings provided in the form are of security relevance:

- Memory faults (e.g., buffer overflows) lead to memory corruptions which can be exploited (e.g., for code injection) (Krsul 1998a).

- Null-pointer exceptions which allow to user-induced segmentation faults (Krsul 1998a).
Initialization checkers determine if allocated resources are used before they are properly initialized causing information leaks or segmentation faults (Krsul 1998a).

Race conditions allowing to lock or link system resources (e.g., processor or physical resources) (Krsul 1998a).

**Tools and systems used:** In this study we had chosen the SAT Coverity Prevent version 3.1. Coverity Prevent is one of the state of the art SAT and has been used in other studies to detect customer reported bugs and vulnerabilities (Emanuelsson and Nilsson 2008). This tool was chosen after an internal study showing that a low false positive rate was combined with a high rate of reported faults. Because the tool is automated and has a low false positive rate (10-20%), industry often uses the SAT as a drop in tool without any training. A more security focused SAT, Fortify, was not used in this study due to its higher false positive rate and other free-ware tools were too simplistic in an industrial setting.

### C.3.4 Operation

The study was conducted in two distinct steps. In step one the developers received an introduction to the study, where each developer received exactly the same information. This included a description of their task (e.g., explanation of the classification) and they were informed that they should do the task individually and should not talk to their peers about it. Furthermore, the forms have been explained and the written guidelines were handed out to the subjects. In the second step the developers conducted the experiment on their own. For the task one hour was recommended by the experimenters. After the individual experiment, the subjects sent back their filled in forms for analysis.

### C.3.5 Threats to Validity

One threat to the generalizability of the results of the study is the usage of one specific SAT. In order to mitigate this risk we choose a state of the art tool with a large industrial acceptance. The largest different between different tools is not the type of defects detected, but the number of false positives compared to false negatives. Because developers examine the output of the tool, the user friendliness can effect the results. But all previously examined SAT, open source tools, Klockworks and Fortify all presented there results in the same way. All SAT showed the results as warning text either in a report or directly in the source code. Because all top tools present the results in the same way, we can assume that the tools interface would not effect generalization. Another
To mitigate the risk of other factors than experience influencing the outcome of the study, the study was voluntary and stressed developers could ignore it. All developers got the same initial information and used the same interface to examine and report their finding. Developers did the study individually and had the same time frame. Thus, the risk of other factors affecting the dependent variables is reduced.

**C.3.6 Static analysis tool output examples**

SAT often provide output directly in the source code so that developers can read the warnings and at the same time see there own code and thereafter judge if it needs to be corrected. In the examples the numbers followed by a : is actual source code while rows without starting numbers are warnings from the tool. Example 1 is a simple off by one buffer overflow that does not cause any security problems but should be fixed by the developers.

**Listing C.1:** A simple off by one warning.

```
Event overrun-local: Overrun of static array "buf" of size 32 at position 32 with index variable "32"
30:  buf[32] = '\0';
```

But the tool can produce more complex warnings were the fault only occurs if the program follows a specific flow. These warnings are harder to understand and the developers have to read and understand every warning message. Example 2 has a more complex flow that if three conditions are met the program would return with an error but not de-allocate it used memory. This is made even worse because the operation is initiated from a user message and the three conditions are controlled by the user. In this case an outside user could use this vulnerability to use up all the systems memory and create a denial of service attack.

**Listing C.2:** More complex denial of service attack were three conditions have to be met before the attack is possible.

```
Event alloc_fn: Called allocation function "operator new (unsigned int)"
7859: daData = new CcnDdrSet(daTagNo);
```
Chapter C. Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?

At conditional (1): "StringTokenizer::hasMoreTokens() != 0" taking true path
7863: if (st.hasMoreTokens())
    7864: {

At conditional (2): "DicosString::compare(const char *) const != 0" taking true path
7867: if (daAmountString.compare("") != 0)
    7868: {

At conditional (3): "daAmount > 1000000" taking true path
7872: if (daAmount > MAX_DA_AMOUNT)
    7873: {

Event leaked_storage: Returned without freeing storage "daData"
7875: return false;

C.4 Data Analysis

In this experiment a group of 34 developers have individually classified 8 (272 in total) false positive, 8 (272 in total) true positives and 6 (204 in total) security warnings. In figures 2 to 6 the average chance rate is 33% shown in every graph for reference, but notice that the actual chance rate is 36% for false positives and true positives and 27% for security warnings due to the different number of items in each category. We calculate change based in the number of correct answers a developer would get if he assumed all warnings would be true. Figure C.2 shows how many percent of the warnings are correctly classified by the developers. In section 3.2 the three warning groups are explained. Only the true positives are identified better then chance, while both false positive and security warnings are within the chance rate of 27-36% (depending on whether the category contains 6 or 8 warnings).

In figure C.3 the first results are divided into groups depending on how confident the developers are in their classification of the warnings. In this division four groups are created, that is: 157 warnings are classified as very confident, 325 as confident, 175 as not confident and 97 as guessing. True positives behave almost as expected, i.e. less confident answers have fewer correct answers. False positive on the other hand behaved
Because Figure C.4 did not show that confidence improves the identification of security vulnerabilities we have excluded confidence from further analysis and all answers, from very confident to guessing, are used. In figure 4 the results are grouped by
the developers general experience. We get three groups with 10 developers in the low group, 14 in the medium and 10 in the high experience group. The results do not vary notably between the three groups. The security vulnerabilities also stay within the chance range.

Figure C.4: Classifications grouped by the developers general experience.

Figure C.5 examines the two most important specific experiences. Here, 15 developers with security experience are compared to 19 without. Furthermore, 14 developers with prior knowledge in SAT are compared to the remaining 20 developers with no knowledge in SAT. While the security group is better than the non security group in detecting vulnerabilities the results is only just better than chance. The group with SAT experience is the first group with substantial improvement in identifying the security vulnerabilities and the non SAT group has a worse security detection result.

The best security detection group is created by combining developers with both security experience and SAT experience. This group consists of only 7 developers while the remaining are 27 developers. The security (SEC) and SAT group correctly classified 67% of the security vulnerabilities which is an improvement to the SAT only group. Figure C.6 shows developers with both SAT and security experience compared to the developers with no specific or only one specific experience.
C.5 Discussion

When software tools are evaluated for industry or written about in articles the focus is almost always on the tools capacity. Static analysis tools have had several studies that have shown the tools ability to detect coding faults, failures and even security vulnerabilities. There are several different tools to choose from that have different behaviors
Chapter C. Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?

and characteristics. Different characteristics might be preferred depending on where in the company or development cycle the tool is deployed. If the static analysis tool is used by a security assurance team the soundness and capability of the tool might be more important than its false positive rate. But if the tool is intended to be used as an early fault detector on a daily basis, then the tool’s actual capacity might, due to inexperienced developers, be lower than its full potential.

In Figure C.2 we examined the average result of a developer in classifying warnings. Without creating any groups regarding experience, or removing any answers due to unconfident answers, we clearly see that true positives have been mostly classified in the correct way.

Dividing the answers based on the developers confidence in their classification does not show the expected results. Relative few answers are guessed and the majority of developers are confident in their classification. A positive judgment is expected in both, the security and false positive warnings. Instead the judged security warnings are randomized and the false positive result improved the more uncertain the developers are, i.e. the outcome improves with increased uncertainty. The difference in results indicates that developers do not correctly judge how correct their classification is. We therefore do not believe that developers can judge when they need assistance in classifying a SAT warning.

The general development experience did not have the expected impact on the number of correctly identified security vulnerabilities or any warnings. It could be assumed that developers that have spent more time in development would be better than novice developers in correctly classifying all types of warnings. While we see a small improvement from low to medium experienced developers, this improvement stagnates within the high experience group. These results are positive for the SAT as it shows that every developer within the project can use the tool and get similar results, how good these results are is another issue. A cause of the lack of improvement could be that the output from the tool is so obscure and hard to read that even more experienced developers have problems understanding the warnings.

In the specific experience we see the first improvement in detecting security vulnerabilities. Tool experience is the single most important factor. Security experience on the other hand did not improve the result substantially. But for the best results the developers should have security and SAT experience. No other specific experience showed better results, indicating that educating developers is not the best way to get better SAT classifications. Instead, they need to get practical experience from projects combining security and the use of SAT.

The tools user friendliness comes into question because tool experience was the most important factor when classifying warnings. Because all tools use the same method to present there results it can be considered an industry accepted standard that
every one follows. More specific studies have to be conducted to determine if the output can be presented in a better way and if the classification result can be improved. But a better solution would be to reduce or eliminate false positives entirely so that developers could always assume the tool was correct.

So, depending on whether the tool is used by a security assurance team or as an early fault detection tool the outcome may vary. For a security assurance team a sound tool with higher false positive rate and a minimum of un-reported warnings is acceptable and even preferable. An early fault detection tool is used daily during implementation and therefore a low false positive rate with a slightly increased rate of un-reported warnings may be preferred. Combined with our results the importance of a low false positive rate in the SAT is further strengthened.

C.6 Conclusion

Overall, all developers in the experiment are good at identifying true positives while false positive and security vulnerabilities are much harder to correctly classify. Neither false positive nor security vulnerabilities are identified better than chance. Also, developers are bad at judging how correct their own classifications are.

We see no improvement in the classification with increased development experience or any individual experiences that help the developers in detecting the false positives. For security vulnerabilities, a combination of security experience and SAT experience leads to the best results where using the tool seems more important than having general experience in security. The combination of SAT and security experience almost triple the number of correct security answers. But just deploying a SAT to all developers will not initially have any better than random classifications of security vulnerabilities. With time, as the developers understanding of the tool increases, the better their security results will be (more than doubling the number of correct answers).

Another observation of this study is that developers trust the tool and often assume that the tool is correct in its detection. At the same time no experience improved the false positive rate and the developers could not determine how good they have classified the warnings. These observations show that for a SAT to be an effective early fault or vulnerability detector its false positive rate has to be very low, which is an important factor in the tools usefulness.
Chapter C. Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?
Paper D

Improving Software Security with Static Automated Code Analysis in an Industry Setting

Dejan Baca, Bengt Carlsson, Kai Petersen, Lars Lundberg

D.1 Introduction

Static automated code analysis refers to an automated process for analyzing code without executing it, i.e. several possible scenarios can be explored in quick succession and therefore obscure vulnerabilities might be detected. A contrast to static analysis is dynamic analysis that does its analysis during runtime on a live system, i.e. the code is executed with test cases to fulfill specific coverage criteria (e.g. path coverage, condition coverage, and so forth) and is generally known to be time intensive. Static analysis tools are an important part of several security development processes. Microsoft secure development lifecycle (?), Cigital’s software security touch-points (McGraw 2006b) and other processes (De Win, Scandariato, Buyens, Grégoire, and Joosen 2009) use static analysis tools as an early vulnerability detector. Vulnerabilities are a type of fault that can be exploited by malicious users to e.g. gain access to a system, manipulate data, or make services of the system unavailable. Vulnerabilities are runtime errors and are not detected by the compilers as they are used in industry today, these would have to be extended by features of static code analysis to focus on vulnerability detec-
Vulnerabilities also rarely affect the intended functionality of the code and therefore testing often fails in detecting them. It is also considered a generally accepted fact that faults, including vulnerability, that are found later in the development process are much more costly to correct than faults found earlier (Boehm 2002). Hence, developers use static analysis tools (SAT) to aid in the prevention and detection of vulnerabilities as early as possible in the process.

Previous research evaluated different tools and their capabilities in detecting vulnerabilities (cf. (Zitser, Lippmann, and Leek 2004b)(Baca, Carlsson, and Lundberg 2008)). These papers show that the different tools have the capability to detect vulnerabilities, but that they also produce incorrect results, referred to as false positives. However, previous studies have not been focusing on how SAT can be successfully deployed to assure its usage, and how developers actually use SAT warnings to identify and correct security relevant warnings. In order to utilize the potential benefits of automated static code analysis tools there is a need to understand their capabilities and how practitioners use them.

To address this research gap we performed an industry case study at Ericsson AB that is using the static automated code analysis tool Coverity Prevent (Almossawi and Sinha). First we examined the vulnerability detection capability of the tool, followed by an evaluation of different deployment strategies used at the company. Finally we examined how developers use the tool to detect and correct vulnerabilities. That way this study focuses on the evaluation of the overall life-cycle of automated static analysis, including the activities of tool deployment, tool usage for vulnerability detection, as well as classification, and correction of vulnerabilities. The focus on the life-cycle is deemed necessary to draw an overall conclusion about the usefulness of the tool as an early vulnerability detector.

The remainder of the paper is structured as follows. Section D.2 presents related work focusing on automated static analysis for identifying security vulnerabilities. Section D.3 provides an overview of the research method used, the data collection methods are described in Section D.4. Section D.5 presents the results of the study. Thereafter, the results are discussed in Section D.6. Section D.7 concludes the paper.

D.2 Related Work

To detect the vulnerabilities, static analysis tools use predefined rules or checkers that explain how vulnerabilities look. However, the checkers can report incorrect warnings that do not cause any problem during execution; these are referred to as false positives. Unfortunately, the precision of the analysis usually depends on analysis time. Therefore, precision and analysis time have to be balanced. This is a very subtle trade-off,
if the analysis is fast it is likely to report many false positives in which case the alarms
provided by the SAT tool cannot be trusted. This is especially true for instant feedback
tools that perform fast analysis. On the other hand, a very precise analysis is unlikely
to terminate in reasonable time for large programs. One way to avoid false positives is
to filter the result of the analysis, removing potential errors that are unlikely. However,
this may result in the removal of positives, which are indeed defects. This is known as
a false negative, an actual problem that is not reported. Further details and discussion
on static program analysis and related algorithms can be found in (Nielsen, Nielsen,
and Hankin 1999).

A framework for static analysis is sound if all defects checked for are reported.
However, most commercial systems today (Coverity Prevent, Klocwork K7 and For-
tify) are not sound as they will not find all actual defects and they still produce false
positives (Emanuelsson and Nilsson 2008).

Automated static analysis has been used and researched during several years, e.g.
an early and simple semantic analysis tool was presented by Viega (Viega, Bloch,
Kohno, and McGraw 2000a). Compared to other tools at the time it focused specif-
cically on security. Several research papers (Chess and McGraw 2004c) (Evans and
Larochelle 2002) presented on the benefits of automated static analysis, but often did
not present any real world data to support their claim, instead relying on the theoretical
capabilities of the tool.

Zitser (Zitser, Lippmann, and Leek 2004b) expanded the initial research by eval-
uating how different tools detected known buffer overflows in open source projects.
Carlsson and Baca (Carlsson and Baca 2005) performed similar research, but used se-
curity specific tools and industry software instead. Both determined that the tools did
indeed detect vulnerabilities, but had a high false positive rate that might discourage
users. However, in both papers the data relied on the authors’ use of the tool and not
on results from developers.

Ayewah (Ayewah, Hovemeyer, Morgenthaler, Penix, and Pugh 2008) used Sun JDK
code to evaluate Findbugs, a static analysis tool used to evaluate Java code. Again, the
author performed the analysis and did not use developers to acquire the data. How-
ever, the author performed a short survey to determine the adoption and usage of the
tool and stated: “it seems clear to us that development, measurement, validation and
adoption of best practices for static analysis tools is key to allowing these tools to be
used effectively.” Emanuelsson and Nilsson (?) wrote a paper that evaluated different
state of the art tools in an industry setting and provided some industry experience on
different tools’ capabilities, but limited the study to developer opinions and the theo-
retical capabilities of the tools. With regard to developers using static analysis Baca et
al. (Baca, Petersen, Carlsson, and Lundberg 2009) investigated the effect of different
levels of experiences on the accuracy with which software developers identify security
relevant warnings from static analysis tools, finding that combinations of security and static analysis experience lead to a better than random result in classifying warnings correctly.

So far, it seems that previous research is lacking empirical research on industry usage and deployment of automated static analysis tools and their impact on software security. This paper aims at contributing to fill the research gap of limited empirical evidence of usage of SAT tools to detect and correct vulnerabilities, providing industry experience on Tools capabilities (what types of vulnerabilities the tool detects and volume of faults it reports), Deployment (how different projects deployed and integrated the tool) and Usage (how developers handled the output of the tool to identify and correct vulnerabilities).

D.3 Research Method

We use case study as a research method (Yin. 2003) to examine the entire cycle of using a static code analysis tool in an industry setting. Case study research is used to gain an in-depth understanding of a situation in real-world context and is able to provide rich descriptions of what is happening in that context (cf. (Yin. 2003) and (?)). The case studied was Ericsson AB. The overall goal of the case study was to understand the benefits and limitations of SAT in an industrial context to improve software security. As mentioned earlier the overall goal was broken down into three sub-goals to cover the life-cycle of SAT usage, namely investigating defect detection capability with respect to giving security relevant warnings, tool deployment, and understanding how developers use the tools to identify and correct warnings. With regard to the research methodology we describe the research context, units of analysis and research goals. Thereafter, in Section D.4 the data collection methods are presented in further detail.

D.3.1 Units of Analysis and Case Study Context

To aid generalizability of the results and in which context they hold it is important to provide details on the studied context (Petersen and Wohlin 2009b).

Ericsson AB is a leading and global company offering solutions in the area of telecommunication and multimedia. Such solutions include billing systems for mobile phones, multimedia solutions and network solutions. The company is ISO 9001:2000 certified. The SAT tool was introduced into the development process two years prior to this study and, while all the examined projects reside within the same company, the products are not restricted to one development site within the company. Some of the examined products were developed on several sites simultaneously and therefore this
study presents data from different development sites in China, Ireland, India and Sweden. All products have passed several revisions while using Ericsson Streamline development process, which is initially a more flexible waterfall process that has evolved over time and is today more agile (for further details of the development process and its evolution see (Tomaszewski, Berander, and Damm 2008b; Petersen and Wohlin 2010; Petersen and Wohlin 2009a)). All products were developed using C++ and followed a server-client architecture. Product A, B and D have been on the market for more than 5 years while Product C has only been available for 1.5 years. All products have a major release every one to one and a half years, the examined data and source code is from the latest major release of each product. The units of analysis in this case study are four different products using Coverity Prevent:

- **Product A** had about 600,000 lines of analyzed code. The code base included some third party code and an in-house framework. The framework was also included in the case study because bugs on the framework were reported to the product. Product A received, retrieved and stored data. It also handled and processed user data. There has been an attempt to perform mandatory manual code reviews in the product, but these initiatives have all been short lived. The product was developed at two different development sites both independently and together on the same release. The product has been in development for 8 years.

- **Product B** had about 300,000 lines of analyzed code. Product B handled large amounts of user data but did not do any heavy processing. Its primary function was to shuffle data between different endpoints. Historically this product used lint (Johnson 1977), an open source static code analysis tool. However, the practice stopped several years before the start of the study. The product has been in development for 6 years.

- **Product C** had about 50,000 lines of analyzed code. Because of its size it was developed by a more dedicated small group of developers. Product C served and processed data to end users, but compared to product A and B it could not handle the same amount of possible input combinations. The product has been in development for 3 years and only one major release.

- **Product D** had about 800,000 lines of analyzed code. The product’s main role was as a statistic data gatherer. Even though it did handle large quantities of data it did minor processing and focused more on throughput and storage. The product was developed on three different sites, but at a maximum of two sites was involved in its development at the same time. The product has been in development for 6 years.
In this study, we have chosen the tool Coverity Prevent (version 3.1). Coverity Prevent is one of the state of the art SATs and has been used in other studies to detect bugs and vulnerabilities in the Linux kernel (Guo and Engler). Coverity has a low number of false positive (10-20%) and it detects a high number of faults (Baca, Carlsson, and Lundberg 2008). The company did not decide on a specific adoption strategy or any mandatory training in using the tool. A more security focused SAT (Fortify) was excluded due to its higher false positive rate and other “free-ware” tools were too simple for an industrial setting and produced too many false positives.

D.3.2 Research Goal

The goal of this study was to provide a comprehensive evaluation of static code analysis for improving the quality of software products from a security perspective. The company’s initial trial with static analysis tool showed a larger potential than the perceived outcome after deployment, this was especially true for security bugs (Baca, Carlsson, and Lundberg 2008). To determine why detectable vulnerabilities were slipping through we examined the overall life-cycle of SAT, including:

1. Tool capabilities: Static analysis is limited to the information present in the source code and therefore not all types of vulnerabilities can be detected by the tool. We were interested in knowing what types of vulnerabilities a static code analysis tool can detect in an industry setting using release ready source code and already known vulnerabilities. In Sections D.4.1 we explain how we evaluated the tools capabilities in finding possible vulnerabilities and already known vulnerabilities in legacy code.

2. Deployment: When introducing a new method or process there is always a high risk that this introduction might fail, e.g. due to resistance to change. In order to facilitate the successful introduction we looked at different strategies that were used at the company for introducing SAT and evaluate them. The method used to evaluate the adoption strategy is further explained in Section D.4.2.

3. Usage by Developers (Vulnerability Identification and Correction): Even if a tool provides accurate and relevant results it is of no use if developers can not understand what the output means. We therefore investigated if it is harder to understand security warnings compared to regular warnings and false positives, e.g. incorrect warnings. Vulnerabilities can be especially hard to understand as it is often required that the software enters a certain state or that a specific data flow occurs. These states and data flows are often not in the original design and the developer never intended them to be present. Thus, it is easy for developers
to dismiss vulnerabilities with “that will never happen”. In Section D.4.3 we present how we examined developers’ effectiveness in identifying and correcting the vulnerabilities presented by the SAT tool.

The following section provides details on how the data has been collected to investigate tool capabilities, deployment strategies, and usage of the tool by developers.

**D.4 Data Collection Methods**

In this study three data collection methods have been used, namely post-mortem-analysis, open-ended interviews, and questionnaire. The post-mortem-analysis was used to investigate the tools capabilities by studying historical data and the vulnerabilities that had been detected. To evaluate deployment of the SAT tool we conducted semi-structured interviews. A questionnaire was used to examine how developers classified warnings, and we observed the correction of the warnings contributing to our understanding how developers interact with the tool.

**D.4.1 Tool capabilities**

The tool’s capability was determined by evaluating the warnings the SAT tool reported and how developers corrected them. We also examined how well the tool could detect already known bugs by running an analysis on old source code with known faults.

**Static analysis findings**

To determine the tools detection capabilities in an industry environment we relied on historical data provided by both the tool and the bug reporting system. This was done as a post-mortem analysis (Myllyaho and Koskela 2004). The tool’s historical data was used to investigate which warnings developers corrected and how they classified the warnings. This information was also critical in determining how successful the deployment of the tool was. By examining the description of the warnings and the source code (both being visible in the tools interface) we could further classify the severity of the corrected warnings. We classified the warnings in the following three categories:

- *False positive* is an incorrectly reported warning. These are often caused by bugs in the tool, bug pattern or by the weakness of the checker. The developers classified these warnings themselves and no further verification was done to determine if it was a correct classification.
Chapter D. Improving Software Security with Static Automated Code Analysis in an Industry Setting

- **True positive** is a correctly identified fault that does not affect the “In Service Performance” from a security point of view and was often ranked as low priority by the developers. “In Service Performance” thereby refers to the performance of the system when being used by the customer. Faults that are not possible to be induced by a user and affecting in service performance are automatically classified as true positives as the user has to be able to trigger the fault for it to be relevant from a security perspective.

- **Security**: A correct warning that has been corrected and can propagate into a user induced failure of the system. We (authors of this paper) classified a fault as security relevant if it could result in any of the following scenarios: denial of service, unauthorized disclosure of information, unauthorized destruction of data or unauthorized modification of data. The requirement that a user of the system can trigger the fault is important since many of the warnings (true positives) are correct, but can never be triggered and hence are not effected by malicious users. As an example, buffer overflow is only a security warning if a user can in any way effect the input into that buffer. A possible reason why the buffer overflow is not relevant from a security perspective is that the user input has been validated outside the visible source code following the telecommunication “Abstract Syntax Notation One (ASN.1)” standard and therefore preventing the buffer overflow.

We are distinguishing true positives and security warnings because this study focuses on the security aspect of static analyses. To classify the security warnings we used the taxonomy from Tsipenyuk and Chess (?) that is more specific towards implementation faults. It was selected because static analysis only detects implementation related faults. The outcome is presented in each category with the name of the checker that detected the fault. An overview of the categories and examples of vulnerabilities within the categories are provided in Table D.1.

**Bug report comparison**

We used existing bug reports to determine whether existing security relevant bugs could have been prevented by using SAT. To identify the bugs that could have been prevented we conducted the following steps:

1. Retrieved a list of bug reports that were classified as effecting “In Service Performance” e.g. these warnings could result into one of our four failures presented as a security warning in Section D.4.1.
Table D.1: Taxonomy used for classification.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Vulnerabilities (Examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input validation and representation</td>
<td>buffer overflow, command injection, cross-site scripting, format string, illegal pointer value, integer overflow, SQL injection, etc.</td>
</tr>
<tr>
<td>API abuse</td>
<td>Contract violations between caller and callee, e.g. due to dangerous function, heap injection, unchecked return value, etc.</td>
</tr>
<tr>
<td>Time and State</td>
<td>Deadlocks, file access race conditions (TOCTOU), insecure temporary files, etc.</td>
</tr>
<tr>
<td>Security features</td>
<td>Authentication, access control, cryptography issues caused by e.g. insecure randomness, least privilege violation, missing access control, etc.</td>
</tr>
<tr>
<td>Errors</td>
<td>Improper handling of errors by e.g. catch “NullPointerException”, empty catch block, overly broad catch/throws blocks/declarations, etc.</td>
</tr>
<tr>
<td>Code Quality</td>
<td>Unpredicted behavior due to poor code quality e.g. due to inconsistent implementations, memory leaks, obsolete functions, faults due to initialization, use and release of resources, etc.</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>Weak boundary definitions related to data, users, access rights, and classes, e.g. due to data leaking between users, mobile code, trust boundary violation, etc.</td>
</tr>
<tr>
<td>Environment</td>
<td>Security issues caused outside the actual source code, e.g. storage of passwords in files, accessibility to debug files providing info to malicious users, etc.</td>
</tr>
</tbody>
</table>

2. Running static code analysis on a year-old version of the product and on the latest version (but before deploying the tool). Warnings that are not present in the latest version had been removed during the maintenance phase of the product that consists mostly of bug corrections.

3. Comparing changed lines of code from the bug patches with the static analysis warnings from step 2. Warnings that are within the vicinity of the lines changes were selected for further examination.

4. Determined if the warning could have detected the bug and therefore prevented the bug report.

5. Developers were asked to verify that the tools warning message would have prevented the bug report.


Chapter D. Improving Software Security with Static Automated Code Analysis in an Industry Setting

D.4.2 Deployment

To evaluate the adoption strategies and deployment of the tool we conducted semi-structured interviews with 11 project managers with the purpose of identifying which projects successfully deployed the tool. The interviews with the project managers were followed by 15 semi-structured interviews with the developers or people that had been involved in using the tool. We asked about their experience in using the tool and what problems they had in their adoption strategy. In more detail, we asked the interviewees to provide feedback on two aspects:

1. Deployment of the tool: Initially we investigated how the different projects deployed and used the static analysis tool (SAT), this was necessary because there was no global directive how the tool should be used or deployed. This was achieved by contacting the project leaders and asking them about SAT tool deployment and usage. Besides answering the interview, the projects also provided access to their revision repository, bug tracker and static analysis database. The static analysis database was part of the SAT and provides historical data of every analysis.

2. Usefulness of the tool: We knew from the Post-Mortem-Analysis (see Section D.4.1) how effective the tool has been in discovering vulnerabilities. However, it was also important to evaluate how useful the developers feel the tool had been. By interviewing developers that used the tool we gathered their experience.

The interview was documented by taking notes during the interview and by clustering and structuring the notes according to themes (e.g. clustering benefits and issues mentioned). Based on the summary we narratively described the results.

The interview questions can be found in Appendix D.9.

D.4.3 Usage by Developers

To understand how developers used the tool we investigated two activities. First, the developers were asked to use the tool’s web interface and classify a random selection of warnings. Secondly, developers used the IDE to correct the warnings they had classified before. The classifications the developers could do is further explained in Section D.4.1.

Classifying warnings

While examining new warnings or correcting a large quantity of legacy warnings, the first step would be to determine if it is worth the effort to correct the perceived fault.
This is a necessary step to understand the warnings and it is necessary since the tool can report false positives that should not be corrected. To examine how well developers identify false positives and security warnings we asked a random group of software developers from different development sites (Sweden and India) to use the output of the SAT tool for classifying a randomly selected list of warnings.

**Warnings**: The original classification was done by the developers working on the product. The random sample was also examined further by the authors to ensure that the original classification was correct. In some cases the vulnerabilities were also practically verified for being accurate, however this was not performed on all of them because of the time required to write working exploits. The random sample consisted of 22 warnings of which 8 were false positives and 9 were security warnings, the rest were true positives. At the time of the investigation the product had an approximated ratio of 25 true positives and 8 false positives for every vulnerability, i.e. true positives are under-represented and security warnings are highly over-represented in the questionnaire due to the need for sufficient data points. All developers classified the same randomly generated sample.

**Subjects**: The study was voluntary for the developers and they all received the same written guideline. The guideline explained their task, including explanations of classification types (false positives, true positives, and security fault). Security faults are further explained in the instructions following the definitions of vulnerabilities (Krsul 1998b). This definition includes the coding faults that cause a Denial of Service attack, unauthorized disclosure, unauthorized destruction of data, and unauthorized modification of data. In total 34 developers from two different development sites answered the questionnaire.

**Instrumentation**: The developers used the tool’s web interface to read the warnings and access the source code. To classify the warnings developers filled in a digital or paper form where they could select what type a specific warning was. This procedure was similar to how classification would have been done in the web interface but the web interface could not be used during the study as developers would write over each other’s classification. Instead they had to fill in the check boxes in the form. The questionnaire for classifying the warnings can be found in Appendix D.10.

**Correcting warnings**

After the classification study we asked developers to correct the warnings they had previously classified. The developers used their IDE and the tool’s web interface just as they would in a live setting.

**Warnings**: The same warnings as in the classification study was used. The acceptable corrections of the warnings were determined with the aid of a senior developer.
Chapter D. Improving Software Security with Static Automated Code Analysis in an Industry Setting

In the project. The developers used their own classification when making corrections, which means they are following a realistic scenario.

**Subjects:** The study was voluntary for the developers and they all received the same written guideline. In total 32 developers from two different development sites answered the study and consisted of a subgroup from the developers that did the classification study.

**Instrumentation:** The static analysis tool was used to examine the source code and to understand the warnings from the analyses. To correct the warnings developers used Eclipse (Vaughan-Nichols 2003), an open source IDE that is part of the usual development process. However, not all developers could compile the entire code while correcting the faults. As such we ignored any simple grammatical or schematic errors in the correction that would have been detected by the compiler.

**Evaluation:** With the aid of a senior developer the corrections were examined to determine if they solved the fault, failed to solve it or if a new fault had been introduced. Furthermore, any new faults were evaluated to determine whether they presented a threat to the system. If the correction was faulty we tried to determine why this occurred and if needed we interviewed the developer. The static analysis was also rerun on the corrected code to determine if the warnings had been removed.

## D.5 Results

The results are divided into three main sections. In Section D.5.1 we present the historical findings and capabilities of the tool. Section D.5.2 shows results related to the deployment of the tool, and finally in Section D.5.3 we present results related to the usage of the tool by developers to identify and correct vulnerabilities.

### D.5.1 Tool capabilities

Four products that used the static analysis tool over a period of time were selected for an evaluation. Each product had all of its warnings classified, either as part of this study or by the developers that used the tool. The classification was done with the aid of the Coverity web interface and is considered an initial step in using static code analyses tools. For some of the security warnings we had to verify them practically because it was unclear if any user of the system could trigger the fault (i.e. by writing an exploit). Table D.2 shows the total number of warnings, how many of those that are incorrect and deemed false positives, the number of vulnerability warnings that could be exploited by users and lastly how many of the warnings could have prevented an already corrected bug report, see Section D.4.1 on how we identified these. What can
be observed from the data is that for the larger systems approximately the same ratio of
warnings (around one quarter) were identified as false positives. Fewer warnings were
reported for the small system (Product C) and for Product B that is not focusing on
data processing and is smaller in comparison to Product A and D. The table also shows
that for all products security relevant warnings were identified with Products A and B
having the highest absolute number of security warnings due to their size. The highest
ratio of security warnings was identified in Product C where almost one third of all
warnings were security related. All products have bugs that could have been prevented
when using static code analysis.

Table D.2: Initial classification of total amount of reported warnings, the number of
reported false positives by the developers, the number of security warnings and the
number of known bug reports that were also reported by the static analysis tool.

<table>
<thead>
<tr>
<th>Product</th>
<th>Total warnings</th>
<th>False positives</th>
<th>Security warnings</th>
<th>Bug reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product A</td>
<td>1680</td>
<td>371 (22%)</td>
<td>86 (5%)</td>
<td>3</td>
</tr>
<tr>
<td>Product B</td>
<td>132</td>
<td>7 (5%)</td>
<td>16 (12%)</td>
<td>2</td>
</tr>
<tr>
<td>Product C</td>
<td>33</td>
<td>2 (6%)</td>
<td>9 (27%)</td>
<td>1</td>
</tr>
<tr>
<td>Product D</td>
<td>3236</td>
<td>817 (25%)</td>
<td>57 (1.7%)</td>
<td>8</td>
</tr>
</tbody>
</table>

Vulnerability detection

From the eight classifications in the taxonomy (?) only four groups were detected in
our examined products. Vulnerabilities were identified in the categories of validation
and representation, API abuse, time and state, and code quality/coding errors. Each
table below explains in more detail how many of the vulnerabilities were detected and
which category they belong to.

Table D.3 is the largest group and every product had most of its vulnerabilities in
this category. Most of the vulnerabilities in this category were related to array memory
management and the vulnerabilities can often be exploited severely. Buffer overflows
are buffers that can be overwritten by non-validated input data. Format String also
often result in buffer overflows but in this case the attacker does not have as much
control because he can force extra data into the buffer but is limited to what that data
is. String Termination Error are arrays or strings that the attacker can insert as non
terminated, any read from these string would continue reading until a termination was
found. Looking at the data in Table D.3 it is apparent that for Product A most types
of vulnerabilities were detected. Furthermore, all products led to warnings related to
buffer overflows independently of their size or tasks (such as data processing, storage,
Chapter D. Improving Software Security with Static Automated Code Analysis in an Industry Setting

and so forth).

Table D.3: Detected vulnerabilities caused by lack of input validation and representation problems. Left column name refeers to the static analysis checker used to detect the vulnerability.

<table>
<thead>
<tr>
<th>Input validation and representation</th>
<th>Product A</th>
<th>Product B</th>
<th>Product C</th>
<th>Product D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Overflow</td>
<td>23</td>
<td>9</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>Format String</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>String Termination Error</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Setting Manipulation</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table D.4 shows API abuses and product D in particular stands out, while product C that is much smaller has no API vulnerabilities. The size of the software and number of third party libraries affect this category greatly. Missing Check against Null are instances where an API might return null or not perform the intended action resulting in a null pointer or reference. The tool detects these warnings in two different ways, it either shows the direct path how a possible Null pointer is used or it statistically shows that calls to this API is often verified and there are some instances where this verification is not done. Unchecked Return Value functions in similar way to the Missing Check against Null vulnerabilities, but here the return values are often negative when only positive values are expected. Exception Handling are exceptions that are ignored by the software even if they can be thrown.

Table D.4: Detected vulnerabilities caused by breaches in the API contract between a caller and a callee. Left column name refeers to the static analysis checker used to detect the vulnerability.

<table>
<thead>
<tr>
<th>API abuse</th>
<th>Product A</th>
<th>Product B</th>
<th>Product C</th>
<th>Product D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing Check against Null</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>Unchecked Return Value</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table D.5 shows different types of race conditions. These vulnerabilities are often very hard for a SAT to detect because they often require some flow analysis. Insecure Temporary Files are temporary files that use poor random names and are vulnerable to file based race conditions. File System Access are file base race conditions were the file
is first verified and then without any locking used and assumed nothing has changed. Missing Lock are possible deadlocks or instances where a variable should have been locked before it is changed. Table D.5 shows that race conditions are only identified for the largest products, i.e. Product A and D.

Table D.5: Detected vulnerabilities caused by concurrency or other time based faults. Left column name reefers to the static analysis checker used to detect the vulnerability.

<table>
<thead>
<tr>
<th>Time and state</th>
<th>Product A</th>
<th>Product B</th>
<th>Product C</th>
<th>Product D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insecure Temporary File</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>File System Access</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Missing Lock</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Table D.6 are code quality issues that are not always directly security related, but many of the detected faults could be used to create denial of service attacks against the servers. This is a major issue for telecommunication providers where uptime is a contract-based requirement. Memory Leak are user induced memory or resource leaks where a user can, through legitimate or faked requests, waste system resources and create a denial of service attack. Null Dereference is caused when a pointer with a value of NULL is used as though it pointed to a valid memory area, often resulting in a system crash. Uninitialized Variables are variables that can be read without any initializing or prior use, what these values contain could be random and the behavior of the system can vary. “Use After Free” is a pointer that is used after it has been freed and are similar to Null Dereference but have specifically been freed already. Function Not Invoked is unused dead code. Similar to the previous observations Table D.6 shows that the larger the system, the higher the variety of faults/vulnerabilities in terms of fault categories covered.

Table D.6: Detected faults caused by bad code quality. Left column name reefers to the static analysis checker used to detect the vulnerability.

<table>
<thead>
<tr>
<th>Code quality</th>
<th>Product A</th>
<th>Product B</th>
<th>Product C</th>
<th>Product D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Leak: User controlled</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Null Dereference</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Use After Free</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Function Not Invoked</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>
Chapter D. Improving Software Security with Static Automated Code Analysis in an Industry Setting

In Figure D.1 the four groups are combined in one figure for easier overview. It is easily seen that input validation and representation is the largest group, this group includes buffer overflow which often creates problems for C++ products. Some groups of vulnerabilities were not detected in any of the products and product C had mostly only buffer overflows. The results and usefulness of SAT varied between the products. Product A had several input and validation faults and the SAT detected a vulnerability for every ~7000 lines of code. Product C that on the other hand only had a few non-security faults, found one vulnerability for every ~5500 lines of code. Product B and D on the other hand found few vulnerabilities per line of code, one in ~19000 lines of code and one in ~14000 lines of code. This can be important to examine because the tool is paid by the lines of examined code, it might be possible that the tool and the examination might be more costly than to detect the vulnerability later in development.

Bugreport Detection

Table D.2 provided the number of bug reports that could have been detected if static analysis would have been used. The following paragraphs provide further details on these bugs.

Product A Three bug reports were detected by the static analysis tool. They were all buffer overflows that occurred during a common copy operation involving the “sprintf” command. The bugs were detected by product testers and rated as high priority.

Product B Two resource leaks that were reported after product release were detected by the tool. The resource leaks were triggered by a specific type of input message that was not very common. This was probably the reason why testers had not detected it before release.

Product C Only one bug report was detected by the static analysis and it was a buffer overflow, too. The fault was immediately detected by the testers and resulted in a “segmentation fault” during testing. The bug report stated that the fault should have been detected earlier during basic unit testing instead of the later test phase, which required a quick fix to continue testing.

Product D Eight bug reports were detected by the tool. Four involved buffer overflows that were caused by similar copy operations and that were reported by testers. One format string exploit was reported by customers, although not as a security bug. Two “segmentation faults” that were found by testers and involved bad dereferences in specific cases. Lastly, a race condition that was caused by a
missing “lock” operation. However, the bug report did not specify race condition or security in any way. The report only reported of strange and inconsistent errors. The correction to the bug report did add the missing “lock” operation and it was determined that the same correction would have been done when solving the warning.
D.5.2 Deployment

Three distinctly different approaches of deploying and using the tool have been identified at Ericsson. Each project could decide on their own how to introduce the tool, because of this there were deployment and usage variation within the company. Ericsson’s goal was to introduce the static analysis tool as an early vulnerability detector with as little overhead as possible.

1. **No overhead approach.** These projects had no overhead at all for introducing the tool. There were no requirements to use the tools or any verification that the tool is used. The tool is intended to be used voluntarily by the developers.

2. **Champion approach.** In these projects a champion or responsible person for the tool was selected. This person’s main responsibility was to keep the tool operational and help other developers to use the tool.

3. **Configuration management driven approach.** These projects integrated the tool into their bug report system and had one or several developers responsible for classifying and reporting any warnings.

In the following we summarize the feedback given by the interviewees with respect to the three strategies identified.

**No Overhead:** The first method had a large number of failed deployments. Most often the projects paid for the tool, but in the end never actually used it on their source code or, in some cases, they conducted one test run and never integrated it into their development process. Developers complained about the lack of time to use or integrate the tool, also several developers stated that they choose not to use the tool. In order, these are the most commonly stated reasons by developers and project managers why they did not use the tool:

- Lack of time to use the tool.
- Did not understand the tool.
- Lack of warnings in the new code (the tool had to be rerun manually).

**Champion Approach:** In the second method projects had integrated the tool into their development process and they could either daily or on demand run an analysis. In one project where the champion had left the project the tool had been abandoned after it stopped reporting faults, this was due to an expired license and was not discovered before this study was done. In these projects, the responsibility to correct warnings was usually on the champion. Most developers assumed the champion will correct all
warnings and few developers ever used the tool. These projects often ignored their legacy warnings and only corrected new warnings. According to the developers this method had the following weaknesses:

- Dependent on a single person (the champion)
- No knowledge transfer between champion and team.
- Developers did not understand the tool, only the champion did.

**Configuration Management Integration:** The third method had the largest overhead and the tool was often not used as an early fault detector. However, compared to the other methods these projects had more people using the tool. When deploying the tool the projects classified all legacy warnings and reported any warnings that needed correction to the bug handling system. The project either had one champion for the entire project or one developer per component that was responsible for reporting warnings. These reports were often done at the end of a release. Because this method used the bug report system it created a large overhead, very often it was easier to correct the bug than to answer the bug report correctly. However, the number of reported bugs from champions decreased over time because the developers started to use the tool by themselves, if they did this they would not have to answer the bug report. Projects using this deployment strategy had more tools and technical complains, compared to the adoption complains of the previous two methods:

- Time consuming to report warnings
- For some warnings it was hard to understand why it was a fault
- Many of the warnings were perceived as unimportant

Most of the qualitative data for this paper was acquired by projects using the third method, because they were the only projects that used the tool to such an extent that useful data could be collected, indicating that the last strategy is the most useful one in the given context.

**D.5.3 Usage**

**Warnings classification**

The first task for all developers who use SAT’s is to classify the warnings, even if the classification is only done in the mind of the developers they still have to perform the same task and determine if this warning is worth the time to correct. When classifying
the warnings we gave the developers three choices, either the warning was a false positive, a true positive or a vulnerability. The distinction between true positive and security warnings can be important to the way they later correct the warning and it is a way to determine if developers understand what the tool is showing. For an early vulnerability detector to be effective even non-security developers should be able to detect vulnerabilities with the tool.

In this questionnaire a group of 34 developers have individually classified 8 false positive and 9 security warnings. The warnings were originally classified by the developers writing the code for the products under investigation. Furthermore, they were practically verified by writing exploits for them. It should be emphasized that the tool provides no classification determining whether a fault is security relevant or not, they classify the faults as, for example, an uninitialized variable (see Table D.11). Hence, the developer has to decide whether a warning is deemed security relevant. Because most developers classified the true positives correctly we will only explain the false positives and security warnings in greater detail. Also we extended the initial study presented in (Baca, Petersen, Carlsson, and Lundberg 2009) by asking developers to correct the warnings in these two groups.

Table D.7: Developers capability to identify and classify false positives. The dark grey column, False positives, is the correct answer.

<table>
<thead>
<tr>
<th>Checksers</th>
<th>False positive</th>
<th>True positive</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized Variable</td>
<td>12</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>9</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>13</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>7</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>Null Dereference</td>
<td>22</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Erroneous Synchronization</td>
<td>21</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Use After Free</td>
<td>10</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>8</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>37,5%</td>
<td>56,6%</td>
<td>5,9%</td>
</tr>
</tbody>
</table>

In Table D.7 shows the warnings of the checker that were false positives, and how the developers classified them. For example, 12 persons classified uninitialized variable as a false positive (i.e. correctly), 22 as a true positive, and no person said that it is a vulnerability. The majority identified few of the warnings as false positives and it seems that in this case the developers overestimated the correctness of the tool. Several developers also thought that the false buffer overflow was a security risk that needed
to be corrected, this later created more problems when they tried to correct the non-existing fault.

Table D.8: Developers capability to identify and classify vulnerabilities. The dark grey column, Vulnerability, is the correct answer, while False positive is a very dangerous answer as these would be ignored.

<table>
<thead>
<tr>
<th>Vulnerabilities</th>
<th>False positives</th>
<th>True positive</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Checkers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>2</td>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>Unchecked Return Value</td>
<td>7</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>Null Dereference</td>
<td>7</td>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>Null Dereference</td>
<td>8</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>Insecure Temporary File</td>
<td>3</td>
<td>24</td>
<td>7</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>1</td>
<td>23</td>
<td>10</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>0</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>5</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>File System Access</td>
<td>9</td>
<td>17</td>
<td>8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>13,7%</strong></td>
<td><strong>67,7%</strong></td>
<td><strong>18,6%</strong></td>
</tr>
</tbody>
</table>

The classification of the security warnings is similar to the false positive, again the majority thought the warnings were correct and very few developers saw that warning as a vulnerability. Only on the buffer overflows did 1/3 of the developers correctly classify a warning as a vulnerability. The two race conditions had some correct classifications while the simpler user induced segmentation faults from reference and dereference operations were hardly seen as any vulnerabilities.

The security classification is not as important as the next subsection where the corrections are compared, however the classification is the first developer interaction with the warnings and seeing as many project first classify the warnings and then later correct them it is important that the classification does not hide vulnerabilities and that the correct people then correct them.

**Warnings correction**

We first examine how the developers corrected the false positives. These warnings were incorrect and should not be corrected at all. That is, if developers ignored false positives in the correction they recognized them as such. In Table D.9 the false positives and how many developers corrected them are displayed in detail. The three uninitialized variable warnings were not harmful to correct and it was the tool that did not understand that the
variable had been initialized in parts before usage. Similarly, the incorrect exception handling warning posed no threat. The null forward warning only had two instances where the correction would cause segmentation faults during execution. However, for the remaining three warnings most of the corrections caused functional faults. The use after free correction prevented the program from accessing necessary data that had not been freed. In the case of the buffer overflow, seven of the corrections actually created a new vulnerability. This was caused because the developers removed the validation that prevented the buffer overflow. Only the buffer overflow correction would not have been detected during normal testing and it is therefore considered more severe.

Table D.9: Developers correction of false positives. Ideally all developers should ignore, grey column, all of these warnings. Harmless corrections alter code but do not affect the software, while Harmful corrections would harm the software. Warnings are sorted based on the severity of the Harmful correction, with the most harmful at the bottom.

<table>
<thead>
<tr>
<th>False positives</th>
<th>Checkers</th>
<th>Ignored</th>
<th>Harmless corr.</th>
<th>Harmful corr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized Variable</td>
<td>11</td>
<td>21</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>11</td>
<td>21</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>8</td>
<td>24</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Exception Handling</td>
<td>6</td>
<td>26</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Null Dereference</td>
<td>21</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Erroneous Synchronization</td>
<td>20</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Use After Free</td>
<td>10</td>
<td>0</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>7</td>
<td>18</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>36.7%</td>
<td>47.3%</td>
<td>16%</td>
<td></td>
</tr>
</tbody>
</table>

Table D.10 shows the results from the actual vulnerabilities. Except for the last vulnerability, the majority of developers would have corrected the fault in such a way that the vulnerability would be prevented. This shows one of the strengths of static analysis because the tool can precisely show the developer what the fault is and the developer can correct it even though he might not have the full understanding why this fault would create a vulnerability. The unsafe correction for the insecure temporary file did not use the safe "mkstemp" call but instead tried and failed to implement a solution of its own, the usage of "mkstemp" is recommended by the tool and the coding guideline and is the most probable reason why so many developers corrected the issue securely. While a similar vulnerability, the file system access warning only had eight secure corrections, the same eight developers that correctly classified the warnings as a
vulnerability. In this case, the tool reported a possible "Time of Check - Time of Use" attack for a system file. Most of the unsafe correction moved the check and use closer in the code or changed privileges to the file in an insecure way so the vulnerability remained. The buffer overflows varied from string overflows to string formats and the incorrect solutions sometimes just increased the size of the buffer or created the possibility of a lacking Null terminated buffer.

A code example of a vulnerability identified by the tool and a detailed explanation of its classification and correction by developers is shown and explained in Appendix D.11.

Table D.10: Developers correction of vulnerabilities, Only the gray column, Secure correction, can be consider an acceptable answer. Ignored were not correct because developers have previously classified the warnings as false positivist and unsafe correction do not address the vulnerability.

<table>
<thead>
<tr>
<th>Vulnerabilities</th>
<th>Checkers</th>
<th>Ignored</th>
<th>Unsafe corr.</th>
<th>Secure corr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized Variable</td>
<td>2</td>
<td>1</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Unchecked Return Value</td>
<td>6</td>
<td>0</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Null Dereference</td>
<td>6</td>
<td>0</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>7</td>
<td>0</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>1</td>
<td>9</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Insecure Temporary File</td>
<td>0</td>
<td>10</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>2</td>
<td>9</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>4</td>
<td>13</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>File System Access</td>
<td>7</td>
<td>17</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>12,1%</td>
<td>20,5%</td>
<td>67,4%</td>
<td></td>
</tr>
</tbody>
</table>

D.6 Discussion

D.6.1 Tool Capabilities

After examining the usage of SAT on four different products, it is interesting to see that all products had detected a large group of memory related vulnerabilities. These vulnerabilities were often input validation faults but also API abuses, most of the vulnerabilities would not have been possible if the product was written in a more memory secure language, for example Java (Livshits and Lam 2005). This is further shown as none of other products at the company that were developed in Java had any reported
Chapter D. Improving Software Security with Static Automated Code Analysis in an Industry Setting

warnings that we could classify according to our taxonomy. There are numerous studies (Cowan, Wagle, Pu, Beattie, and Walpole 2000) showing that memory management causes many of the software vulnerabilities, it is therefore good that SAT seems effective in detecting these types of faults. Other types of vulnerabilities were detected more sporadically, but this is probably because these vulnerabilities are less common and not based on the tool’s ability to detect them.

It is also important to highlight that the tool did not detect any vulnerabilities related to the categories security features, errors, encapsulation, and environment. A possible explanation is that, even though the categorization is focusing on implementation faults, many of the vulnerabilities in these categories are design related and therefore are not detected by the tool, e.g. categories security features, encapsulation, and environment. For example, how authentication is handled and what approach for cryptography to choose are all design issues. Another example is environment that is outside the control of the SAT tool and very much related to a security aware behavior (e.g. by making sure that password files are not easily accessible).

In conclusion we have shown that the tool is able to detect vulnerabilities in the categories input validation and representation, API abuse, time and state, and code quality. In addition, we have shown that the SAT could have detected severe vulnerabilities earlier that were later detected in testing.

D.6.2 Deployment

The usefulness of the static code analysis tool was hindered by its lack of adoption in the company; just stating that SAT should be used was not enough to achieve wide adoption. Only adoption strategies that used a configuration management approach had a good success rate, which is contradictory to an early fault detection tool, as it should have as little overhead as possible. Instead, this approach created overhead and increased the lead-time between detection and correction of a vulnerability. However, there were indications of a positive trend where developers started on their own initiative to examine the tool’s output. This was caused by the burden of answering bug report from SAT output, as the output was often easier to correct than answering the bug report. But the tool’s output was also the reason why the tool was not always used. Some warnings that the tool presented as vulnerabilities were seen, by developers, as unimportant and scenarios that never happened. Hence, the developers did not classify them as security relevant (see Table D.8) However, when fixing what the developers believed was a true positive that in fact was a security threat, the majority of developers made a secure correction (see Table D.10). Overall, we can conclude that from a practical point of view the configuration management approach is recommended as it gets people to use the tool, which is a pre-requisite for making use of the tool’s output.
Overall, we conclude that the configuration management approach is the most suitable of the observed ones in the company, as it was the only one providing sufficient data for an analysis of the capability of SAT to detect vulnerabilities. A practical solution to improve efficiency of the configuration management approach is to educate developers about efficient usage, and why they should correct reported vulnerabilities immediately (i.e. to save time going through official bug reporting and defect fixing processes).

D.6.3 Usage

For the SAT to be an effective early vulnerability detector developers have to be able to use the tool during their development. To examine if this is true we first asked a group of random developers to classify a random sample of warnings. The initial results were not as promising as expected because many of the false positive and security vulnerabilities were not correctly classified. Because of the way the tool is used the initial classification is important, ideally, a developer should receive a warning and immediately try to correct it. Unfortunately we have observed that the procedure is more often that the warnings is first classified as a true positives or false positive and in some cases prioritized. This adds extra weight to the initial classification and its importance. A vulnerability that is classified as a false positive or as a low priority fault would be ignored and the proper people would never see it.

The next problem was discovered during the correction of the warnings, because the tool could incorrectly report a fault where the developers incorrectly tried to correct a fault that did not exist. While the majority of the corrections were harmless by most developers, some warnings created almost only harmful corrections. Even though the faulty correction would be detected by testing it added unnecessary cost to the development cycle and creates a risk with using SAT as an early fault detector. But even worse we found seven instances where developers created a new vulnerability when they tried to correct a false positive. That is unacceptable for an early vulnerability detector to create new vulnerabilities in the product. Thankfully, only seven out of 32 developers made this grave error. Similarly, there were developers that could not correct vulnerabilities in such a manner that their implementation was safe. Buffer overflows that were similar to the false positive warning had constantly 9-12 developers that could not correct the code in a safe way. The worst result was shown in a ToC-ToU warning, even developers with security experience had a hard time correcting the warning. The tool can be partly blamed because of its rather obscure warning message.

Given that there were disagreements between developers in classification and different decisions of whether to correct a fault and how one practical approach could be to consult multiple views when making the judgment of how to classify and correct
a fault. This requires more resources, but is more likely to yield accurate results. A similar approach is used in evidence based software engineering where the selection of evidence is done by at least two persons who later on discuss their selection in case of disagreement (cf. (Kitchenham 2004)).

D.6.4 Validity threats and study limitations

Every research design in empirical studies has threats to validity that need to be addressed during the design to reduce, or in the best case, mitigate the threat (Wohlin, Runeson, Höst, Ohlsson, Regnell, and Wesslén 2000). In this section we discuss threats to validity relevant for this study, and actions we have taken in order to reduce the threats.

**Correct Data:** There is a risk that the data collected is not correct, in this case due to wrong classification of faults or lack of trust by the interviewees towards the researchers. If the SAT is used by all developers then security experience cannot be guaranteed and not all developers can determine if a warning really is exploitable or not. Because our initial analysis relied on historical data from developers, it is therefore possible that even more vulnerabilities reside within the code and after being reported by the SAT has been classified as a false positive, an incorrect warning. However, even if our results in Section D.5.1 do not represent the full capacity of the tool they show the result four industry projects managed to achieve with the tool. However, with regard to classification we have taken several measures to increase the correctness of the classification, such as:

- double check classification with security experts in the organization to assure valid data.
- check with senior developer whether corrections done to the code where safe or unsafe corrections.
- practically verify vulnerabilities by writing exploits where needed due to uncertainty.

With respect to trust the presence of the researcher might lead to a reactive bias and hence influence the outcome of the interviews. Given that two of the authors are employed at Ericsson and have been collaborating with the company over a longer period of time this risk is mitigated. Another risk for correct data was that interview questions could be misinterpreted by the interviewee, which is an open threat in the investigation. The threat was reduced by piloting the questionnaire.
Selection of Interviewees: The persons selected impact the outcome of the study. The high number of interviews conducted allowed to cover a variety of opinions, and hence reduced the threat of bias focusing on opinions of individuals.

Generalizability of Results: Case studies are limited in their generalization in comparison to, for example, surveys as they focus on a single case. However, the benefit is that rich descriptions and detailed results can be obtained which is not possible with a population oriented study. In order to aid in the generalizability we took care to carefully describe the context. In particular, the results are relevant for companies developing large-scale products using unsafe programming languages (in this case study the language used was C++).

Knowledge of Defect Population: No results of the defect detection effectiveness can be made, given that the total population of faults and vulnerabilities is unknown and quality assurance practices such as the use of SAT can only show the presence of defects, but not their absence. Hence, we were able to show what the tool is able to detect, but we can not make strong claims about what the tool is unable to detect.

D.7 Conclusions

This paper evaluated the use of automated static analysis for improving software security in an industrial context using case study as a research method. The overall life-cycle of SAT usage was evaluated, considering ability to detect vulnerabilities, strategies for deployment, and usage of SAT to identify and correct vulnerabilities.

Detect vulnerabilities: In the four examined products, static code analysis was able to detect several types of vulnerabilities that would have been more expensive to solve if they had been detected later in development. Common for all products are the memory related vulnerabilities and, in this study, these have been the strength of the tool. Most of the output from the tool is not security oriented and the percentage of interesting warnings shifted from 1.7% to 27%. The amount of false positives also varied between 5% to 25%. The false positives were often of the same type and repeated themselves in code. While examining already corrected code we managed to find known bug reports with the tool. The number of detected bug reports was small both in number and percentage wise of total amount of bug reports. However, a bug report is very expensive and takes a long time to correct while correcting the fault from a static analysis tool is more cost effective. We conclude that for unsafe languages static code analysis identifies vulnerabilities in several different categories, while more design oriented vulnerabilities are not identified. The tool was not providing any security relevant problems for type safe languages (such as Java).

Deployment: The adoption strategy for the tools played a significant role in the end
result. Projects that relied on the tools and its merits fared much worse than the project that forced developers to use the tool. Only 36% of the projects had collected enough data since deployment to merit furtherer studies. The unwillingness of developers to freely start using the tools indicates that developers do not immediately see the benefits. We conclude that a configuration management approach where the tool is integrated in the development process as a mandatory part is the best adoption strategy that is only efficient if developers are educated in order to make use of the tool to correct identified vulnerabilities as soon as possible after detection.

Usage: Our case study that evaluated the usage operation of the tool showed some interesting finding. Classifying warnings can create problems and false positives were often assumed to be correct faults. Only 37.5% of the false positives were correctly classified as such. Security warnings were also often not recognized and instead classified as true positives and 13.7% of the time seen as false positives. However, the incorrect classification of security warnings as true positives does not cause a problem if the correction is still safe. With regard to the correction of warnings some problems were created. Most of the false positives having a correction that added a validation check to prevent the warnings would not had gone away from that correction. Developers that corrected security warnings had problems with very topic specific warnings. In a “file system access” warning only 8 out of 32 developers corrected the warning in such a way that the threat was removed. The other developers could not with the aid of the description figure out how to correctly fix the problem. Because it was a “time of check - time of use” attack most of the unsafe correction just moved the check in the source code close, as in number of lines, to the usage of the file resource. We found that 20.5% of the corrections were done incorrectly and would not solve the threat the warnings was trying to highlight. Even more disturbing were the 16% of harmful corrections of false positives where developers caused more harm to the system when trying to remove warnings.

Developers often complained that they did not understand why the warning was there, even though the warnings do provide an explanation and a data flow in the tools web interface. This had a negative effect on developers’ opinion of the tool and effected the adoption of it. The tool’s false positives created an unforeseen problem where some developers had a tendency to correct false positives and one very bad example seven developers managed to create a new vulnerability when correcting a false positive. The false positive was reported on an input validation that was according to the senior developer “code efficient and smart” but hard to understand for both tool and other developers. As such the tool reported the input validation as a buffer overflow and because other developers did not understand it they opened it up and created a real buffer overflow further in the code. To our knowledge this effect of false positives had not earlier been discussed within an industrial setting in the scientific literature and
there are indications during the interviews that developers have had to revert correction that were done based on warnings because they broke the software. This merits extra cautions when dealing with false positives and make it more important to correctly identify them. We conclude that it is a risk to rely on individual classification and hence use multiple persons to conduct the classification, and later discuss and resolve disagreements. The same holds for taking decisions on how to act on a warning to increase the likelihood that a safe correction is done.

As future work we propose to conduct similar studies in different contexts (e.g. different types of systems, different complexities, different process models) in order to further contribute to the generalizability of the results. Furthermore, comparative studies of static automated analysis and penetration testing are of interest. In particular, investigating whether there is an overlap of detected vulnerabilities would indicate whether one approach can replace the other, or whether they should be used complementary.

D.8 Acknowledgment

The authors want to thank Martin Hylerstedt for proofreading the text. We would also like to thank Ericsson AB for the opportunity and partial funding to perform this study.

D.9 Interview Guide for Deployment

• Facts about usage of Coverity prevent deployment in projects
  – Have you installed and used Coverity on your product (for Java and c++ if you use both)?
  – Frequency of using Coverity prevent, in what time are identified defects corrected (daily/weekly/before release)?
  – Do you have any automated system to report new warnings to developers?
  – Do you use the Coverity GUI?

• Deployment, usefulness and observations
  – How have you deployed coverity prevent?
  – Have you had/observed any problems during the deployment of Coverity? What were they about?
– Do you or your project perceive Coverity as a useful early fault detector? Why/why not?

– Do you have any general feedback of using or deploying Coverity? Explain.

D.10 Questionnaire for Classification of Warnings

Table D.11 shows the questionnaire used for capturing the classification of warnings done by developers. The form includes the options false positive (FP), true positive (TP) and security relevant warning. Only one alternative should be chosen. Furthermore, developers rated their confidence (Conf.) in the judgment they made. The confidence part was not used in this study, but another study using the same data for evaluating the impact of experience on accuracy and confidence in classification (see (Baca, Petersen, Carlsson, and Lundberg 2009)).
### Table D.11: Form for Classification of Warnings.

<table>
<thead>
<tr>
<th>ID</th>
<th>Checker</th>
<th>FP</th>
<th>TP</th>
<th>Sec.</th>
<th>Guess</th>
<th>Not Conf.</th>
<th>Conf.</th>
<th>Very Con.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2753</td>
<td>UNUSEDVALUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2746</td>
<td>UNREACHABLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2744</td>
<td>UNINITCTOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2583</td>
<td>UNINIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2392</td>
<td>UNCAUGHTEXCEPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1790</td>
<td>OVERRUNSTATIC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1765</td>
<td>STRINGOVERFLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1518</td>
<td>SECURETEMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1490</td>
<td>REVERSEINULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1485</td>
<td>STRINGOVERFLOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1482</td>
<td>USEAFTERFREE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1401</td>
<td>TOCTOU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1338</td>
<td>BUFFERSIZEWARNING</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1335</td>
<td>CHECKEDRETURN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1321</td>
<td>UNINITCTOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1291</td>
<td>MISSINGRETURN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1278</td>
<td>UNINIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1170</td>
<td>STACKUSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>721</td>
<td>FORWARDNULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>632</td>
<td>UNINIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>FORWARDNULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>FORWARDNULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### D.11 Example of Developer Classification and Correction of a Warning

The projects that integrated the tool into the development cycle thought the task was easy, however they also believed the tool lacked in IDE integration and would have preferred a solution where the tool’s database would be part of their existing bug tracker instead of its own GUI. The false positive rate was also considered manageable and according to the developers did not hinder the tool’s usage. Several developers expected the tool to find more severe faults than what they perceived the tool actually found.

In this example the SAT detected a resource leak in the source code. For the resource leak to occur the user input had to be constructed in a specific manner. The source code and warnings messages from the SAT are written below in a simplified
Chapter D. Improving Software Security with Static Automated Code Analysis in an Industry Setting

form.

Listing D.1: Required data flow example.

```c++
Event alloc_fn: Called allocation function

"operator new (Com::Message)"

Com::Message aMessage = new Com::Message();

At conditional (1 - n): Takes a false path (several existed)
if (messageType == MMS)
    mmsMessage = aMessage;
    ...

At conditional (2): Takes a true path
if (messageType == CONTROL)
    // Does not store aMessage because
    // type CONTROL should not have a message

At conditional (3): Takes a false path
else
    delete aMessage;

Event leaked_storage: Returned without freeing storage "aMessage"
return;
```

In this example, Listing D.1, we could determine from the historical data that the warning was first classified as a fault that needed correction. Later the warning was classified as a false positive even though it was not only a fault but also a user induced resource leak that could be used as a vulnerability. When examining the source code revision it was observed that the developer did not fully understand or read the warning correctly. The developer had corrected the fault by always freeing the memory before the last statement, in this case the return code. He did not observe that the fault only applied for one possible data flow path and that freeing the memory in any of the other paths was against the intended functionality of the code. As such the correction was removed after function test failed and the warning was labeled as a false positive.

The reported memory leak was also present in the bug tracking system, as an unresolved issue. The bug report stated that a memory leak was present during a message type test case where all different types of message both successful and unsuccessful constructs were executed. The information in the bug report was very vague compared to the SAT showing the direct path required to achieve the memory leak. However,
because the developer did not fully read or understand the warning the fault was never corrected. The knowledge that the tool can report false positive also creates the opportunity for developers to ignore warnings because it is probably a false positive.
Chapter D. Improving Software Security with Static Automated Code Analysis in an Industry Setting
Paper E

Agile Development with Security Engineering Activities

Dejan Baca and Bengt Carlsson

E.1 Introduction

For the last years, large parts of industry have shifted software development from a rigid waterfall to a more flexible Agile software development process. This shift is performed as an attempt to increase the effectiveness of software development. This is done by having a flexible structure with short development increments that handle change and new requirements that are easier than the older process. The Agile processes do impose limitations on the projects, it is no longer possible to create a complete picture of a product as all requirements are not yet known and no attempts are made to acquire this information. As stated in other literature studies (Wayrynen, Bodn, and Bostrm 2004) (Bostrom, Wayrynen, Boden, Beznosov, and Kruchten 2006) (Keramati and Mirian-Hosseinabadi 2008a), that compare security engineering (SE) (Davis 2005) with Agile projects, this lack of a complete overview makes it harder and outright prevents some common SE practices from being performed in an Agile project. In this paper we interview developers that work in an Agile development process, with the basics of three mature SE processes (Win, Scandariato, Buyens, Grgoire, and Joosen 2009). With the interviews as a base we want to identify what parts of the SE processes are most compatible and beneficial to the project and what security activities the developers believe are not possible to perform in an Agile development
The aim of this study is to, through industry experience, identify what practices from mature SE processes are easily integrated and also provide a benefit to Agile projects. A suggested best practices based on the results will present an enhanced Agile development process that will have some of the security measures that SE processes relies on. We are also interested in why developers believe some of the practices do not work in an Agile process and what they believe prevents the integration.

E.2 Related work

Previous works in this topic have focused on literature work and few studies have used industry experience and empirical data for strengthen their results. Hossein Keramati et al. (Keramati and Mirian-Hosseinabadi 2008b) identify and evaluate security practices and activities in two SE process, Microsoft SDL (Howard and Lipner 2006) and Comprehensive Lightweight Application Security Process (Viega 2005). The paper presents an algorithm, called Agility Degree, for rating activities and its compatibility with an Agile process. Mikko Siponen et al. (Siponen, Baskerville, and Kuivalainen 2005) identify the problems with integrating security in agile and devised an agile friendly method to identify, track and implement security features. Their method uses several known security engineering activities but is in the need of more practical experimentation to fine-tune the process. Beznosov & Kruchten (Beznosov and Kruchten 2004) examined mismatches between security assurance techniques and agile development methods. The paper is based on literature studies and the authors identify some techniques that fit well with agile and others that are clear mismatches. Jaana Wäyrynen et al. (Wyrynen, Bodn, and Bostrm 2004) also conducted a theoretical analyses, comparing Common Criteria with Agile and determined they would benefit from more imperial data to asses their results and premises. Laurie Williams et al. (Williams, Meneely, and Shipley 2010b) devised a protection poker planning game that uses developer interactions and common Agile pratices to perform risk analyses during development. In the paper the author performs a case study at Red Hat IT and evaluates its risk analyses method.

E.3 Background

In this section we briefly explain our generalization of the three examined SE process; Cigatel Touchpoints, Common Criteria and Microsoft SDL. The practices of these three processes were extracted by the author and explained as activities that should be per-
formed in different phases of the development process. Based on the recommendation from the SE process each activity is also mapped to a specific phase of process. Four phases where identified as distinct steps, the boarders for these phases become vague in an Agile project but where well documented in the SE process. In order, the project is expected to pass these phases: requirements (Rq), design (D), Implementation (I), Test (T) and Release (R). The same information as in this section was presented to the interviews as explanations what a specific activity meant, what it provided and how it would be integrated into the process.

E.3.1 Cigatel Touchpoints

Cigital Software Security Touchpoints(McGraw 2006b) has often been described as an lightweight SE process that integrates core activities in an existing development process and improves the quality and security aspect of the end product (Steven 2006). From the process we can extrapolate the following activities and claims within each phase:

- **Security Requirements (Rq):** Security Requirements cover both overt functional security and emergent characteristics that are best captured by Abuse Cases and attack patterns.

- **Abuse Cases (Rq):** Abuse Cases describe the system’s behavior under attack; building Abuse Cases requires explicit coverage of what should be protected, from whom, and for how long.

- **Risk Analyses (D):** Security analysts uncover and rank architectural flaws so that mitigation can begin. Disregarding risk analysis at the early phases leads to costly problems down the road.

- **Assumption Documentation (D):** Designers, architects and analysts should clearly document assumptions and identify possible attacks.

- **Static Code Analyses (I):** All software projects produce at least one artifact: source code. At the code level, the focus is on implementation bugs, especially those that static analysis tools, that scan source code for common vulnerabilities, can discover. Code review is a necessary practice, but not sufficient for achieving secure software.

- **Penetration Testing (T):** Penetration Testing provides a good understanding of fielded software in its real environment. It does this by simulating real world working conditions and attack patterns.
Chapter E. Agile Development with Security Engineering Activities

- **Red Team Testing (T):** Manual testing security functionality with standard functional testing techniques.

- **Risk Based Testing (T):** Manual risk-based security testing based on attack patterns.

- **External Review (R):** External analysis (outside the design team) that reviews existing touchpoints and performs their own.

E.3.2 Common Criteria

Common Criteria (Keblawi and Sullivan 2006) is mature and well used SE principal that is ISO certified. A previous study has examine Common Criteria from an Agile perspective and identified activities similarly to ours (Mellado, Fernandez-Medina, and Piattini 2006).

- **Security Requirements (Rq):** Identifying and documenting security and functionality for a given software project.

- **Agree on definitions (Rq):** The first task for the organization is to define the stakeholders and to agree upon a common set of security definitions, along with the definition of the organizational security policies and the security vision of the IS. It is in this activity when the Vision Document artifact is created and should follow one of the available ISO standards.

- **Risk Analyses (D):** Risk must normally be determined from application to application. The final goal to achieve is the 100% risk acceptance. This is captured in the Risk Assessment Document, which is refined in subsequent iterations.

- **Critical Assets (D):** This is where the Security Readiness Review (SRR) is used for the first time. It consists of the identification of the different kinds of valuable or critical assets as well as vulnerable assets by the requirements engineer.

- **UMLSec (D):** Each asset is targeted by threat’s that can prevent the security objective from being achieved. First of all, it is necessary to find all the threats that target these assets with the help of the SRR. In addition, it could be necessary to develop artifacts such as misuse cases or attack trees diagrams or UMLSec use cases and classes or sequence/state diagrams to develop new specific or generic threats or requirements.
• **Requirements Inspection (D):** Requirements Inspection is carried out in order to validate all the generated artifacts and it is generated as a Validation Report. Its aim is to review the quality of the team’s work and deliverables as well as assess the security requirements engineering process.

• **Repository Improvement (R):** The new model elements found throughout the development of the previous activities and which are considered as likely to be used in forthcoming applications and with enough quality, according to the Validation Report, are introduced into the SRR. Furthermore, the model elements already in the repository could be modified in order to improve their quality. Thereby, all these new or modified model elements/artifacts, which have been introduced into the SRR, altogether constitute a baseline.

### E.3.3 Microsoft Security Development Lifecycle Process

The Microsoft Security Development Lifecycle (SDL) (Howard and Lipner 2006) is a software development process used and proposed by Microsoft to reduce software maintenance costs and increase reliability of software concerning software security related bugs. It is based on the classical spiral model but there are attempts in making it more Agile friendly (Sullivan 2008).

• **Security Requirements (Rq):** Identify and enumerating security and privacy functionality for a given software project.

• **Role Matrix (Rq):** Identifying all possible user roles and their access level to the software.

• **Design Requirements (Rq):** Validate the technical design specifications and ensure they are appropriate relative to the Security Requirements for a given software project.

• **Quality Gates (D):** Create appropriate security and privacy quality measures, including activities that need to be done for a fulfillment of requirement.

• **Cost Analysis (D):** Analyses the cost implications of the different possible threats.

• **Threat Modeling (D):** Create new threat models or validate existing threat models for correctness based on the products design. The threat model should identify security vulnerabilities.

• **Attack Surface Reduction (D):** Reduce the design and end products attack surfaces, by limiting entry points and simplifying interfaces.
• **Security Tools (I):** Use commercially available, open source and inhouse developed security tools to assist the project.

• **Coding Rules (I):** Clarify the rationale for the deprecation of unsafe functions and help identify and recommend alternatives for these unsafe functions.

• **Static Analysis (I):** Have mandatory Static Code Analyses with predefined rules and priorities.

• **Dynamic Analysis (T):** Use dynamic testing tools and perform an evaluation, triage the output, explain the results and develop a mitigation strategy for a given software program.

• **Fuzzy Testing (T):** Use fuzzy test tools and perform an evaluation, triage the output, explain the results and develop a mitigation strategy for a given software program.

• **Code Review (T):** Manual source code reviews of risk components.

• **Incident Response Planning (R):** A response checklist that provides clear guidelines of action in the event of a security breach.

• **Final Security Review (R):** A final, before release, security review that; reviews all threat models, validates security tool results, reviews all outstanding/deferred security bugs, reviews all exception requests as part of the security program.

### E.3.4 Other

Besides the three SE processes there are some common knowledge security activity that are often recommended but are not distinct steps in any of the three SE process.

• **Countermeasure Graphs (D):** An risk analyses method that focuses on identifying security features and prioritizing them (Baca and Petersen 2010).

• **Diff. review (I):** Performing source code reviews on patches before the code change is committed to the source code repository (Rigby, German, and Storey 2008).

• **Pair Programming (I):** An Agile concept where developers code in a pairs. Solving and reviewing problems directly as the code is written (Williams, Kessler, Cunningham, and Jeffries 2000).
E.4 Research Method

E.4.1 Case Study Context

As a complement to the process model description, the context of the study was as follows. Ericsson AB is a leading global company offering solutions in the area of telecommunication and multimedia. Such solutions include charging systems for mobile phones, multimedia solutions and network solutions. The company is ISO 9001:2000 certified. The market in which the company operates can be characterized as highly dynamic with high innovation in products and solutions. The development model is market-driven, meaning that the requirements are collected from a large base of potential end-customers without knowing exactly who the customer will be. Furthermore, the market demands highly customized solutions, specifically due to differences in services between countries.

E.4.2 Research Context

The process model used at the company is described and thereafter its principles are mapped to the incremental and iterative development; SCRUM, and Extreme Programming (XP). The model is primarily described to set the context for the case study, but the description also illustrates how a company has implemented an incremental and agile way of working. Due to the introduction of incremental and agile development at the company the following company specific practices have been introduced:

The first principle is to have small teams conducting short projects lasting for three months. The duration of the project determines the number of requirements selected for a requirement package. Each project includes all phases of development, from pre-study to testing. The result of one development project is an increment of the system and projects can be run in parallel.

The packaging of requirements for projects is driven by requirement priorities. Requirements with the highest priorities are selected and packaged to be implemented. Another criterion for the selection of requirements is that they fit well together and thus can be implemented in one coherent project.

If a project is integrated with the previous baseline of the system, a new baseline is created. This is referred to as the latest system version (LSV). Therefore, only one product exists at one point in time, helping to reduce the effort for product maintenance. The LSV can also be considered as a container where the increments developed by the projects (including software and documentation) are put together. On the project level, the goal is to focus on the development of the requirements while the LSV sees the overall system where the results of the projects are integrated. When the LSV phase is
completed, the system is ready to be shipped.

The anatomy plan determines the content of each LSV and the point in time when a LSV is supposed to be completed. It is based on the dependencies between parts of the system developed which are developed in small projects, thus influencing the time-line in which projects have to be executed.

If every release is pushed onto the market, there are too many releases used by customers that need to be supported. In order to avoid this, not every LSV has to be released, but it has to be of sufficient quality to be possible to release to customers. LSVs not released to the customers are referred to as potential releases. The release project in itself is responsible for making the product commercially available, performing any changes required to alter a development version into a release version.

In Figure E.1 an overview of the development process is provided. The requirements packages are created from high priority requirements stored in the repository (1). These requirements packages are implemented in projects resulting in a new increment of the product. Such a project is referred to as a Small Project (3) and has a duration of approximately three months (time-boxed). When a project is finished developing the increment, the increment is integrated with the latest version of the system, referred to as last system version (LSV) (4). The LSV has a predefined cycle (for
example, projects have to drop their components within a specific time frame to the LSV. From the LSV there can be different releases (5) of the system. These are either potential releases or customer releases.

E.4.3 Selection of Interviewees

The interviewees were selected so that the overall development life cycle is covered, from requirements to testing and product packaging. Furthermore, each role in the development process should be represented by at least two persons if possible. The following distinct roles were present:

- **Unit manager**, is responsible for distributing manpower and is responsible for any out of product projects, such as quality or process improvements.

- **Project owner**, has the overall responsibility of the product and has a final say in anything that affects his products projects.

- **Requirements engineers**, write the requirements and are part of the prioritization process.

- **Architects and experts**, solve design issues in the product and also participate in creating the anatomy plan.

- **Developers**, work in the small projects and implement and test their requirements.

- **Testers**, are part of the LSV team and focus only on verification and validation.

When selecting the interviewees we followed this procedure:

1. A complete list of people available for each subsystem was provided by management.

2. For each role of development at least two people were randomly selected for the study. An exception was made for unit manager and project owner as these two roles were occupied by one individual each.

3. The selected interviewees received an e-mail explaining why they have been selected for the study. Furthermore, the mail contained information of the purpose of the study and an invitation for the interview. Overall, 12 persons were contacted and participated in the interview.
E.4.4 Interview Design

The interview was divided into five parts, divided by the different phases of development: requirement, design, implementation, testing and release. The duration of the interviews was set to approximately one hour. Each part was first explained by the interviewer based on the information present in section E.3. The interviewee could then ask more specific questions on subjects he was not familiar with. The interview was designed to collect issues and advantages from the interviewees. In order to collect as many issues as possible, the questions were asked from two perspectives; integrating an activity as part of core loop (cost) and the value of continuously using the activity (benefit). The interviewees had to rate every activity in that phase, to rate an activity the interviewee had 10 points per activity in that phase. He could then distribute them freely between all activities. This voting scheme has the benefit of clearly weighting activity against each other instead of a more traditional rating each activity individuality. Also, the interviewees always had to state what their reason for their answer was and justify their ranking of the activities.

E.4.5 Threats to Validity

Threats to the validity of the outcome of the study are important to consider during the design of the study, allowing actions to be taken mitigating them. Threats to validity in case study research are reported in Yin (Yin 2002). The threats to validity can be divided into four types: construct validity, internal validity, external validity and reliability.

Construct validity: One threat was the selection of people to obtain the appropriate sample for answering the research questions. Therefore, experienced people from the company selected a pool of interviewees as they know the persons and organization best. From this pool the random sample was taken. The selection by the representatives of the company was done having the following aspects in mind: process knowledge, roles, distribution across subsystem components, and having a sufficient number of people involved (although balancing against costs). Furthermore, it was a threat that the presence of the researcher influenced the outcome of the study. The threat was reduced as there has been a long cooperation between the company and university and the author collecting the data is also employed by the company and not viewed as being external. Construct validity was also threatened if interview questions are misunderstood or misinterpreted. To mitigate the threat an initial trial interview was conducted and the interview format was improved to be better understood.

Internal validity: The ratings collected by the subjects were construed with the internal validity in mind. By providing the subjects a fix number of points that they
distribute between the different actives, we assure that the rating is actually comparing these specific activities and not other that some subjects might be aware of depending on their different experience levels.

External validity: The process studied was an adoption of practices from different general process models. Care was taken to draw conclusions and map results to these general models, to draw general conclusions and not solely discussing issues that are present due to the specific instantiation of the process at the studied setting. However, if one maps the general findings in this paper to other development processes their context must be taken into account. Furthermore, a potential threat was that the actual case study was conducted within one company. To minimize the influence of the study being conducted at one company, the objective was to map the findings from the company specific processes and issues to general processes. The characteristics of the context and practices used in the process are made explicit to ease the mapping.

Reliability: There is always a risk that the outcome of the study is affected by the interpretation of the researcher. To mitigate this threat, the study was designed so that data was collected from different sources, i.e. to conduct triangulation to ensure the correctness of the findings. The interviews were recorded and the correct interpretation of the data has been validated through workshops with representatives of the company. The study will also be continued with a practical study of the results. In the process of accepting the practical study the company examined and validated the results before coming to future studies.

E.5 Results

First, project members grading of different security issues are reported and then this result is compared to the function of traditional software engineering processes.

E.5.1 Grading Security Issues

In all 12 professional project workers were asked to grade security issues with respect to requirement, design, implementation, testing and release within a project. First a within-subject analysis of variance, an F-test (Ugarte, Militino, and Arnholt 2008), was done for the different issues comparing costs and benefits. For costs significant differences were found within Requirement (0.001), Design (0.001), Implementation (0.01), Testing (0.05) and Release (0.001), i.e. there is a less than 0.1%/1%/5% chance that the differences are the result of pure chance. For benefit significant differences were found within Requirement (0.001), Design (0.05), Testing (0.1) and Release (0.05). These
differences are further used for comparing the different issues within each step of the Agile development process.

The requirement phase is at the beginning of a project where the wishes of the stakeholder are gathered, analyzed, understood and specified. Figure E.2 indicates that Security Requirement, Role Matrix and Abuse Cases are preferred against the other issues. To be more precise, the visual differences above/below average are confirmed by a Bonferroni-Dunn (Ugarte, Militino, and Arnholt 2008) test showing statistical differences where above average show preferred issues (and below show disliked issues). The significant differences show that one (or more) issues are compared, not all contradicting issues, i.e. this is more of a guideline for further analysis.

The Design phase was explained, to the subjects, as the phase when the requirements are mapped into an architecture. A plan for the rest of the project is also finalized during this phase. Figure E.3 indicates that Assumption Documentation Countermeasure Graphs, Requirement Inspection and Critical Assets are preferred against Threat Modeling and UMLSEC. Remark that all statistical differences originate from the cost comparison, i.e. it seems more difficult to compare benefits during the design phase. For the Critical Assets issue the results are contradictory with benefit just outside the (negative) 10% significant range.

During the implementation phase, see Figure E.4, developers use the requirements and plan to write the source code and create the end product. Ericsson uses test driven development and as such does basic testing and test implementation during this phase. However, none of the examined SE processes have any specific test-driven activities.

Figure E.2: Above/below zero average for different requirements measured as cost and benefit.
Figure E.3: Above/below zero average for different design issues measured as cost and benefit.

Figure E.4: Above/below zero average for different implementations measured as cost and benefit.

during this phase. For cost Static Code Analyses and Coding Rules are preferred against Security Tools. While for benefits, no over-all significant differences (F-test) were found which was also confirmed when the issues were individually examined.

Testing includes all verification steps that are performed by the LSV test team. This does not include any basic testing the SP development teams perform as part of their small project time-line. The testing phase can be seen at the bottom of Figure E.1. In
Figure E.5: Above/below zero average for different testing issues measured as cost and benefit.

Figure E.5 Test Plan is preferred against Fuzzy Testing from a cost perspective, while Dynamic Analyses is preferred against Test Plan from the benefit point of view. So Test Plan is both preferred (cost) and disliked (benefit), i.e. it is cheap to introduce but gives little pay back. Obviously this is not a preferable issue for further consideration. Risk Based Testing is just outside the 10% scope for benefit making it a candidate for preferred issues.

Figure E.6: Above/below zero average for different release issues measured as cost and benefit.
In the release phase the product is completed for public access. Here, see Figure E.6, Repository Improvement is better than all the other methods and contrary External Review is worse than all other for cost. For benefit Repository Improvement is better than Incident Response Planning.

E.5.2 Feedback Discussion

Using the interviews as a base we can create the framework of a new SE process that has the potential of being Agile friendly and create secure software. It is therefore important to examine what benefit the security provide and what is excluded from the process and why.

Writing Security Requirements was a strongly endorsed activity for integrating both cost and benefit. Developers believed that Security Requirements would identify what the easy gains are and guide the project right. Both Role Matrixes and Abuse Cases scored high and provided unique benefits to the process. Developers did however have a reservation with Role matrixes, some raised a concern that during the requirement phase it is still too early to have a grasp what roles will be in the end product. This concern applied more to new products then already mature projects. Developers also suggested that Abuse Cases should not be written during the requirement phase but instead by the SP development teams. The remaining activities were deemed non-useful or to intrusive. Cost Analyses was the lowers ranked activity and developers believed it to not provide useful information and require decisions that are not yet made in an Agile project at the requirements phase. They also believed that nobody working with requirements would have the necessary knowledge to perform a cost analyses.

In the design phase we presented three different threat analyses methods, while the overall goal is the same. It should be noted that the Countermeasure Graphs were designed especially for Agile projects and in the same context as the interviewed developers. As such, the threat analyses value might be skewed. Threat modeling and UMLSec were both deemed too hard to implement. The last step in the Threat modeling activity, identifying vulnerabilities, was seen as to hard or impossible in this phase by most developers. Compared to countermeasure graphs were the last analyses part is to identify how the product could or is directly stopping the attack. Because of the lack of a finished design, even after the design phase, developers believed that UMLSec would not be possible to perform or it would hinder the project too much. Attack surface reduction was seen as an effective concept but several developers raised concerns with the added complexity and cost of encapsulating and reducing the surface area at this stage of software development and instead advocated an ad-hoc solution where developers would raise the issue of attack surface reduction during stand-ups if it was required.
During implementation there were concerns that even Static Code Analyses do not provide any large benefit as few of the warnings reported by the Static Code Analyses tools actually produce any real bug reports. Previous studies (?) in the same company have shown that as low as 9% of the warnings can result into a bug report and even less are security related. Specialized security tools were seen as hard to integrate and also against the intentions of the Agile manifesto. Similarly Diff reviews where seen as hampering the projects flow or not providing any benefit. Diff reviews outside the team were seen as intrusive and cumbersome as the outside review is not familiar with the code. So, Diff reviews should be done within the team familiar with the code but developers now instead believed the code review to be ineffective as both reviewers have worked with the code, i.e. blind to defects in ones own code. An third alternative, the open source way, submitting patches to a component owner was dismissed because it would severely hamper the Agile flow. Pair programming, which is not a SE activity but instead belongs to SCRUM activities, had mixed responses.

Looking at the interview answers supplement the understanding of the main problem involved with testing. Red Team Testing was seen as too time intensive and would remove hours from the important LSV testing were the end functionality is validated and verified. Penetration Testing was hard to do because the project would not have a reliable environment to test the product in. This might be more specific to Telecom products because they require a more complex environment to operate compared to regular datacom products. Risk Based Testing made developers skeptic to its requirement of a risk analyses that most likely had been performed before the finished product had been created and which might have changed in design since. For fuzzy testing the developers were worried about the large number of test results and verification that would be needed after the automated fuzzing.

During the release phase it is clear that only Repository Improvements or lessons learned fitted well with the process. This is not surprising as the development process already implements this step for the general success of the development project. The other activities were not received well; especially the External security review was heavily criticized for interfering with the development process. The idea to not ship a release before an external party has reviewed the work is in harsh contrast to an Agile, quick communicating team.

### E.5.3 Security Engineering and Agile Comparison

All tools or testing issues described above are collected from well known processes from Microsoft SDL (M), Cigatel touchpoints (T), Common Criteria (CC) and in a few
cases from other processes used in the company environment (O). Those traditional software engineering processes mainly concern large waterfall projects instead of the today more common agile projects. In Table E.1 significant differences in all test cases are described where + and - are positive and negative respectively. A + or - means any significance (0.001, 0.01 or 0.05) above or below zero average, i.e. the issue is significant different to at least one other issue within the investigated phase. Also, the comparison is based on a relative measurement separating one issue from other issues. If no significance is present among the involved issues, as in most benefit requirements, it is hard to draw any conclusions at all about the involved issues. Therefore, our conclusions mostly concern the cost aspects of the agile process. This is probably due to the interviewed subjects, because they work in a agile environment they are capable in estimating the cost of using an activity. However, because they are not security experts their knowledge in security is more diverse and therefore their answer divert so much that no answer is significant.

E.6 Discussion

In the discussion we will first compare how the developers believed the existing SE processes could integrate with Agile. This is followed by our own suggested processes that is based on the developers recommendations and our own judgment of required activities.

E.6.1 Evaluating Software Engineering Processes

Microsoft’s SDL had the largest negative effect on an Agile development process. Developers where particularly negative to Cost analyses and the purpose of a Cost analyses is to focus the security push on the area that creates potentially the greatest cost and damage. However, developers did not believe that it would be possible to perform a sound Cost analyses at that early stage of development and that the benefit is no longer as good at the later phases of development. The lack of a Cost analyses is partly mitigated by having a more complex Risk Analyses and by the short iterations of a Agile process. Even if a development project has the wrong focus the sprint is short and the focus can be realigned after feedback from early customers or from retrospect meetings. Microsoft’s Threat modeling (a form of Risk Analyses) was seen as too costly and a more solution based analyses was recommended instead. The concept of writing specific secure tools is a direct contradiction to the Agile manifesto (?) that states that collaboration and small teams should be prioritized instead of advanced
### Chapter E. Agile Development with Security Engineering Activities

#### Requirement

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Cost</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security requirm. (T,M,CC)</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Role Matrix (M)</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Abuse Cases (T)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Design requirem. (M)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Quality gates (M)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Cost analyses (M)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Agree on definitions (CC)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Design

| Assumpt. Document. (T)                      | +    |
| UMLSec (CC)                                | -    |
| Requirem. Inspec. (CC)                     | +    |

#### Implementation

| Static Code Analyses (T, M)                | +    |
| Coding Rules (M)                          | +    |
| Security tools (M)                        | -    |

#### Testing

| Test plan (T)                              | +    |
| Fuzzy Testing (M)                         | -    |

#### Release

| External Review (T)                       | -    |
| Incident resp. planning (M)               | +    |
| Repository improvem. (CC)                 | +    |

| **Table E.1:** Grading security issues within a project cycle. M - Microsoft SDL, T - Cigatel touchpoints, CC - Common Criteria and O - Others |

During testing the Dynamic analyses was seen as favorable. However, both fuzzy testing and a specific component code review where seen as costly too implement in the process. The code review is not as important for Agile teams because they collaborate more and have informal code reviews and feedback during daily stand-ups. The fuzzy testing does not have any equal improvement in the Agile process. Even though it is
seen as a hamper to the process, projects will have to decide by them selves if Fuzzy Testing is an option. Lastly, Microsofts Incident response planning was not seen as a hinder but according to the developers it lacked the benefit for the work required.

Cigatel's Touchpoints did not have any negative activities during requirement, design or implementation phase. The basic Risk Analyses recommend by Cigatel can instead be replaced with a more Agile friendly alternative like Countermeaure Graphs (Baca and Petersen 2010). However, during testing there are several test methods that raise the cost without providing a large benefit. The project in this study relies on test driven development and a large suit of automated tests for regression. The new automated test cases are also written at the same time as the implementation code. The Touchpoints testing methods all relies on having a ready product and then conduct a, mostly manual, test phase where the implementation code is not changed. This requirement by the tests methods makes them very Agile unfriendly, more specifically test driven unfriendly. An alternative would be to adapt or device a new test method that has the same benefit but uses test-driven and a large automated test suit instead of manual after release test methods. Lastly the concept of having an external, out of project, review is very hostile to a process that is depended on no bottlenecks or other blocking issues. An Agile retrospective meeting can in some sense replace a review but does not fulfill the requirement of being independent, e.g. external.

Common Criteria that is more of a metrics and measurements scheme does not have any implementation or testing activities. However, its activities for the products architecture assumes that there is an initial and final design created and then resulting in an implementation phase. For Agile projects the design phase if often very short and any design problems are solved during implementation by the team in a group effort. The group collaboration and daily stand-ups could replace the concept of a final architectural design.

E.6.2 An Agile Security Process

If we combine the most compatible and beneficial activities, shown in Table 2, with an Agile process we can create an Agile security process that implements the most cost effective benefits from the three SE process. As we have discussed above some activities are not present at all but we have shown how an Agile process can cope without this activities but still have a high security standard. To better integrate the SE activities we have to move them from there recommended phase and placed them with the development team so the activity can be performed during a work package. Abuse Cases where moved to the design phase and the coding developers should write them instead of the requirement engineers. Also the release phase, Repository Improvement fits very well with the retrospect meeting that development team perform at the end
Chapter E. Agile Development with Security Engineering Activities

<table>
<thead>
<tr>
<th>Product Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
</tr>
<tr>
<td>Security Requirements</td>
</tr>
<tr>
<td>SP Development Team</td>
</tr>
<tr>
<td>Design</td>
</tr>
<tr>
<td>Countermeasure graphs</td>
</tr>
<tr>
<td>Assumption Documentation</td>
</tr>
<tr>
<td>Abuse Cases</td>
</tr>
<tr>
<td>Requirement Inspection</td>
</tr>
<tr>
<td>Repository Improvement (retrospect meeting)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSV Test Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
</tr>
<tr>
<td>Dynamic analyses</td>
</tr>
</tbody>
</table>

Table E.2: Agile Security a basic Agile process with the most compatible SE activities.

of a work sprint. Integrating these two activities is made easier by moving them to a more developer heavy phase. An Agile process is a small team process that should focus its efforts on the Small Project teams and not have heavy pre- or post-coding phases. Comparing to Figure E.1 most of the activities and security focus will be done by the development teams in step 3 of the process, the small project teams. During requirement the security enhanced Agile process primary focus is to write specific requirements for security goals, these will aid both programmers and tests to know what goals are required and need verification. A Role matrix aids the writing of user stories that programmers use to write the code. Having better security specified user stories makes it easier to write design safe software. The main focus for the improvements are with the developers teams and their work sprints. The design phase has several small planning and analyses steps that together create better security, while implementation focuses on automated tools that provide quick but rough code reviews. During the design phase the Agile process creates user stories and use cases to keep track of planned activities. Similarly Abuse Cases could be written for scenarios that may not happen. Developers can then keep track on how they prevent the scenario or verify with basic test cases that it is not possible. The testing phase is performed by the LSV team and
consists of other developers then the person that have written the code. Most of the
test methods recommended by the SE process focus on manual testing. As such only
the dynamic analyses scored high, since it does not require manual testing and uses the
automated test suit.

E.7 Conclusion

Project developers from Ericsson AB, a global company offering telecom solutions,
were asked to grade security improvement activities with respect to requirement, de-
sign, implementation, testing and release within a project using agile development
methods. These security activities were chosen from three leading software develop-
ment processes (Cigatel Touchpoints, Common Criteria and Microsoft SDL) in order to
compare traditional large scale waterfall development security benefits with a smaller,
more flexible, agile software development process.

All three traditional processes scaled badly to the chosen agile security processes
because of too high costs or not being enough beneficial for the agile conditions, i.e.
time constraints and reiterated increments of the product. Preliminary findings, based
on the interviews, show that especially the design and testing phase scaled badly. In-
stead a new agile security process is proposed, integrating selected activities from the
other traditional security development processes mainly into the development team but
also to product owner and test team. In a future work these proposed security processes
will be integrated and tested in a practical experiment with the existing development
process.
Paper F

Integrating security engineering in an agile process

Dejan Baca and Bengt Carlsson

F.1 Introduction

Going back a few years, the process of developing software was a rigid, costly and slow process involving large manual resources and resulting in a definite, fixed version of the software, i.e. it was only with difficulty possible to generate updated versions. Often a sequential design process, a waterfall model, was used involving requirements, design, implementation, testing and release. From a security point of view the waterfall model implies an overall control of involved security issues before software release, i.e. by using processes like Cigatel Touchpoints, Common Criteria and Microsoft SDL. The drawback is both a general problem, requirements and limitations cannot be entirely known before completion and a more practical one, some of the software industry does not use these long-term processes any more.

Today large parts of the industry have shifted software development from a former waterfall model to a more flexible agile software development process. These shifts are performed as an attempt to increase the effectiveness of software development and are also a result of more online distributed software and platforms. A key aspect of agile software is a flexible structure with short development increments that handle change and new requirements easier than the older process. The agile processes do impose limitations on the projects, it is no longer possible to create a complete picture of a
product as all requirements are not yet known and no attempts are made to acquire this information.

As stated in other literature studies (Wyrynen, Bodn, and Boström 2004) (Bostrom, Wayrynen, Boden, Beznosov, and Kruchten 2006) (Keramati and Mirian-Hosseinabadi 2008a), that compare security engineering (SE) (Davis 2005) with agile projects, the lack of a complete overview makes it harder and outright prevents some common SE practices from being performed in an agile project. So, SE tools commonly used in the waterfall model are partly neither tested nor plausible in an agile industrial setting. After referring related work and describing the SE processes Cigatell Touchpoints, Common Criteria and Microsoft SDL we interview developers that work in an agile development process, with the basics of these SE processes (Win, Scandariato, Buyens, Grgoire, and Joosen 2009). With the interviews as a base we identified what parts of the SE processes are most compatible and beneficial to the project and what security activities the developers believe are not possible to perform in an agile development process.

A suggested best practice, based on the results, presents an enhanced agile development process, i.e. both security measures that SE processes relies on and practices that do not work in an agile process. This represents a model for a new agile SE process, which is further investigated in a real industrial study including several ongoing projects using the best practice experiences. The aim of this study is to, through industry experience, identify what practices from mature SE processes are easily integrated and also provide a benefit to agile projects.

F.2 Related work

Previous works in this topic have focused on literature work and few studies have used industry experience and empirical data to strengthen their results. Hossein Keramati et al. (Keramati and Mirian-Hosseinabadi 2008b) identify and evaluate security practices and activities in two SE process, Microsoft SDL (Howard and Lipner 2006) and Comprehensive Lightweight Application Security Process (Viega 2005). The paper presents an algorithm, called Agility Degree, for rating activities and its compatibility with an agile process. Mikko Siponen et al. (Siponen, Baskerville, and Kuivalainen 2005) identify the problems with integrating security in an agile setting and devised an agile friendly method to identify, track and implement security features. Their method uses several known security engineering activities but is in the need of more practical experimentation to fine-tune the process. Beznosov & Kruchten (Beznosov and Kruchten 2004) examined mismatches between security assurance techniques and agile development methods. The paper is based on literature studies and the authors
identify some techniques that fit well with agile and others that are clear mismatches. Jaana Wäyrynen et al. (Wyrynen, Bodn, and Bostrm 2004) also conducted a theoretical analysis, comparing Common Criteria with agile and determined that they would benefit from more empirical data to assess their results and premises. Laurie Williams et al. (Williams, Meneely, and Shipley 2010b) devised a protection poker planning game that uses developer interactions and common agile practices to perform risk analyses during development. In the paper the authors perform a case study at Red Hat IT and evaluates its risk analyses method.

F.3 Background

In this section we briefly explain our generalization of the three examined SE process; Cigatel Touchpoints, Common Criteria and Microsoft SDL. The practices of these three processes were extracted by the authors and explained as activities that should be performed in different phases of the development process. Based on the recommendation from the SE process each activity is also mapped to a specific phase of process. Five phases where identified as distinct steps, the borders for these phases become vague in an agile project but were well documented in the SE process. In order, the project is expected to pass these phases: Requirements, Design, Implementation, Test and Release. Some activities might exist in two different phases because the different SE processes did not agree when the activity should be performed. The same information as in this section was presented to the interviewed as explanations what a specific activity meant, what it provided and how it would be integrated into the process.

Cigital Software Security Touchpoints(McGraw 2006b) has often been described as a lightweight SE process that integrates core activities in an existing development process and improves the quality and security aspect of the end product (Steven 2006). Common Criteria (Keblawi and Sullivan 2006) is mature and well used SE principal that is ISO certified. A previous study has examined Common Criteria from an agile perspective and identified activities similarly to ours (Mellado, Fernandez-Medina, and Piattini 2006).

The Microsoft Security Development Lifecycle (SDL) (Howard and Lipner 2006) is a software development process used and proposed by Microsoft to reduce software maintenance costs and increase reliability of software concerning software security related bugs. It is based on the classical waterfall model but there are attempts in making it more agile friendly (Sullivan 2008).

Besides the three SE processes there are some common knowledge security activity that are often recommended but are not distinct steps in any of the three SE processes.

When naming activities we have first used any agile process activity named that
Chapter F. Integrating security engineering in an agile process

matches the security engineering activity. If none existed we use the most common security engineering name, if similar tasks had the same name but differentiated in their execution we used different names to keep them apart.

F.3.1 Requirement

- Security Requirement: Security Requirements cover both overt functional security and emergent characteristics that are best captured by Abuse Cases and attack patterns. Identifying and documenting security and functionality for a given software project.

- Abuse Cases: Abuse Cases describe the system’s behavior under attack; building Abuse Cases requires explicit coverage of what should be protected, from whom, and for how long.

- Agree on Definitions: The first task for the organization is to define the stakeholders and to agree upon a common set of security definitions, along with the definition of the organizational security policies and the security vision of the IS. It is in this activity when the Vision Document artifact is created and should follow one of the available ISO standards.

- Role Matrix: Identifying all possible user roles and their access level to the software.

- Design Requirements: Validate the technical design specifications and ensure they are appropriate relative to the Security Requirements for a given software project.

- Cost Analysis: Analyses the cost implications of the different possible threats.

F.3.2 Design

- Risk Analyses: Security analyses uncover and rank architectural flaws so that mitigation can begin. Disregarding risk analysis at the early phases leads to costly problems down the road. Risk must normally be determined from application to application.

- Assumption Documentation: Designers, architects and analysts should clearly document assumptions and identify possible attacks.
• Critical Assets: The creation of a document that consists of the identification of the different kinds of valuable or critical assets as well as vulnerable assets by the requirements engineer.

• Identify Threats and Risks in Requirements: Each asset is targeted by threats that can prevent the security objective from being achieved. First of all, it is necessary to find all the threats that target these assets. In addition, it could be necessary to develop artifacts such as misuse cases or attack trees diagrams or use cases and classes or sequence/state diagrams to develop new specific or generic threats or requirements.

• Requirements Inspection: Requirements Inspection is carried out in order to validate all the generated artifacts and it is generated as a Validation Report. Its aim is to review the quality of the team’s work and deliverables as well as assess the security requirements engineering process.

• Quality Gates: Create appropriate security and privacy quality measures, including activities that need to be done for a fulfillment of requirement and allow the process to proceed to the next step in development.

• Threat Modeling: Create new threat models or validate existing threat models for correctness based on the products design. The threat model should identify security vulnerabilities.

• Attack Surface Reduction: Reduce the design and end products attack surfaces, by limiting entry points and simplifying interfaces.

• Countermeasure Graphs: A risk analysis method that focuses on identifying security features and prioritizing them (Baca and Petersen 2010).

F.3.3 Implementation

• Static Code Analyses: All software projects produce at least one artifact: source code. At the code level, the focus is on implementation bugs, especially those that static analysis tools, that scan source code for common vulnerabilities, can discover. A process should have mandatory Static Code Analyses with predefined rules and priorities.

• Security Tools: Use commercially available, open source and in-house developed Security Tools to assist the project.
Chapter F. Integrating security engineering in an agile process

- Coding Rules: Clarify the rationale for the deprecation of unsafe functions and help identify and recommend alternatives for these unsafe functions.

- Code Review: Performing source Code Reviews on source code before the code change is committed to the source code repository (Rigby, German, and Storey 2008).

- Pair Programming: An agile concept where developers code in pairs. Solving and reviewing problems directly as the code is written (Williams, Kessler, Cunningham, and Jeffries 2000).

F.3.4 Test

- Penetration Testing: Penetration Testing provides a good understanding of fielded software in its real environment. It does this by simulating real world working conditions and attack patterns.

- Red Team Testing: Manual testing of security functionality that is performed by an outside team. Similar to Penetration Testing but instead uses a external bad guys team that attacks the product.

- Risk Based Testing: Manual risk-based security testing based on attack patterns. The risk analysis is used to guide and focus the test cases towards the most important parts of the product.

- Dynamic Analysis: Use dynamic testing tools and perform an evaluation, triage the output, explain the results and develop a mitigation strategy for a given software program.

- Fuzzy Testing: Use fuzzy test tools and perform an evaluation, prioritize and sort the output, explain the results and develop a mitigation strategy for a given software program.

- External Code Review: Manual source Code Reviews of specific components that is performed by an outside team, usually a verification and validation team.

F.3.5 Release

- External Review: External analysis (outside the development team) that reviews existing touchpoints and performs their own.
• Retrospective (Repository Improvement): The new model elements found throughout the development of the previous activities and which are considered as likely to be used in forthcoming applications and with enough quality, according to the Validation Report, are introduced into the SRR.

• Incident Response Planning: A response checklist that provides clear guidelines of action in the event of a security breach.

• Final Security Review: A final, before release, security review that; reviews all threat models, validates security tool results, reviews all outstanding/deferred security bugs, reviews all exception requests as part of the security program. Furthermore, the model elements already in the repository could be modified in order to improve their quality. Thereby, all these new or modified model elements/artifacts, which have been introduced into the SRR, altogether constitute a baseline.

F.4 Research Method

F.4.1 Case Study Context

As a complement to the process model description, the context of the study was as follows. Ericsson AB is a leading global company offering solutions in the area of telecommunication and multimedia. Such solutions include billing systems for mobile phones, multimedia solutions and network solutions. The company is ISO 9001:2000 certified. The market in which the company operates can be characterized as highly dynamic with high innovation in products and solutions. The development model is market-driven, meaning that the requirements are collected from a large base of potential end-customers without knowing exactly who the customer will be. Furthermore, the market demands highly customized solutions, specifically due to differences in services between countries.

F.4.2 Research Context

The process model used at the company is described and thereafter its principles are mapped to the incremental and iterative development; SCRUM, and Extreme Programming (XP). The model is primarily described to set the context for the case study, but the description also illustrates how a company has implemented an incremental and agile way of working. Due to the introduction of incremental and agile development at the company the following company specific practices have been introduced:
The first principle is to have small teams conducting short projects lasting for at most three months. The duration of the project determines the number of requirements selected for a requirement package. Each project includes all phases of development, from pre-study to testing. The result of one development project is an increment of the system and projects can be run in parallel.

The packaging of requirements for projects is driven by requirement priorities. Requirements with the highest priorities are selected and packaged to be implemented. Another criterion for the selection of requirements is that they fit well together and thus can be implemented in one coherent project.

If a project is integrated with the previous baseline of the system, a new baseline is created. This is referred to as the latest system version (LSV). Therefore, only one product exists at one point in time, helping to reduce the effort for product maintenance. The LSV can also be considered as a container where the increments developed by the projects (including software and documentation) are put together. On the project level, the goal is to focus on the development of the requirements while the LSV sees the overall system where the results of the projects are integrated. When the LSV phase is completed, the system is ready to be shipped.

The anatomy plan determines the content of each LSV and the point in time when a LSV is supposed to be completed. It is based on the dependencies between parts of the system developed which are developed in development projects, thus influencing the time-line in which projects have to be executed.

If every release is pushed onto the market, there are too many releases used by customers that need to be supported. In order to avoid this, not every LSV has to be released, but it has to be of sufficient quality for a potential customer release. The release project in itself is responsible for making the product commercially available, performing any changes required to alter a development version into a release version.

In figure F.1 an overview of the development process is provided. The requirements packages are created from high priority requirements stored in the repository. These requirements packages are implemented in projects resulting in a new increment of the product. Such a project is referred to as a Development Project and has a duration of approximately three weeks (time-boxed). Each DP contains design, implementation and testing of the requirements. It might happen that a DP is started in one sprint to later be released in another. When a project is finished developing the increment, the increment is integrated with the latest version of the system, referred to as last system version (LSV). The LSV has a predefined cycle and no components dropped after a new sprint will wait until the next sprint to be tested. From the LSV there can be different releases of the system. These are either potential releases or customer releases.
Figure F.1: An overview model of the agile processes used by the examined projects. Three distinct roles (SPM, DP and LSV) are relevant to our study and shown in the model.

**F.4.3 Case Study**

Our study has been divided into four distinct steps; first we examined available literature and extracted practical activities from existing mature security engineering processes. As a second step we conducted a static validation with developers to determine how well the activities integrate in an agile process. Afterwards we perform a dynamic validation by allowing developers in ongoing projects to use the activities. Lastly the suggested improvements, of the examined agile process, are presented to the develop-
ers and any feedback is incorporate into the process.

**Security engineering practices**

Current literature and research for the five most common security engineering processes were used to extract all practical activities that are performed while using the processes. If a specific tool or method was recommended by the process the activity was generalized to make our process vendor neutral, e.g. if an established process recommended a specific static analysis tool we are generalizing that recommendation to use any static analysis tool.

The results of literature study can be found in section F.3.

**Developers’ activity selection**

Since the activities from the literature study are both numerous, sometimes contradictory and in some processes the same activity can be recommended on different phases of development, we performed an initial static validation with the developers. Based on the available literature the developers were introduced to the different activities and their benefits. They were then asked to weight the different activities to each other. Two weightings were done, first the activities compatibility based on how well the activity would integrate into the existing agile process (compatibility) and the activities benefit based on how effective they perceived the activity would be (benefit). For the weighting process each developer was assigned 10 points per activity that he could distribute between the activities. This method was used because we wanted developers to weight the activities between each other and because the method guides developers in making a decision (Berander and Svahnberg 2009b). From this initial validation we could exclude those activities that did not provide enough of a benefit for the cost of integrating it in the process. The full results and method of this step can be read in the paper agile development with security engineering activities (Baca and Carlsson 2011).

The results relevant to this study are presented in section F.5.

**Practical study**

To practically verify the different activities, compatibility and perceived effectiveness we observed several development teams while they performed a short development project (see figure F.1) and at the same time performed 3-4 randomly selected activities. Because we did not want to overburden any team and at the same time make sure many of the voluntary teams performed their tasks, we limited the number of assigned tasks so that a developer would spend 1 man hour per week on the study. The teams
were selected by management and they were all introduced to their security activities just before they started a new development project. During this introduction the developers learned about the activity, how to perform it, why it is done and what the current best practices are. The developers were however free to perform the activity in any way they seemed fit. It was also determined how they would measure the success factor of the activity and how they would report on its usage to management or if there was any way of creating a metric. The practical study ran for the entire development project while developers were observed, one of the authors participated in the stand-up meetings each morning and noted any discussions or problems with the activities but avoided to provide input for performing or analysing the results until after the study was completed.

The results from the practical study with observations and developer feedback is in section F.6.

Final validation

The results from the previous studies was used by the authors to create an agile process that incorporates the most beneficial concepts from security engineering. As a final step the new revamped agile process, with security activities, was presented to a group of agile processes experts and project managers. In this presentation a practical implementation of the process was suggested and feedback from the participants was used to create the final version of the process.

F.4.4 Threats to validity

While presenting the benefit and operations of security activities we limited our information and answers to questions to only the information present in literature, this was important to prevent any author bias from the activities. Equally the goals and success of an activity was determined by the developers and not by the author. For the same reason developers were used in the final static validation.

F.5 Evaluating waterfall activities

In total 12 developers were interviewed and the identified security activities were weighted by them. By adding the developers’ weighting for effectiveness and ease of use for each activity we produced an acceptance rating. Using the acceptance rating and feedback from the developers we decided which activities merited a practical study.
The figures in this section also show what activities ended up being used in the practical study. The activities in the "Original" column were used without any major changes to them. The "Modified" column shows activities that were either joined together with other activities for a new altered activity or they were moved from their intended phase of development to a more suitable phase in the agile process that might not be ideal for effectiveness. Lastly, the "Not Used" column are activities that were excluded.

Three requirement activities had a higher than average acceptance, see figure F.2. Security Requirements as concept was deemed as both compatible and effective. While instead of writing a specific Role Matrix it was suggested to instead write user stories with distinct roles, fulfilling the same goal. The entire Abuse Case procedure, as explained in "Building security in" (McGraw 2006b), was rejected as too time consuming and restricted. Instead a part of the procedure, the creation of misuse or abuse cases by DP projects, was proposed and accepted for future study.

Quality Gates had below average acceptance but was seen as very important by some developers. There was a consensus that no quality gates should be used within a DP project but instead checked for what was an acceptable requirement and release.

Three activities were directly rejected by the rating and the developers; Design Requirements were not seen as feasible because the design is unsure or non-existing
at this stage; to perform a design inspection by the SPM’s when the DP project has started was deemed as too intrusive for the process. Instead the DP project should use the planning game together with the SPM to iron out any design questions, as is already done in agile processes. With the exceptions of requirements that are based on standard or certification, developers deemed that agreeing on definitions (standards) as non-useful for the product’s security. They also thought non requirement standards should be decided by the programmers and not during requirement. Having a separate Cost Analyses for the requirements was seen as a step of a risk analysis and nothing that should be done as a separate task.

Figure F.3: Developers comparative weighting on design phase security activities. Divided into three groups depending on how the activities were later used in the practical study. See section F.4.3 for clarification of the weighting process.

For the design activities only one was immediately dismissed. However, several were combined or considered already part of the agile process. Requirements Inspection was considered a natural step in the planning game by the DP team. The concept of a pre implementation Assumption Documentation was rejected. However, an agile alternative was suggested where the DP team would document their assumptions during implementation and then deliver them at the end of the sprint. Risk analyses, both Countermeasure Graph and others, were deemed necessary to split up. So that there is a general requirements based risk analysis by the SPM team followed by a detailed specific risk analysis for the DP team that only focuses on the user stories they have. Critical Assets, as an own activity was questioned for its usefulness; instead an assets identification should be part of any risk analysis. Microsoft’s specific risk anal-

163
ysis, Threat Modeling, was seen as too heavy, especially with the recommendation of performing it every sprint. Instead the developers preferred the countermeasure graph. Having a clear outcome to Identify Threats and Risks in Requirements for the security review was seen as unrealistic as it will prevent the possibility of change and adaptability; as such the Identify Threats and Risks in Requirements activity was removed.

**Figure F.4:** Developers comparative weighting on implementation phase security activities. Divided into three groups depending on how the activities were later used in the practical study. See section F.4.3 for clarification of the weighting process.

Static code analyses were seen as a good alternative to Code Reviews if it could be built into the continuous integration process. Another requirement was that it was properly configured to avoid warning spam and false positives. Having strict predefined Coding Rules and training in them was considered as very important to improve security. Pair Programming developers should be able to identify risks easier. However, some were reserved about the added cost.

A general Code Review was rejected. Instead, if a risk analysis could identify the most dangerous requirements, then specific targeted Risk Based Code Review could be performed on that code alone instead of all the altered code. It was considered unlikely for developers to create their own Security Tools to test or verify a security feature. It was also agile unfriendly as it builds reliant on tools instead of user interaction.

Both DP and LSV are responsible for different phases of testing. The dynamic analysis tool was placed in the test phase by the different SE processes because the tools required a running binary, something that sometimes cannot be guaranteed during implementation. The developers did not have any strong agreement on the security testing activities and therefore the activities in figure F.5 are of similar compatibility.
as with the agile process. However, while the Test Plan was easy to perform it was not considered to have any impact on security and prevented flexibility in case it has to be followed to the letter. There was also great concern about fuzzy tools’ ability to be part of a continuous integration process. Developers were worried about the large amounts of output such a tool could produce and would prefer to run the tool manually by a specific security team instead of LSV. The weakest activity Code Reviews was considered out of phase and that the time could be spent on testing instead and that any Code Review should be done by the DP before they start testing.

For the release of the software the developers pointed out that a potential release is created after each sprint, as such it would be impossible to have any time consuming activity during release, i.e. the process breaks agile compatibility and creates internal and external release that differentiate processes. That would mitigate the point of having short sprints with working software from which the customer can demo and receive updates quickly. As seen in figure F.6 one activity, Repository improvement was highly recommended. The developers believed that it could be integrated in the existing retrospective meetings. While External security reviews were completely incompatible and hampered the process very negatively and probably would double the time of a sprint release.
Chapter F. Integrating security engineering in an agile process

Figure F.6: Developers comparative weighting on release phase security activities. Divided into three groups depending on how the activities were later used in the practical study. See section F.4.3 for clarification of the weighting process.

F.6 Integrating with agile

Using the results from section F.5 we performed a practical study that was used to examine how developers used the security activities in an agile setting. The figures in this section shows how the different activities were eventually integrated into the process. The results in this section are based on feedback from the developers that performed the study and observations on how they used and integrated the activities.

18 teams performed group of the activities as part of their daily work. On average each team had 7 developers and performed 2-4 activities. 9 teams did not have a requirement to perform the activities but instead viewed them as ways of working, activities without formal requirements. From the 9 teams without formal requirements, 4 teams did not deliver or perform any of their assigned activities while 3 partially completed and 2 performed all activities. The other 9 teams had the activities as part of their Quality Gate check and for them 2 teams partly delivered and the rest performed all tasks as expected. These results indicate that the security activities should be formally part of the process, Quality Gates or any other way. Relying on ways of working or other informal processes (bottom up approach) to introduce security had in this study a very negative result.

In the examined agile process we identified 3 different teams that can affect the security of the end product. The software product managers (SPM), the development
project teams (DP) and latest stable version test team (LSV), see figure F.1 for a detailed description of their roles. For the practical study developers from the company and the authors agree on these activities for the different teams and their roles.

Figure F.7: The SPM role of the agile process. Security Requirements, modified User Stories and a General Risk Analysis have been added to the role.

It was learned during the practical study that the developers wanted a risk analysis during the early stages of development and to use it as a base for the security requirements. The purpose of the SPM level risk analysis is to identify general risks early and aid in the creation of security requirements. Without a risk analysis the developers had a hard time writing security requirements and needed aid. Two more problems were identified with the security requirements, the requirements were often of low priority and stayed in the prioritization stack for several sprints and security requirements caused by another requirement was not identified until after the source requirement had already been implemented.

The issue with low priority security requirements created problems with the developers as they felt the risk analysis was a waste of time since none of the findings were ever used. To increase the adoption of security requirements a separate security requirement list was created. Because the security requirements seldom had a direct business value they had a low priority but at the same time their estimated implementation time was lower. By separating them from functional requirements it became easier for the developers to add security requirements into sprints that had time to spare. The second issue with late security requirements was partly corrected with a risk analysis. Even more missing security requirements were however discovered by the DP teams during their risk analysis. This is not seen as a problem but instead the way agile development should work with incremental improvements. Instead of trying to solve all
the problems before implementation, the developers used the short sprints and incrementally added new security requirements as they were found and implemented them in the next sprint.

Role Matrices were tried by the developers but the lack of an initial template hindered the developers, since no role matrix had been created before it would take considerable time to implement one on an already mature product. Instead of creating a matrix, developers used the concept of defining the access and capabilities of a role and integrating them with the users story they provide to the DP teams. Each user stories had a specific role and made it very clear what roles should be allowed to perform the intended action and what roles were not.

![Figure F.8: The DP role of the agile process. For the planning game a Detailed Risk Analysis was added. During implementation, Static Code Analysis, Assumption Documentation, Coding Rules and Risk Based Code Reviews are added. While testing uses the results of the risks analysis and performs Risk Based Testing. Also the team is responsible for the Retrospective meetings to improve security.](image)

With waterfall processes developers spend a considerable time performing pre-studies and designing the product before they start implementing it. This is a stark difference from the short planning game that the DP team performs at the start of a new sprint. It is therefore important that any security activity for planning and design is short and aided by the SPM teams. As such only a detailed risk analysis was successfully used by the DP team. The best results were performed by the teams that could reuse the risk analysis from the SPM’s and build on it with more details for their specific requirements/user stories and the tasks they identified for it.

The result from the risk analysis was used in several ways. The risk analysis could
directly create new tasks that the team time estimated and put as part of their sprint. Some risks were too large for the sprint and where sent back to SPM as possible security requirements. The risk analysis was also used to guide if a Risk Based Code Review was necessary for a task. By weighing the risks and having predefined risk steps it was easy for the developers to determine what parts of their implementation should undergo a Code Review. Reviewing all the code was seen as too obstructive and less effective since it became a daily chore instead of a special activity that was only performed on specifically vulnerable code. Lastly the risk analysis was used as the basis for the validation testing done by the DP team. Just as extra focus was put on the Code Review of the sensitive tasks the same was done with the testing and extra focus with negative test cases was put on the implementations that were deemed as riskful.

Static analysis was seen as very important and was integrated as part of the continuous integration process. The tool can however be verbose and reports many non-issues to the developers. It was therefore important to have clear goals and policies in place before the DP teams started using the tool. This is especially true if there are many legacy warnings that would overwhelm the developers. The teams with the most success measured their total amount of warnings and had a goal of reducing it by 10% each week. In the end it was decided that a few warnings were especially important and that no DP team could release to LSV if their code contained these warnings, these warnings were put as a Quality Gate. Assumption documentation was put as part of the DP teams whiteboards and the teams used it to write back to SPM any assumptions they had made during implementation. Some of these assumptions resulted in changes to the manual/user documentations while other created new requirements.

Figure F.9: The LSV role of the agile process. Dynamic Analysis and Penetrations Testing were added as extra responsibilities of the team. They could also be used to perform a Final Security Review.
Dynamic Analysis was first tried as part of the DP teams and was intended to be used as part of the continuous integration process, similar to the static analysis tools. However, due to the high load on the test system and the long process time it became impossible to perform the dynamic analysis in an automated way. As a solution the dynamic analysis was instead manually performed and analyzed by the LSV team. This was not ideal since any finding now had to be corrected as a bug report instead of a quick fix by the developer. However, the dynamic analysis ability to detect problems in multithreaded programs was seen as too great of a benefit to be excluded from the process and the LSV teams thought the tool significantly aided their nonfunctional verification. The LSV teams also got the responsibility of performing Security Testing on the product. This included using fuzzy testing tools with the test suite to detect vulnerabilities. The fuzzing tools itself could not be part of the continuous integration process since it created too much data that was hard to evaluate and took a long time. Instead the LSV teams used the tool as part of their penetration test and focused on a specific requirement instead of the entire product. The final security review was a last Quality gate task that verified that all previous security gates had passed correctly.

F.7 Security activities

Table F.1: Number of waterfall security activities used, modified or not used within an agile setting.

<table>
<thead>
<tr>
<th></th>
<th>Requirement</th>
<th>Design</th>
<th>Implementation</th>
<th>Test</th>
<th>Release</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Modified</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>Not used</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

Table F.1 shows how many of the different security activities that were integrated in the final agile process. In all 31 activities that were investigated originally developed within a waterfall model using Cigatel Touchpoints, Common Criteria and Microsoft Security Development tools. 6 of these activities (19%) were directly adapted to the agile settings and 11 activities (35%) were not used at all. The remaining 14 activities (45%) were modified, i.e. partly used and/or two or more activities combined. In all, the modified versions represent 7 different agile activities. So, out of 31 waterfall activities, 20 are used forming 13 different agile activities. On average each agile activity consists of 2.2 modified waterfall activities. Also, 3 out of 7 modified agile activities were, at least partly, moved to another phase. For all original activities they
remained in the same phase as in the waterfall model when transferred to the agile roles. Figures F.2 - F.6 show which of the activities where used with or without modification or not at all.

The further analysis must pay attention to two things, first there is no general waterfall method using 31 different activities. There is an overlapping between different tools and also different views of what to investigate. The activities were compiled without the intention to show an alternative waterfall process but to include as many activities as possible in a different agile setting. Second, an agile setting does not need to include all waterfall phases, i.e. requirement, design, implementation, test and release. In the industrial context used here, based on SCRUM and XP, these phases are partly absent or intermixed with each other. Agile roles are both faster (less time for detailed security analysis) and more complex (more interacting between phases).

As already described, agile security issues are integrated within 3 roles, performing parts of the original waterfall phases. SPM includes requirement, design phases and release, DP includes design, implementation, test and release and finally LSV includes test and release.

For the waterfall requirement phase, only security requirements were used as described in existing documentation. While role matrices, abuse cases and quality gates had to be altered or moved in the process to better fit with an agile development process. None of the security activities from the design phase ended up being used as intended in the agile process. Instead the risk analysis was divided into two parts, one general risk analysis that SPM is responsible for and a detailed risk analysis for the DP. Contrary to the Design phase almost half of the activities from waterfall implementation phase could be used as is in the agile process. With static analysis and coding rules used unchanged while Code Reviews could be used if modified to better fit a quick iteration process.

The security activities from the test phase overlap, causing some of them to be excluded not because of their lack of compatibility but because similar method was used instead.

In the release phase External Review is too costly, retrospective meetings are used in its original setting by LSV and other modified activities are used by both LSV and DP. All together the five processes within the waterfall model are covered by the agile model, where all but the design phase are handled by two different teams.

### F.7.1 Excluded activities

After the interviews and practical study there were a few activities that developers decided not to use, either because they could not integrate the activity smoothly or
because they got the impression the activity did not provide suitable benefit for the cost of using it. We have grouped these in three categories:

I  Activities that go against the agile way of working (5 activities)

II  Activities that are covered by other activities or the process itself and were therefore not practically evaluated (3 activities)

III  Activities that the developers believed did not provide enough benefit and do not have agile replacement (3 activities)

From the security engineering requirement phase the Agree in Definitions (I) activity was seen as to plan heavy that forced requirement engineers to make decisions that should be made by the programmers instead. However, this means that there is a risk that the DP teams do not implement the requirements the way SPM intended. This is partly mitigated by the short development cycle and stand-up meetings where SPM and DP communicate frequently. The Cost Analysis (II) was seen as a part of a Risk Analysis and not an individual task.

The design phase did not have a single original activity left in the agile process and three activities ended up getting removed completely. The Requirement Inspection (II), a validation of all generated requirements, was removed because it added extra validation instead of implementation. This makes sense in an agile process that has a three week development cycle (sprint). Attack Surface Reduction (I) (I) was seen as an activity that developers could perform when needed, just as refactoring. Therefore the developers did not believe it should be part of a process. The risk is that developers will fall into a feature rush and never perform any Attack Surface Reduction (I). It is also much harder to do after code has been written as it will require a rewrite compare to perform it during the design. Identify Threats in Requirements (II) was also seen as not necessary for the agile process.

During implementation the developers rejected the concept of writing specific Security Tools (I) to test and verify security issues. This is in line with the agile motto of interaction before tools. However, because of the lack of tools it will be hard to automatically do the verification of certain security issues and there is no guarantee that future interaction between developers will remember the issues. Pair Programming (III) on the other hand is recommended by some agile processes. Its security benefit could be the same as the bug preventing benefit of having two developers look at the same code. However, developers rejected it due to its high cost of using two developers instead of one.

The testing phase also had large cuts just as the design phase. Red Team Testing (I) that relies on outside sources was deemed too intrusive to the process. The Test Plan
(III) on was removed because it added documentation instead of interaction between developers. This increases the risk that nonfunctional security test are not performed in future iterations. However, if the testers and developers do not raise an issue during the planning game then the risk might have subsided and the nonfunctional security testing can focus on new issues. The External Code Review (III) during the testing phase should have been performed by an outside team that verifies the integrity of the implementation. Without this verification the code integrity only relies on the team that implements it and any automated tools they use.

For the release phase it was External Review (I) activity that was completely removed. Without external sources that examine the release the security state relies completely on the knowledge and precision of the development team. For bugs and feature verification it is argued that the agile processes quick release and customer interaction leads to stable software. However, there is no research that supports that this also applies to security vulnerabilities. Instead, there is research suggesting that vulnerabilities are not detected or behave as functional bug detection. Meaning that this benefit probably does not improve the software security state.

F.8 Discussion

In the studied industrial case, software development projects involve different phases using external tools and methods for improving quality and security aspects within originally a waterfall model. The examined tools and methods are not developed specifically for an agile environment. In a former paper we put together all found security related activities used by any of the three development process, in all 31 activities (see section F.3) distributed among five different phases; requirement, design, implementation, test and release. Some of the activities were repeated or they used different methods for solving the same problem, so not all activities are relevant to use. The issue is instead to find the appropriate combination regarding security activities. Moreover this should be done in an agile environment, i.e. with restricted time and limited overall resources for the teams involved.

For all the activities, skilled agile developers were asked to weight each activity for effectiveness and ease of use within a specific phase. This relies on the structure used within the waterfall model making it possible to validate these activities against a real industrial environment. The drawback is that a positive validation for an activity within one phase does not necessarily mean that this is a supported agile activity because all phases are not within a certain project. Integration needs to be tested within real projects, preferably with a formal requirement within the teams to perform the activities. The results can be seen in figures F.2 to F.6 and the developers comment can be
## Table F.2: The security engineering activities in the left column are mapped to their original or modified security activity in the agile process and the role that should perform the activity. Activities numbered 1-6 are used unmodified and can be considered as original activities. Activities numbered 7-13 are modified in a significant way and had to be adapted to better integrate with the agile process.

|-----------------|-------|------------------------|---------------------------|----------------------|------------------|----------------|----------------|------------------------|---------------------|

**Chapter F: Integrating security engineering in an agile process**
read in section F.5. From the 31 activities, 6 were removed and not used in the practical study.

In the practical study, section F.6, the waterfall activities where examined in an agile setting. The incorporation of each activity is discussed in section F.7. But from the 25 activities that where examined practically, 6 were used without adapting them to agile and 5 new activities were excluded (see section F.7.1) from the process. 14 activities were modified to fit the process.

We identified three different teams that should improve the security of the end product, figures F.7 to F.9. All these teams do not necessarily follow the phases of the waterfall model or dedicate certain activities to such a phase, some activities may be rejected as too time consuming or being too restricted in a simplified agile setting. So instead of having a pro-active waterfall model the agile activities are reactive, i.e. development instead of trying to solve very complex problems before they understand the product.

The agile process is good at dividing and conquering the known problems in the software and focusing software development towards the real business needs. The short iterations make it easy to accept changes and re-prioritize the requirements. However, secure code development is more about coding principles and process activities that lead to better security, often depending on traditional top-down, document-centric approaches which are agile unfriendly. In the same way agile cannot help enforce code style, it won’t help with secure coding guidelines but it will hamper the integration of mature security activities. This was most clearly seen in the design activities that often relied on a finished product design that does not exist in the agile process. It therefore become very hard to reuse existing security practices and instead requires rethinking security for the agile process. The 20 activities that had passed the interview study and the practical were then condensed to 13 agile activities as shown in table F.2.

F.9  Conclusions

In the studied industrial case, agile software development projects involve different phases using external tools for improving quality and security aspects within originally a waterfall model. The examined tools, Cigatel Touchpoints, Common Criteria and Microsoft Security Development Lifecycle Process, are neither customized for security issues nor developed for an agile environment. Instead three different teams within an agile project were investigated by performing interviews; the software product managers (SPM), the development project teams (DP) and latest stable version test team (LSV). Security integration needs to be tested within real projects, preferably with a formal requirement within the teams to perform the activities. We found that 7 out of 9 teams with formal requirements fulfilled the task completely were only 2 out of 9
teams without formal requirements did the same. This prompted us to include Quality Gates, a formal check, in the process even though developers believed it is too rigid for an agile process.

In all 31 activities were investigated that were originally used within a waterfall model. From these activities 11 were excluded either because they: (I) are not compatible with agile, (II) the purpose of doing the activity has no meaning in agile and (III) the activity does not, according to the developers in this study, provide enough benefit to interrupt the process even if there is no replacement for it. At the other end of the spectrum, 6 activities (Table F.2 activities 1-6) were used as explained in the Security Engineering process. These activities did not require any modification or rearrangement when they should be performed. The remaining 14 activities were modified, i.e. partly used and/or two or more activities combined. In all, the modified versions represent 7 different agile activities (Table F.2 activities 7-13). So, out of 31 waterfall activities, 20 are used forming 13 different agile activities. From the 11 activities that were excluded the developers concluded that 5 of them provided a benefit to security but were hard to integrate while the remaining 6 either provided too small of a benefit or the developers believed the agile process will handle the risk without the activity. Moreover, we found SPM covering requirement and design, DP design, implementation, test and release and finally LSV test and release.

The major difference between a waterfall and an agile model is when corrections are done. Instead of having a pro-active waterfall model the agile activities are reactive, i.e. prioritize development instead of trying to solve very complex problems before they happen.

F.10 Future works

One problem with the reactive view of agile projects is that some security issues may never appear during the repeated process of security review, i.e. the in-depth analysis that exists in the waterfall project disappears. To investigate, we need to run a full product cycle using an agile development including all security issues proposed for the agile environment. Applicable parts should then be compared with older projects based on the waterfall model. Another objection is that this study was done on a specific industrial setting, i.e. it may be difficult to extend the results to other areas. This is particularly true for those agile applications not based on SCRUM so future studies need to be done outside this area.
References

Almossawi, L. and Sinha. Analysis tool evaluation: Coverity prevent.


REFERENCES


Guo, P. J. and D. Engler. Linux kernel developer responses to static analysis bug reports.


Krsul, I. (1998a, may). *Software vulnerability analysis*. Ph. D. thesis, Computer Sciences Department, Purdue University, Lafayette, IN, USA.


REFERENCES


REFERENCES


List of Figures

2.1 Difference between design and actual implementation. Circle A, the design, does not always overlap with Circle B, the actual implementation. .................................................. 4
2.2 Waterfall Development process at Ericsson AB. ................................. 5
2.3 Cigital Software Security Touchpoints development process. ............... 6
2.4 Origin and propagation of vulnerabilities. ........................................... 9

3.1 The thesis first research question and its three sub-question. .............. 18
3.2 The thesis second research question and its four sub-question. .......... 18
3.3 The thesis third and final research question and its three sub-question. 19
3.4 An overview model of the agile processes used by the examined projects. Three distinct roles (SPM, DP and LSV) are relevant to our study and shown in the model. .......................... 25

A.1 Metamodel of the Extension to Attack Trees .................................. 45
A.2 Example of a Countermeasure Graph ............................................. 46
A.3 Cumulative Hierarchical Voting ..................................................... 48
A.4 Combination of Effectiveness with Cost ......................................... 50
A.5 Cumulative Hierarchical Voting ..................................................... 51
A.6 Countermeasure Graph for Open Source Game .............................. 51
A.7 Combination of Effectiveness with Cost ......................................... 53

B.1 Metamodel of the Extension to Attack Trees .................................. 61
B.2 Example of a Countermeasure Graph ............................................. 62
B.3 Cumulative Hierarchical Voting ..................................................... 64
B.4 Combination of Effectiveness with Cost ......................................... 65
B.5 Action Research Approach ........................................................... 68
## LIST OF FIGURES

| B.6       | Metamodel of the Countermeasure Graph | 76  |
| B.7       | Open Source tool that implements the Countermeasure Graph | 77  |
| C.1       | Independent and dependent variables | 84  |
| C.2       | Correct classifications per warning type | 89  |
| C.3       | Correct answers per confident group | 89  |
| C.4       | Classifications grouped by the developers general experience | 90  |
| C.5       | Classifications based on two specific experience groups | 91  |
| C.6       | Correct classifications from developers with a combined specific experience | 91  |
| D.1       | Total amount of reported security warnings grouped according to the used taxonomy. Input validation and representation contains the largest amount of warnings. Product D was selected for future studies because of its even distribution of warnings | 111 |
| E.1       | A quick overview of the Agile development process | 136 |
| E.2       | Above/below zero average for different requirements measured as cost and benefit | 140 |
| E.3       | Above/below zero average for different design issues measured as cost and benefit | 141 |
| E.4       | Above/below zero average for different implementations measured as cost and benefit | 141 |
| E.5       | Above/below zero average for different testing issues measured as cost and benefit | 142 |
| E.6       | Above/below zero average for different release issues measured as cost and benefit | 142 |
| F.1       | An overview model of the agile processes used by the examined projects. Three distinct roles (SPM, DP and LSV) are relevant to our study and shown in the model | 159 |
| F.2       | Developers comparative weighting on requirement phase security activities. Divided into three groups depending on how the activities were later used in the practical study. See section F.4.3 for clarification of the weighting process | 162 |
| F.3       | Developers comparative weighting on design phase security activities. Divided into three groups depending on how the activities were later used in the practical study. See section F.4.3 for clarification of the weighting process | 163 |
F.4 Developers comparative weighting on implementation phase security activities. Divided into three groups depending on how the activities were later used in the practical study. See section F.4.3 for clarification of the weighting process. .................................................. 164

F.5 Developers comparative weighting on testing phase security activities. Divided into three groups depending on how the activities were later used in the practical study. See section F.4.3 for clarification of the weighting process. .................................................. 165

F.6 Developers comparative weighting on release phase security activities. Divided into three groups depending on how the activities were later used in the practical study. See section F.4.3 for clarification of the weighting process. .................................................. 166

F.7 The SPM role of the agile process. Security Requirements, modified User Stories and a General Risk Analysis have been added to the role. .................................................. 167

F.8 The DP role of the agile process. For the planning game a Detailed Risk Analysis was added. During implementation, Static Code Analysis, Assumption Documentation, Coding Rules and Risk Based Code Reviews are added. While testing uses the results of the risks analysis and performs Risk Based Testing. Also the team is responsible for the Retrospective meetings to improve security. .................................................. 168

F.9 The LSV role of the agile process. Dynamic Analysis and Penetrations Testing were added as extra responsibilities of the team. They could also be used to perform a Final Security Review. .................................................. 169
List of Tables

A.1 Goals and Their Damage ................................................ 52
A.2 Agents and Their Threat ............................................... 52
A.3 Attacks ................................................................. 53
A.4 Countermeasures ....................................................... 54

D.1 Taxonomy used for classification. ................................. 103
D.2 Initial classification of total amount of reported warnings, the number of reported false positives by the developers, the number of security warnings and the number of known bug reports that were also reported by the static analysis tool. .................... 107
D.3 Detected vulnerabilities caused by lack of input validation and representation problems. Left column name refers to the static analysis checker used to detect the vulnerability. ....................... 108
D.4 Detected vulnerabilities caused by breaches in the API contract between a caller and a callee. Left column name refers to the static analysis checker used to detect the vulnerability. ....................... 108
D.5 Detected vulnerabilities caused by concurrency or other time based faults. Left column name refers to the static analysis checker used to detect the vulnerability. ....................... 109
D.6 Detected faults caused by bad code quality. Left column name refers to the static analysis checker used to detect the vulnerability. ....................... 109
D.7 Developers capability to identify and classify false positives. The dark grey column, False positives, is the correct answer. ....................... 114
D.8 Developers capability to identify and classify vulnerabilities. The dark grey column, Vulnerability, is the correct answer, while False positive is a very dangerous answer as these would be ignored. ....................... 115
D.9 Developers correction of false positives. Ideally all developers should ignore, grey column, all of these warnings. Harmless corrections alter code but do not affect the software, while Harmful corrections would harm the software. Warnings are sorted based on the severity of the Harmful correction, with the most harmful at the bottom.  

D.10 Developers correction of vulnerabilities, Only the gray column, Secure correction, can be consider an acceptable answer. Ignored were not correct because developers have previously classified the warnings as false positivist and unsafe correction do not address the vulnerability.  

D.11 Form for Classification of Warnings.  

E.1 Grading security issues within a project cycle. M - Microsoft SDL, T - Cigatel touchpoints, CC - Common Criteria and O - Others  

E.2 Agile Security a basic Agile process with the most compatible SE activities.  

F.1 Number of waterfall security activities used, modified or not used within an agile setting.  

F.2 The security engineering activities in the left column are mapped to their original or modified security activity in the agile process and the role that should perform the activity. Activities numbered 1-6 are used unmodified and can be considered as original activities. Activities numbered 7-13 are modified in a significant way and had to be adapted to better integrate with the agile process.
Software Engineering might be science; but that’s not what I do. I’m a hacker, not an engineer.

—Jamie Zawinski
ABSTRACT

Background: Software developers are facing increased pressure to lower development time, release new software versions more frequent to customers and to adapt to a faster market. This new environment forces developers and companies to move from a plan based waterfall development process to a flexible agile process. By minimizing the pre development planning and instead increasing the communication between customers and developers, the agile process tries to create a new, more flexible way of working. This new way of working allows developers to focus their efforts on the features that customers want. With increased connectability and the faster feature release, the security of the software product is stressed. To develop secure software, many companies use security engineering processes that are plan heavy and inflexible. These two approaches are each others opposites and they directly contradict each other.

Objective: The objective of the thesis is to evaluate how to develop secure software in an agile process. In particular, what existing best practices can be incorporated into an agile project and still provide the same benefit if the project was using a waterfall process. How the best practices can be incorporated and adapted to fit the process while still measuring the improvement. Some security engineering concepts are useful but the best practice is not agile compatible and would require extensive adaptation to integrate with an agile project.

Method: The primary research method used throughout the thesis is case studies conducted in a real industry setting. As secondary methods for data collection a variety of approaches have been used, such as semi-structured interviews, workshops, study of literature, and use of historical data from the industry.

Results: The security engineering best practices were investigated though a series of case studies. The base agile and security engineering compatibility was assessed in literature, by developers and in practical studies. The security engineering best practices were group based on their purpose and their compatibility with the agile process. One well known and popular best practice, automated static code analysis, was toughly investigated for its usefulness, deployment and risks of using as part of the process. For the risk analysis practices, a novel approach was introduced and improved. As such, a way of adapting existing practices to agile is proposed.

Conclusion: With regard of agile and security engineering we did not find that any of the investigated processes was agile compatible. Agile is reaction driven that adapts to change, while the security engineering processes are proactive and try to prevent threats before they happen. To develop secure software in an agile process the developers should adopt and adapt key concepts from security engineering. These changes will affect the flexibility of the agile process but it is a necessity if developers want the same software security state as security engineering processes can provide.